8^{th} International Workshop User Interfaces for Theorem Provers

(UITP'08)

TPHOLS'08 Satellite Workshop

Friday, 22nd August 2008

Montréal, Québec, Canada

– Informal Workshop Proceedings –

Preface

The User Interfaces for Theorem Provers workshop series brings together researchers interested in designing, developing and evaluating interfaces for interactive proof systems, such as theorem provers, formal methods tools, and other tools manipulating and presenting mathematical formulas.

While the reasoning capabilities of interactive proof systems have increased dramatically over the last years, the system interfaces have often not enjoyed the same attention as the proof engines themselves. In many cases, interfaces remain relatively basic and under-designed.

The User Interfaces for Theorem Provers workshop series provides a forum for researchers interested in improving human interaction with proof systems. We have solicited contributions from the theorem proving, formal methods and tools, and HCI communities, both to report on experience with existing systems, and to discuss new directions. The topics covered by the workshop include, but are not limited to:

- Application-specific interaction mechanisms or designs for prover interfaces
- Experiments and evaluation of prover interfaces
- Languages and tools for authoring, exchanging and presenting proof
- Implementation techniques (e.g. web services, custom middleware, DSLs)
- Integration of interfaces and tools to explore and construct proof
- Representation and manipulation of mathematical knowledge or objects
- Visualization of mathematical objects and proof
- System descriptions

UITP 2008 is a one-day workshop held on Friday, August 22nd 2008 in Montréal, Canada, as a TPHOLS'08 workshop. Eight papers have been selected by the international programme committee for presentation at the workshop. Additionally, we have two invited system demonstrations, one by Matt Kaufmann and J Strother Moore on interface aspects of the ACL2 theorem proving system and one by Sam Owre on the PVS user interfaces. Finally, Deborah McGuinness demonstrates an explanation infrastructure for TPTP proofs and solicits feedback.

We cordially thank the following programme committee members for their valuable work and support:

– David Aspinall, University of Edinburgh, Scotland
– Yves Bertot, INRIA Sophia-Antipolis, France
– William Billingsley, University of Cambrigde, England
– Paul Cairns, University College London, England
– Ewen Denney, NASA Ames Research Center, USA
– Herman Geuvers, Radboud University Nijmegen, The Netherlands
– Christoph Lüth, University of Bremen and DFKI Bremen, Germany
– Michael Norrish, NICTA, Australia
– Claudio Sacerdoti-Coen, University of Bologna, Italy
– Gem Stapleton, University of Brighton, England
– Geoff Sutcliffe, University of Miami, USA
– Makarius Wenzel, TU Munich, Germany

Finally, we thank the organizers of TPHOLs'08, Sofiène Tahar, Otmane Ait-Mohamed and César Muñoz, for their collaboration and help with respect to the organization of UITP'08 in Montreal, Canada. In the managing of the whole reviewing process, Andrei Voronkov's EasyChair conference management system proved itself an excellent tool.

August 2008

Serge Autexier and Christoph Benzmüller
(DFKI Bremen and Saarland University)
Co-chairs of UITP'08

List of Contributions

An interactive driver for goal directed proof strategies

Andrea Asperti and Enrico Tassi

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY
asperti@cs.unibo.it tassi@cs.unibo.it

Abstract

Interactive Theorem Provers (ITPs) are tools meant to assist the user during the formal development of mathematics. Automatic proof searching procedures are a desirable aid, and most ITPs supply the user with an extensive set of facilities to improve automation. However, the black-box nature of most automatic procedure conflicts with the interactive nature of these tools: a newcomer running an automatic procedure learns nothing by its execution (especially in case of failure), and a trained user has no opportunities to interactively guide the procedure towards the solution, e.g. pruning wrong or not promising branches of the search tree. In this paper we discuss the implementation of the resolution based automatic procedure of the Matita ITP, explicitly conceived to be interactively driven by the user through a suitable, simple graphical interface.

Keywords: Interactive theorem proving, SLD resolution, automation

1 Introduction

Most of the development effort behind Interactive Theorem Provers is devoted to bridge the gap between the high level language used by humans for reasoning and communicating mathematics, and the low level foundational language understood by ITPs. Among all facilities offered by ITPs, a high degree of automation is certainly desirable and several works (see for example [12,11]) have been devoted to the integration of automatic proof search facilities in interactive theorem provers. The machinery employed in this integration is usually hidden to the user: when the automatic procedure finds a proof the interactive theorem prover usually evaluates the trace left by the prover (if any) and converts it, possibly using some reflection mechanism (see [5,6]), to a proof in its foundational dialect. What is neglected by this traditional approach is the interactive nature of the tool. The user has no feeling of what is going on, why the automatic procedure has possibly failed and how he can possibly improve the situation. Moreover, when used in a didactical environment where untrained users are put in front of an interactive theorem prover, it is desirable to let them use automation facilities freely, but providing them the

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

possibility to understand the work done by the automatic procedure or the reasons of its failure.

The aim of this work is to develop a reasonably fast SLD [13,14] based proof searching procedure for the interactive theorem prover Matita [3] that is completely transparent to the user, allowing him to follow the execution of the procedure and to drive it, taking run-time decisions on how the procedures explores the search space. As a side effect we obtain a very handy debugging tool, that proved to be extremely useful to tune and fix the procedure.

To get this result, we develop a SLD engine that performs backtracking without relying on the call stack (i.e. not using stack frames as choice points). This characteristic, together with a carefully chosen selection function, allow us to effectively present to the user a view of the ongoing computation.

2 The proof searching procedure

The way proofs are built in Matita is by instantiation. The foundational dialect of the interactive theorem prover (namely the Calculus of Inductive Constructions [9,16]) is extended with meta-variables [15] (written $?_i$) whose type represents a missing part of the proof, called *goal*.

Definition 2.1 [Proof problem] A *proof problem* $\mathcal{P}$ is a finite list of typing judgement of the form $\Gamma \vdash ?_j : T$ where for each metavariable $?_i$ that occurs in the context Γ and type T there exists a corresponding entry in $\mathcal{P}$.

Each proof step generates a substitution instantiating one or more existing metavariables, whose entries are also removed from $\mathcal{P}$, and possibly adding new entries (new open goals) to $\mathcal{P}$.

Definition 2.2 [Substitution] A metavariable substitution Σ is a list of couples metavariable-term.

$$\Sigma = [?_1 := t_1; \ldots; ?_n := t_n]$$

Substitutions are usually performed lazily, thus the status of the ongoing proof comprises both a proof problem and a substitution. We will call such a pair a *proof status*.

For example, the initial status of the just declared conjecture $\forall x, y : \mathbb{N}.P(x, y) \to Q(x, y)$ will be

$$[] \vdash ?_1 : \forall x, y : \mathbb{N}.P(x, y) \to Q(x, y)$$

together with an empty substitution. After performing hypothesis introduction it will change to

$$x, y : \mathbb{N};\ p :\ P(x, y) \vdash ?_2 : Q(x, y)$$

together with a substitution $\Sigma = [?_1 := \lambda x, y : \mathbb{N}.\lambda p : P(x, y).?_2]$.

The application of a substitution Σ to a term t is denoted with $\Sigma(t)$. This operation is extended to contexts and proof problems, substituting all the types of abstracted variables (in the context) or the types of of missing proofs (in the proof problem).

A proof is over when there are no more proof problems in the proof status, and the proof of the original conjecture can be obtained applying the substitution to the initial metavariable.

The proof searching procedure we implemented in the interactive theorem prover Matita is essentially inspired by SLD resolution [14]: it iterates applications of known results following a depth-first strategy (up to a given depth). No introduction of new hypothesis is done (that amounts to assume to have a horn-like base of knowledge, as it is often the cases), hence the context of the proof remains unchanged during the execution of the procedure.

The classical rule for SLD resolution follows.

SLD

$$\frac{\leftarrow A_1, \ldots, A_n \qquad H \leftarrow B_1, \ldots, B_m \qquad \Sigma = mgu(H, A_i)}{\leftarrow \Sigma(A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)}$$

CIC is a dependently typed, higher order, language where no most general unifier can be found in the general case. Nevertheless, an essentially first order unification heuristic is implemented as part of the so called *refiner*[1] and largely used in the process of building proofs. A detailed description of the unification algorithm implemented in Matita can be found in [18] and some recent extensions are described in [19].

Definition 2.3 [Unification] The process of unifying two terms is denoted with

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash N \stackrel{?}{\equiv} M \ \stackrel{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma'$$

Unification performs only metavariables instantiations, and the resulting Σ' is such that $\Sigma'(N)$ is convertible (that for CIC means equal up to $\beta\iota\delta\zeta$-reduction) with $\Sigma'(M)$ in context $\Sigma'(\Gamma)$ and proof problem $\Sigma'(\mathcal{P}')$.

The SLD resolution rule is implemented in Matita as the *apply* tactic. Since it is meant for interactive usage, both the selection and computation rule are left to the user: in the following presentation the goal i and the clause (lemma) c are user provided. The outcome of the tactic is a proof status or an exception if the unification step fails.

[1] The *refiner* is the component implementing type-inference, as opposed to the *kernel*, implementing type-checking. It is in charge to automatically fill the proof with a lot of negligible information easily inferred by the context. See e.g. [2] for an architectural outline of Curry-Howard based ITPs.

Apply-tac

$$\frac{\begin{array}{l}\mathcal{P} = \Gamma_1 \vdash ?_1 : A_1, \ldots, \Gamma_n \vdash ?n : A_n \\ \mathcal{P}' = \mathcal{R}(\Gamma \vdash ?_{B_1} : B_1, \ldots \Gamma, x_1 : B_1, \ldots, x_{m-1} : B_{m-1} \vdash ?_{B_m} : B_m); \mathcal{P} \\ \Gamma \vdash c\ ?_{B_1}\ \ldots\ ?_{B_m} : H \\ \mathcal{P}',\ \Sigma,\ \Gamma \vdash H \stackrel{?}{\equiv} A_i \overset{\mathcal{U}}{\rightsquigarrow} \mathcal{P}'',\ \Sigma' \\ \Sigma'' = ?_i := c\ ?_{B_1}\ \ldots\ ?_{B_m}; \Sigma'\end{array}}{(\mathcal{P}'', \Sigma'')}$$

With $\Gamma \vdash t : T$ we denote the typing judgement assigning to t the type T in the context Γ. The reordering function $\mathcal{R}$ is applied to the list of new goals, and as we will see in Section 2.1 it allows to implement some heuristics to increase performances and avoid the proliferation of meaningless goals.

Note that unifying H with A_i can in general instantiate some $?_{B_i}$ but not generate new metavariables, thus the set of new goals opened by the apply tactic is a subset of $\{?_{B_1}, \ldots, ?_{B_n}\}$.

Our final goal is to provide the user a tool to observe the automatic procedure running and possibly drive it without stopping it. To do that, we have to make sure that some parts of the computation are reasonably stable, such that the user has enough time to read them before they change. If it was not possible, the user would have to stop the execution and make it advance step by step, inherently loosing the speed modern computers have, or alternatively not use the tactic interactively (just let it run).

To achieve a reasonably stable view of the ongoing computation, we had to adopt a leftmost, depth first, selection rule. The selection function is fixed and always chooses the first goal, in the same spirit of Prolog. The proof the procedure is building up can be seen as some sort of tree: an application of the resolution rule generates a node with a new son for every newly generated goal, and proceeds trying to prove all of them. If one fails it backtracks changing the node (if there are alternative clauses that can be applied). If we assume to have n applicable clauses and a depth limit d, a node at depth i is updated every $(d - i - 1)^n$ iterations, granting a reasonable stability for shallow nodes.

An alternative search strategy, like for example the discount algorithm [17], that generates and continuously refines a set of proved (intermediate) results, would not have worked. What a user needs to know to understand what a discount based automatic prover is doing is the set of intermediate lemmas proved so far. This set is usually really huge and continuously changing: new results are added, weaker results are removed in favour of more general ones, all results are simplified (put in a canonical form) using newly generated equations.

2.1 The reordering function

To understand why reordering newly generated goals can increase performances, and also avoids generating many pointless goals, consider the division operation between natural numbers and the associated predicate *divides*. A natural number q

divides n if there exists a p such that $n = q * p$. In a dependently typed λ-calculus equipped with inductive types, a natural [2] definition for that predicate would be an inductive predicate with a single constructor

$$witness : \forall p, q, n : \mathbb{N}.p * q = n \rightarrow q|n$$

This lemma (actually a constructor), when applied, generates two new goals: $?_p$ of type $\mathbb{N}$ and $?_H$ of type $?_p * q = n$. Attempting to solve $?_p$ first is a bad idea since we have no real information on $?_p$ except that it is a natural number, while we know more information concerning the second goal, for example that it involves the multiplication operation. This piece of information can be exploited by the computational rule to search for applicable clauses. Moreover, almost every solution to goal $?_H$ also forces $?_p$ to be some fixed natural number.

Interactive theorem provers are tools used to create libraries of formalized theorems; as a consequence the environment from which the computation rule may choose a lemma to apply is extremely polluted. In case of goals of just type $\mathbb{N}$, it could even choose to apply the Fibonacci function and then successively try to guess an input such that the second goal can be solved, possibly backtracking and guessing another input for the Fibonacci function. The ability of the CIC logic to compute is very handy in general, but is cases like this one may lead to very long computations.

2.2 The computation rule

The computation rule has to find a clause (in our case an existing lemma), or better a list of clauses, that will be applied in order to solve a given goal. ITPs are equipped with large libraries of already proved results, thus some searching facilities have to be employed to select a reasonably small amount of lemmas that will then be effectively applied. Matita has many built-in searching facilities, extensively described in [1], that can search local and remote libraries for results relevant to a given goal. These facilities are used to fill in an in-memory trie [3] data structure together with some parts of the library the user can declare to be pertinent to what he is doing. On top of this structure a pretty efficient unification approximation can be performed, resulting in a set of lemmas that is later refined using the real unification algorithm.

Since we want to present the user only good alternatives, the computational rule has not only to find good candidates, but also to attempt to apply them, directly pruning false positives. Moreover, suddenly applying all found lemmas allows to sort these alternatives looking for example to the number of newly opened goals. The cands function performs this search and returns a list of alternative proof statuses.

Definition 2.4 [Candidates (of the environment E)] Let g be a goal, $\mathcal{P}$ a proof problem and Σ a substitution environment. Let $\Gamma \vdash ?g : T \in \mathcal{P}$. The function cands applied to a proof status ($\mathcal{P}$, Σ) and a goal g returns a list of tuples

[2] An alternative definition, using the computational fragment of CIC to define the division operation and proving some properties of that function is also possible, but not widely adopted.

[3] A trie is a tree of prefixes, a good compromise between search speed and space consumption adopted, in some of its variants, by many automatic provers.

$(\Sigma', \mathcal{P}', [g_1; \ldots; g_n])$ such that:

- $t \in E$
- $\Gamma \vdash t : \forall x_1 : T_1 \ldots \forall x_n : T_n . T'$
- $\mathcal{P},\ \Sigma,\ \Gamma \vdash T \stackrel{?}{\equiv} T' \ \stackrel{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma'$
- $\Gamma; x_1 : T_1; \ldots; x_{i-1} : T_{i-1} \vdash ?g_i : T_i \in \mathcal{P}' \quad \forall i \in \{1, \ldots, n\}$
- $?_g := (t\ ?g_1\ \ldots\ ?g_n) \in \Sigma'$

2.3 Backtracking

The cands function finds a set of relevant lemmas in the global environment (the library of already proved results) and using the **Apply-tac** rule attempts to apply them to a given goal, returning the list of proof statuses relative to successful applications of that rule. On top of that, an automatic proof searching procedure can easily be implemented by means of two mutually recursive functions.

For each goal to be solved (gl), the function *search* calls the computation rule (implemented by the cands function) that finds a list of lemmas and that uses the **Apply-tac** rule to obtain the list of associated proof statuses (cl). Then it tries to find if one of the resulting proof statuses can be solved, using the *first* function, that recursively calls *search*. If one succeeds, *search* moves to the next goal to be solved. A pseudo-OCaml code for that function follows. The choice of OCaml as the implementation language for the tactic is not arbitrary, since the whole Matita ITP is written in in that language.

```
let rec first f l = function
  | [] → raise Failure
  | hd::tl →
      try f hd
      with Failure → first f tl
let rec search gl (S, P) =
  match gl with
  | [] → S, P
  | g::tl →
      let cl = cands (S, P) g in
      let S',P' = first (fun (S, P, gl) → search gl (S, P)) cl in
      search tl (S', P')
```

The code is oversimplified, many checks are missing: for example there is no bound check, thus this function may diverge. Nevertheless, it is already enough to see the issue arising with this simple and elegant implementation of backtracking.

The problem with this approach is that informations needed to properly backtrack are kept by the OCaml stack. The try/with construct uses stack frames to "label" choice points in the derivation to which the function may backtrack. While this is in general an elegant solution, it can not be employed here, since we want to show the user the current computation, and OCaml (like most of compiled lan-

guages) does not provide enough introspection mechanisms to explore the current call stack.

To reach our objective we have to write a stack-less procedure (that is a tail recursive function). Before detailing such procedure we want to give an overview of the final result we obtained, showing the interface we offer to the user.

3 The graphical user interface

The proof searching procedure elaborates fast, but the depth-first proof searching strategy (that is, selecting always the first goal) makes the shallow part of the computation pretty stable. For that reason we adopted the viewport widget, that allows to display only a subpart of a larger picture, by default the most stable.

In Figure 1 the user interface to drive the automatic procedure is shown. On the background there is the main window of Matita, showing the current open conjecture (conjecture fifteen). The window is divided in three columns:

- the leftmost shows the progressive number of open conjectures, the number identifying the current goal and the depth left (the difference between the user defined bound and the actual depth);
- the column in the middle displays the i-th open conjecture, since it lives in the original context (displayed by the background window) there is no need to print again this information;
- the rightmost column lists all lemmas that can be applied to the conjecture. This column displays the so called choice stack [7], colouring in grey the applied lemma. Some additional information on these lemmas are displayed using tool tips. If a lemma is unknown to the user, its type can be shown holding the mouse on its name.

To attack conjecture fifteen the automatic tactic found a bunch of lemmas that can be applied. The former, witness, has already been applied and is thus coloured in grey. The list of grey items, read top to bottom, is the list of lemmas applied so far. All its alternatives are shown on its right. The application of the witness lemma to a goal of the form $n|m$ opens two conjectures: the former (number 52) is that for a certain $?_{51}$, $m = n*?_{51}$ and the latter (number 51) is the witness $?_{51}$ itself.

The user already sees the result of the reordering function $\mathcal{R}$, since newly opened goals have been sorted, preferring goal 52 to 51.

The next step performed by the automatic procedure is to find relevant lemmas for the conjecture displayed in the second line, place them in the rightmost column, grey the former and display the result of its application. In case one application fails, the next alternative is attempted. In case there are no alternatives left, the next alternative of the previous line it considered. Thus, if no lemmas can be applied to conjecture 52, both line one and two are removed together with the witness lemma that generated them and the lemma div_mod_spec_to_divides is applied.

The user can execute the tactic step by step with the next button, and switch between the running status and the paused one with the buttons pause and play. To drive the proof searching algorithm the user can interact with the lemmas in

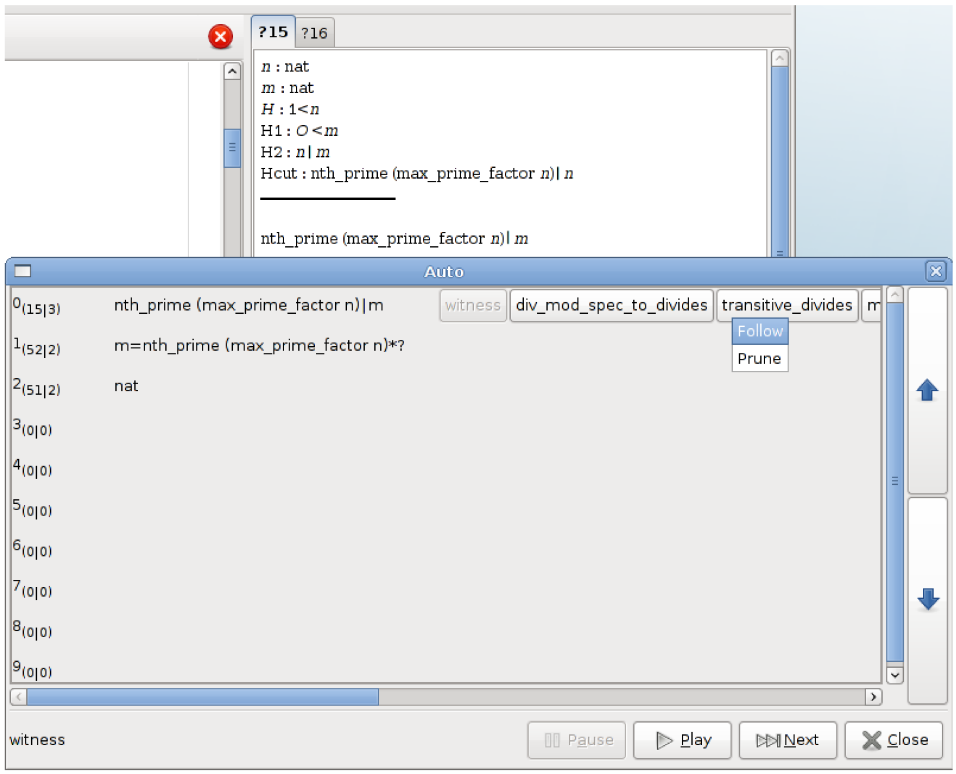


Figure 1. Auto interaction window

the rightmost column. In Figure 1 the user just clicked on the transitive_divides lemma, opening the list of allowed actions. The *prune* action simply removes the lemma for the list of alternatives, the *follow* action makes all alternatives before the one selected immediately fail.

The pair of big arrow buttons on the right allows to move the current viewport, focusing on goals that are examined by the proof searching procedure at depth greater than a fixed amount (ten in this case) in the search tree. The choice of using a viewport allows to cut out the deepest part of the computation, that is likely to change very frequently and not worth being displayed.

When a subgoal is solved, two possible scenario arise, depending if some metavariables are occurring in its statement or not. If some metavariables occur, the solution found may instantiate them in such a way that other goals in which such metavariables occur result false. In that case, the line corresponding to that goal is not removed, and the list of candidates associated to it remains visible and the user can interact with it. If the goal statement contains no metavariables the corresponding line is removed, since no choices relevant for the eventual success of the proof search procedure can be made by the user.

4 Operational description of the tactic

To present the user such a window, the search procedure has to be stack-less. All informations have to be accessible by the graphical user interface at any time. That means the procedure has to be a for-program (or a tail recursive function) keeping

the computation tree (and informations needed for backtracking) into a first order object and possibly pass it to the GUI.

To formally describe how the procedure works and the data structure used to represent the computation status we need to define the following objects.

Definition 4.1 [Proof of goal] Given a goal (metavariable number) g and a substitution Σ, the proof of g denoted with $\Sigma(g)$ is the least fixed point of $\Sigma(\cdot)$ starting from $?g$.

This function is not only used at the end of the tactic to build the proof object for the main conjecture, but also to create (and cache) the proof of intermediate results, avoiding to search twice the same proof.

Definition 4.2 [Metas of term] Given a term t the set of metavariables occurring in t is denoted with $\mathcal{M}(t)$.

As we already anticipated in the previous section, the procedure behaves differently if a metavariable occurs in a goal.

Definition 4.3 [Cache] A cache θ is a partial function from terms (actually types) to terms. Its domain can be extended with the operation $\theta[T \mapsto t]$. All terms in θ live in the same context.

We use the notation $\theta[T \mapsto \Sigma(g)]$ to update θ associating the proof of g with T. We use $\bot$ to represent failures, thus $\theta[T \mapsto \bot]$ extends θ with the information that T has no proof. The cache is an essential ingredient to obtain good performances and avoids many kinds of loops.

Definition 4.4 [Element] We call an element a triple of type (in OCaml notation) proof status $*$ op list $*$ goal list where goal is the type of metavariable indexes and op is the following algebraic type:

```
type op = D of goal | S of goal * term
```

The D constructor will decorate goals that still have to be processed (toDo), while S will decorate goals that have been successfully solved, and whose proof may be cached. The last component of an element is a failure list, containing all goals that have to be considered failed when the element itself fails (i.e. when the *op* list contains some D items that fail).

The last ingredient is the function to find lemmas that can be applied to a given goal, that is the function cands described in Section 2.2. The only needed modification is to make this function also return the applied lemma together with the proof status: this is needed to display the choice stack to the user. Note that cands can easily be extended to look for applicable lemmas not only in the global environment E but also in θ since all elements in θ live in the same context Γ of the goal (the proof searching procedure never alters Γ).

In Table 1 we define the *step* function mapping a list of elements and a cache to a new list of elements equipped with a possibly updated cache. This function is the core of the automatic procedure, and is applied until a Failure or Success status is reached. We use $\circ$ for list concatenation. The complete failure status is

$$(((\mathcal{P},\Sigma)\text{ as }P, S^t_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, tl, fl) :: el', \theta') \quad \text{(i)}$$
when $\mathcal{M}(T) = \emptyset$ and $\Gamma \vdash ?g : T \in \mathcal{P}$
where $\theta' = \theta[T \mapsto \Sigma(g)]$ and $el' = purge(el, tl)$

$$(((\mathcal{P},\Sigma)\text{ as }P, S^t_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, tl, fl) :: el, \theta) \quad \text{(ii)}$$
when $\mathcal{M}(T) \neq \emptyset$ and $\Gamma \vdash ?g : T \in \mathcal{P}$

$$(((\mathcal{P},\Sigma), D_g :: tl, fl) :: el, \theta) \xrightarrow{step} (((\mathcal{P},\Sigma'), tl, fl) :: el, \theta) \quad \text{(iii)}$$
when $\theta(T) \neq \bot$ and $\Gamma \vdash ?g : T \in \mathcal{P}$
where $\Sigma' = \Sigma \circ [?g := \theta(T)]$

$$(((\mathcal{P},\Sigma), D_g :: tl, fl) :: el, \theta) \xrightarrow{step} (el, \theta'_{m+1}) \quad \text{(iv)}$$
when $\theta(T) = \bot$ and $\Gamma \vdash ?g : T \in \mathcal{P}$
where $\theta'_1 = \theta$ and $fl = \{g_1; \ldots; g_m\}$
and $\Gamma_g \vdash ?g : T_g \in \mathcal{P}$ for $g \in \{1, \ldots, m\}$
and $\theta'_{g+1} = \theta'_g[T_g \mapsto \bot]$ for $g \in \{1, \ldots, m\}$

$$(((\mathcal{P},\Sigma), D_g :: tl, fl) :: el, \theta) \xrightarrow{step} (el, \theta'_{m+1}) \quad \text{(v)}$$
when $cands(P, g) = []$
where $\theta'_1 = \theta$ and $fl = \{g_1; \ldots; g_m\}$
and $\Gamma_g \vdash ?g : T_g \in \mathcal{P}$ for $g \in \{1, \ldots, m\}$
and $\theta'_{g+1} = \theta'_g[T_g \mapsto \bot]$ for $g \in \{1, \ldots, m\}$

$$((P, D_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P'_1, l_1@tl, []) :: \ldots :: (P'_m, l_m@tl, g :: fl) :: el, \theta) \quad \text{(vi)}$$
where $cands(P, g) = (t_1, P'_1, g_{1,1} \ldots g_{1,n_i}) :: \ldots :: (t_m, P'_m, g_{m,1} :: \ldots :: g_{m,n_m})$
and $l_i = \mathcal{R}([D_{g_{i,1}} \ldots; D_{g_{i,n_i}}]) \circ [S^{t_i}_g]$ for $i \in \{1 \ldots m\}$

$$((P, [S^t_g], fl) :: el, \theta) \xrightarrow{step} \text{(Success P)} \quad \text{(vii)}$$

$$([], \theta) \xrightarrow{step} \text{Failure} \quad \text{(viii)}$$

Table 1
Automatic procedure operational description

represented by $([], \theta)$: the elements list can be considered to list all the alternatives that can be used prove the initial goal, being empty means that all alternatives have been explored with a negative result. The annotation t in S_g^t is not used in the operational semantic, and t represents the lemma that was applied to g. Remember we have to show the user the history of lemmas applied so far. The procedure starts with the following configuration, where g is the initial goal and P the initial proof status and θ an empty cache.

$$([(P, [D_g], [])], \theta)$$

On such a status the step function applies rule (vi). calling the cands function to get a list of alternative proof statuses. All new goals are decorated with a D constructor, and sorted using the $\mathcal{R}$ function. They are positioned in front of the tl list, separated with an S item for the processed goal g. This item, when processed, will cache the proof found for g, and this will happen only after all newly created D items are solved.

In our example, assuming the result of the cads function amounts to $cands(P, g) = [(t_1, P_1, [g_1]);\ (t_2, P_2, [g_2; g_3])]$ we obtain the following state.

$$([(P, [D_g], [])], \theta) \xrightarrow{step} ([(P_1, [D_{g_1}; S_g^{t_1}], []);\ (P_2, [D_{g_2}; D_{g_3}; S_g^{t_2}], [g])], \theta)$$

Note that a new element is generated for every alternative proof status returned by the cands function. All of them, except the last one, are equipped with an empty failure (fl) list. In that way, if they fail, the cache will not be updated with a failure for g, since there are still valid alternatives for that goal. On the contrary, the last element inherits the failure list and adds to it g.

Rules (i) and (ii) process a success (that is an S item). The first rule is applied when no metavariable occurs in the goal, thus the proof found will not have side effects on the rest of the computation and can be safely added to the cache θ. In that case, the purge function is used to drop alternatives (brothers of g). They can be identified in the flat el list comparing the list of items, since the tl is inherited by all brothers (in rule (vi)) and is never modified.

Rule (iii) solves a D_g item when the cache θ holds a proof for the goal g. The substitution is enriched with an entry for g.

Rules (iv) and (v) are for partial failures, the former is applied when no applicable clauses are found, the latter when a failure was previously cached for the same goal.

Rule (vii) is for success, that is when no more items have to be processed. The final proof status is returned.

4.1 Improvements

The procedure presented in Table 1 can be improved in many ways, for example giving a bound to the search space or refining the caching mechanism. These improvements have been omitted from Table 1 to increase its readability, but are explained in the following.

To limit the search tree explored by the procedure to a certain depth, or even a number of nodes, some additional fields have to be added to the element structure. To efficiently keep track of the depth or size of the tree, the element structure is enriched with two integers representing the depth left and the actual size of tree: every time a D item is processed, the depth limit (as well as the size) is decreased. When an S item is processed the depth is increased again. The additional following rule is then added to the operational description:

$$((P, items, fl, depth, size) :: el, \theta) \xrightarrow{step} (el, \theta) \quad \text{(iii bis)}$$
$$\text{when } depth < 0 \vee size < 0$$

The cache θ is still not optimal, since a goal g of type T can be associated with $\bot$ because the algorithm run out of depth (or size). If the algorithm encounters again the same goal type T with a greater depth, it could retry. To fix this problem, goals have to be paired with the depth at which they have been generated in the failure (fl) list, and the $\bot$ symbol annotated with that depth. Then rule (iv) can be refined as follows:

$$(((\mathcal{P}, \Sigma) \text{ as } P, D_g :: tl, fl, depth, size) :: el, \theta) \xrightarrow{step} (el, \theta'_{m+1}) \quad \text{(iv)}$$
$$\text{when } \theta(T) = \bot_k \text{ and } k \geq depth \text{ and } \Gamma \vdash ?g : T \in \mathcal{P}$$
$$\text{where } \theta'_1 = \theta \text{ and } fl = \{(g_1, d_1); \ldots; (g_m, d_m)\}$$
$$\text{and } \Gamma_g \vdash ?g : T_g \in \mathcal{P} \text{ for } g \in \{1, \ldots, m\}$$
$$\text{and } \theta'_{g+1} = \theta'_g[T_g \mapsto \bot_{d_g}] \text{ for } g \in \{1, \ldots, m\}$$

Note that the last line stores failures for goals in the fl list that have to be enriched with the depth at which they have been processed in rule (vi).

The cands function can be modified to properly sort the list of returned proof statuses, in such a way that the most promising ones are processed first. The simplest heuristic is to count the number of newly generated goals (the length of l_i in rule (iv)).

4.2 Interfacing with the GUI

The GUI and the automatic procedure run in different threads. Rule (vi) checks a condition variable [4], associated with the *pause* button of the GUI, before proceeding. The computation status (the el list) is purely functional and every loop sets a global reference to that variable, allowing the GUI thread to render it.

The element list contains all the information needed by the GUI, but not in an handy format. The automatic procedure and the data structure it manipulates have been designed with both speed and user friendliness in mind, but execution speed has been always preferred to rendering speed or to making the rendering process

[4] A condition variable is a widespread synchronisation mechanism allowing one execution context to wait for a boolean variable to became true, and another execution context to change the value of that variable eventually waking up every thread waiting on that variable.

easier. The function to map the element list into a data structure suitable for the GUI is not interesting, even if far from being trivial, and will not be detailed here. It essentially amounts in processing in parallel all *op* lists (one for every element in the el list), grouping together the lemmas stored in S items. The lemma recorded in S items is shown to the user as the choice made the procedure. The actual statements of goals can be computed using the proof status $P = (\mathcal{P}, \Sigma)$, since all goals have an entry in the proof problem $\mathcal{P}$, and eventual instantiations of metavariables occurring in their types is recorded in the substitution Σ.

5 Related works

Many debugger or trace visualisation tools have been proposed by the logic/constraint programming community. Most of them like the ones described in [21,8] fall in the so called post-mortem trace analyser, allowing the user to inspect the computation once it has terminated.

The recent CLPgui [10] employs 2D and 3D visualisation paradigms to show the user the full search tree, allowing him to navigate it and zoom the interesting parts of the computation trace.

OzExplorer [20] adopts subtree folding to make the whole tree fit the screen, a requisite we do not have and thus we adopt a simpler viewport (a restricted view of the search tree). Moreover we hide solved subgoals (when their solution is not a choice, i.e. they do not instantiate any metavariable present in any other goal). [7] introduces the notion of choice stack (list of choices made so far), similar to our list of grey buttons in the rightmost column.

While our work shares some ideas and follows some visualisation paradigms described in these papers, the use case of our procedure in an ITP is clearly different from the general use case of a CLP program. These differences are summarised in the following:

- our GUI is rarely used to display a huge program (computation), thus it is tailored to the most frequent case of a tree of depth less then ten
- in ITPs like Matita, thanks to the reasonably large library that equips them, the branching factor is very high and that prevents a proper tree display: siblings would be too far to be visually related, thus we dropped the idea of visualising a tree
- every goal has a meaning per se, thus many informations like goals already solved can be hidden. The choice stack tells the user where the goal comes from and this information is enough to follow the computation

For these reasons we had to develop a novel user interface, instead of reusing or adapting one of the aforementioned tools.

6 Conclusions

In this paper we presented a SLD resolution based automatic procedure for the interactive theorem prover Matita, that is designed to be driven by the user through a graphical user interface. In this way we allow unexperienced user to observe the

procedure running, possibly understanding why it fails or how it managed to solve a goal for them. Trained users can easily tune the procedure pruning not promising branches of the computation or following good ones.

A still work in progress addition to this work is making the procedure generate not only a proof object, but also a proof script (the list of primitive commands to generate the proofs object) in the spirit of [4]. The choices made by the user interactively have to be recorded so that running again the automatic procedure possibly honours the same user requests. Having a proof script does not only show the user what the procedure did, but also greatly decreases the amount of time needed to re-check the proof script (since proof search has not to be performed again). Formalising mathematics with an ITP is not an easy task, and refining definitions is a really frequent activity that usually breaks many already proved lemmas. Having just a call to an automatic procedure can slow down the process of mending broken proof scripts, especially if there is no way to inspect what the procedure does, making it harder to understand the reasons of a failure. Our work already ameliorates this situation, but having a proof script that details the previously found proof, would be even better, allowing a fast re-execution and detection of the problem, and allowing the user to fix the proof directly if possible, or re-run the automatic procedure driving it towards a working proof.

References

[1] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Post-proceedings of the Types 2004 International Conference*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.

[2] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *Proceedings of Types 2006: Conference of the Types Project. Nottingham, UK – April 18-21*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2007. Special Issue on User Interfaces for Theorem Proving. To appear.

[4] Andrea Asperti and Enrico Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In *Calculemus/MKM*, pages 146–160, 2007.

[5] Gilles Barthe, Mark Ruys, and Henk Barendregt. A two-level approach towards lean proof-checking. In *Types for Proofs and Programs (Types 1995)*, volume 1158 of *LNCS*, pages 16–35. Springer-Verlag, 1995.

[6] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 515–529. Springer-Verlag, 1997.

[7] Christiane Bracchi, Christophe Gefflot, and Frederic Paulin. Combining propagation information and search tree visualization using ilog opl studio. In *WLPE*, 2001.

[8] M. Carro and M. V. Hermenegildo. Tools for search tree visualization: the apt tool. In P. Deransart, M. V. Hermenegildo, and J. Malusynsky, editors, *Analysis and Visualization tools for constraint programming, constraint debugging, LNCS*, volume 1870, 2000.

[9] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[10] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: a generic graphical user interface for constraint logic programming. *Journal of Constraints, Special Issue on User-Interaction in Constraint Satisfaction*, 9(4):241–262, October 2004.

[11] C. Quigley J. Meng and L. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[12] J.Meng and L.Paulson. Experiments on supporting interactive proof using resolution. In *David Basin and Michael Rusinowitch (editors), IJCAR*, 2004.

[13] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[14] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[15] César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[16] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supŕieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[17] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.

[18] Claudio Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5.

[19] Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *TYPES*, 2007. To appear.

[20] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 1997. The MIT Press: Cambridge, MA, USA.

[21] H. Simonis and A. Aggoun. Search-tree visualization. In P. Deransart, M. V. Hermenegildo, and J. Malusynsky, editors, *Analysis and Visualization tools for constraint programming, constraint debugging, LNCS*, volume 1870, 2000.

User Interfaces for Mathematical Systems that Allow Ambiguous Formulae

Claudio Sacerdoti Coen [1,2]

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY

Abstract

Mathematical systems that understand the usual ambiguous mathematical notation need well thought user interfaces 1) to provide feedback on the way formulae are automatically interpreted, when a single best interpretation exists; 2) to dialogue with the user when human intervention is required because multiple best interpretations exist; 3) to present sets of errors to the user when no correct interpretation exists. In this paper we discuss how we handle ambiguity in the user interfaces of the Matita interactive theorem prover and the Whelp search engine.

Key words: Overloading, ambiguity, user interface, theorem prover, Matita

1 Introduction

The traditional bi-dimensional mathematical notation is well known to be highly ambiguous. According to the usual phases of a compiler, we can see ambiguity everywhere.

Lexical analysis is ambiguous since, for instance, x_2 can be recognized either as two tokens (when the user is taking the second element of a sequence) or as a single token (when the user is referring to a bound variable x_2).

The grammar is inherently ambiguous, and precedence and associativity rules do not help since to the same symbol should be given multiple precedences according to its semantics: for instance, equality on propositions (denoted by $=$, a notational abuse for co-implication) has precedence higher than conjunction (denoted by $\wedge$), which is higher than equality on set elements (also denoted by $=$), which is higher than meet for lattice elements (also denoted

[1] sacerdot@cs.unibo.it

[2] Partially supported by the strategic project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

by $\wedge$). Thus $A = B \wedge P$ can be parsed either as $(A = B) \wedge P$ (a conjunction of propositions) or as $A = (B \wedge P)$ (equality of lattice elements).

Semantic analysis is the phase most affected by ambiguity. First of all mathematical structures usually belong to deep inheritance hierarchies, such as the algebraic (magmas, semigroups, groups, rings, fields, ...) and numerical ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$, ...) ones. Depending on the logic and semantical framework used to represent formulae, a formula that requires subsumption to be understood must be represented either as it is, or by insertion of explicit coercions [7]. Since multiple derivations can usually give meaning to a formula, semantic analysis becomes a one to many relation, at least when it inserts coercions. Secondly, even ignoring inheritance and subsumption, mathematical symbols are often highly overloaded in different mathematical contexts. As a trivial example, $-^{-1}$ is overloaded on semigroup elements (and thus on numbers by instantiation) and on relations (and thus on functions by inheritance). Moreover, x^{-1} can be understood either as x at the power of -1, or as the inverse of x (which is a semantically equivalent, but intensionally different operation).

Another problem in giving semantics to formulae is that the α-conversion equivalence relation, which semantically identifies formulae up to bound variable names, does not hold syntactically since it is common practice to reserve names for variables ranging over some set, omitting to specify for quantifiers the sort of the bound symbols. For instance, $f, g, \ldots$ usually range over functions, $x, y, z, \ldots$ over domain elements and R over relations, suggesting the expected interpretation for f^{-1} and x^{-1} in a context where f and x are implicitly universally quantified. More generally, mathematical texts often start setting up a local context of conventions used for disambiguation, and it is this context that drives disambiguation.

The last phase of a compiler is the translation of the semantically enriched abstract syntax tree in the target language. Loosely speaking, in the case of mathematical systems (and theorem provers in particular) which are based on a logic or a type system, this phase corresponds to the final internalization (representation) of the formula in the logic. For instance, an equality over rational numbers can be represented using Leibniz equality (the smallest reflexive relation), using a decidable equality over unique representations of rational numbers (e.g. as lists of exponents for the unique factorization), using an equivalence relation over equivalence classes of non unique representations (such as pairs of integer numbers representing the enumerator and the denominator), and possibly in many other ways. And all this without considering different representations of functions (e.g. as intensional, possibly executable algorithms, or as functional relations).

Since the standard mathematical notation is so ambiguous, it is worth asking if we want to address it in mathematical systems and if it is worth of, or if we better do like in the programming language community, where only artificial unambiguous languages are used. It can be argued, and we

considerably agree, that, even from the user point of view, it is worth avoiding ambiguity unless we are obliged to consider it. For instance, the user of a proof assistant who is working on a particular topic already needs to study not only the logic and the commands the system provides, but also to get acquainted with the library of already proved results, in order to avoid wasting time relying on lemmas that are not available. In these cases, learning also the ad-hoc notation adopted by the previous contributors to the library is probably a slightly annoying, but not very time consuming thing to do. Moreover, the benefits could be high since less ambiguity means better understanding of errors (since the user and the system assign for sure the same meaning for what has been written) and since no (or less) ambiguity can speed up the system in significant ways (since the set of interpretations for ambiguous terms is often exponential in the size of the formula).

Nevertheless, there are important situations where we cannot avoid ambiguity. The first one is user interaction with a mathematical library whose content is unknown. Suppose the user needs to look for some theorem or formula on the Web (maybe in a Wiki of mathematical results) or in the library of a proof assistant without knowing what is in the library and which notation has been used. If he is lucky, what he is looking for will have a name (for instance, if it is a famous theorem) or a set of keywords to identify it. More often, though, he could be interested in some technical result which he can only looked for by writing down a formula and looking for instances, generalizations or logical equivalences of that formula in the library. This is what happens all the time if we start using a proof assistant with a large, unstructured library: we know which technical lemma we need to prove some result, but we have no idea where the theorem could have been stored in the library, nor what notation has been used. Indeed, it has already been observed several times in the past that users tend to store ad-hoc, seemingly elsewhere useless technical lemmas relating to basic notions of the library in their own file on some advanced topic, to avoid the burden of polluting the library, or for the possibility of extending someone else files (especially if belonging to some "standard " library of the system). In this scenario, the only lingua franca to find what we need is the standard mathematical notation.

Another scenario where diverging from the standard notation may be difficult is in applications of theorem provers to didactics. According to our experience, it is possible to convince mathematical teachers to make students experiment with a proof assistant only if it understands exactly the set of formulae presented informally.

In a series of previous papers [9,3,10,11] we studied efficient algorithms that exploit type inference algorithms to speed up semantic analysis of ambiguous formulae. Our algorithms partition all possible interpretations of a formula in equivalence classes such as every equivalence class contains either one single well-typed interpretation, or a set of interpretations all characterized by the same typing error. The latter equivalence class is represented

User Provided Formula:

$\forall a, b, c, d.$

$(a +_1 b) *_1 (c +_2 d) =$

$a *_2 c +_3 a *_3 d +_4 b *_4 c +_5 b *_5 d$

(Partial) Interpretation:

$= \mapsto =_{\mathcal{L}} \quad *_1 \mapsto *_{\mathbb{N}} \quad +_1 \mapsto *_{\mathbb{Z}}$

Corresponding term with placeholders:

$\forall a, b, c, d.(a +_Z b) *_N ?_1 =_{\mathcal{L}} ?_2$

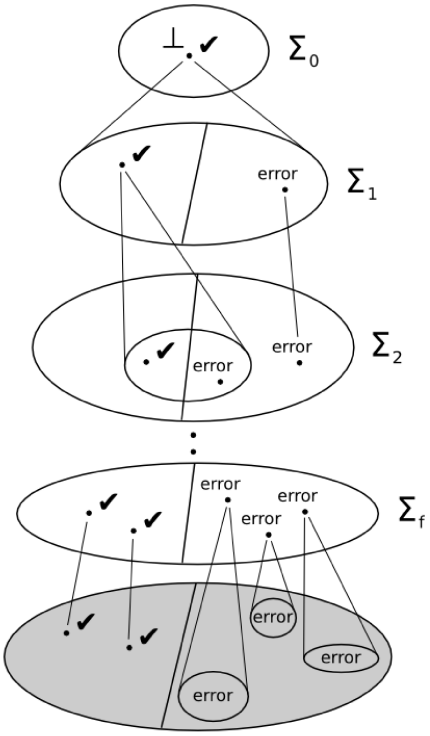
Refinement result:

`Error:` $a +_{\mathbb{Z}} b$ `has type` $\mathbb{Z}$ `but is here used with type` $\mathbb{N}$

Represented terms:

all instances of the formula that respect the partial interpretation constraints. They refinement error applies to all of them.

(a)



(b)

Fig. 1. (a) Partial interpretations (maps from overloaded symbol occurrences to their possible semantics) are useful to represent concisely set of terms that are all characterized by the same typing error. Note that each symbol occurrence can be given a different meaning. (b) General schema for efficient disambiguation algorithms [11]. Each Σ_i is a set of partial interpretations. If the term with placeholders that corresponds to an interpretation is ill-typed, the interpretation is not refined any more, but it is propagated as it is. If it is not ill-typed (✓ mark), the interpretation is refined in Σ_{i+1} by defining it in any possible way on one more overloaded symbol occurrence. Σ_f is the final set where every interpretation marked with ✓ is now total and uniquely identifies a single term (without placeholders). All other interpretations, marked with `error`, identify ill-typed disjoint sets of terms (without placeholders) which are all instances of the ill-typed partial term corresponding to the interpretation. The set in gray is the set of all terms without placeholders which are ill-typed semantics of the ambiguous formula. This set is never explicitly represented by the system.

by a formula containing placeholders such that every possible instantiation of the placeholders is an ill-typed formula. In turn, this formula containing placeholders is represented by a partial interpretation which maps some of the overloaded symbols to one of their possible meanings, and all remaining symbols to placeholders (see Fig. 1 (a)).

The main idea, explained in [9] and, formally, in [11] is that the cardi-

nality of the set of all possible interpretations is exponential in the number of overloaded symbols, whereas the cardinality of the partition computed by the algorithm is much smaller. Moreover, we can compute that partition by consecutive refinements of coarser partitions of equivalence classes of interpretations which are represented by terms with placeholders which are either ill-typed (not needing further refinement) or which are not known to be ill-typed (needing further refinement only if they contain at least one placeholder), see Fig. 1 (b).

Ambiguities introduced by the lexical analysis and parsing phases are not addressed by our algorithms for semantic analysis, but they can be previously addressed with parsers recognizing GLR grammars (see, for instance, [6]) in order to produce, from the input stream, compact representations of a set of abstract syntax trees to be feed to our semantic analysis. The final output can still be represented by a single partition that satisfies the properties required above.

Computing the partitions efficiently is, however, only part of the problem. The most critical aspects of ambiguity is indeed interaction with the user, that we can analyze according to the shape of the partition returned by the parsing and semantic analysis phases.

The simplest scenario is a partition containing only one equivalence class representing a single well-typed term. In other words, there exists only one interpretation that "makes sense" (is well-typed), and possibly many others which do not. It is thus natural that the system picks the correct one without any interaction with the user, since it is unlikely (but not impossible) that the interpretation the user has in mind is a different one. Sect. 2 addresses the problem of providing non-invasive feedback to the user about the chosen interpretation.

The second scenario is the one where there exists in the partition multiple equivalence classes representing well-typed terms. When subsumption is explicitly represented by coercions in the semantics, this happens all the time since a formula like $x + y$ can live at any level of the algebraic numeric hierarchies. Even when this is not the case, if we do not consider the contexts the formula lives in, many formulae receive multiple well-typed semantics.

In [3] we addressed the problem by introducing aliases. An alias is a preference for some interpretation of a symbol that can be given explicitly by the user, or that is automatically inferred by the system looking at previous recent uses of the symbol. Equivalence classes representing correct terms in the partition are ranked according to their degree of respect for aliases. Special aliases can be used to control insertion of coercions.

Aliases are not sufficient to impose a linear order on the interpretations. For instance, consider a formula where there occurs two overloaded symbols $+$ and $*$ and a partition with only two correct interpretations: the first one respects only the alias for $+$, and the second one only that for $*$. Thus no interpretation is ranked absolutely higher than the previous one.

In [3], to force a linear order on ranks, we have introduced the notion of passes: each pass is characterized by a class of aliases used to constraint the accepted interpretations. For instance, in Matita we are currently using five passes.

(i) do not insert coercions and use only the most recent alias for each symbol, i.e. use only the last recently used interpretations for a symbol

(ii) as the first one, but insert coercions

(iii) do not insert coercions and use only already used aliases, without inserting new ones, i.e. do not automatically pick a new interpretation for a symbol which has not been used yet; Fourth pass: as the third one, but insert coercions

(iv) look for all interpretations of a symbol, adding new aliases for the chosen interpretation. Equivalence classes in the partition are ranked according to the pass that produces them.

Aliases and passes do not fully solve the problem, since multiple correct interpretations generated in the same pass are on purpose ranked in the same way. Thus the user interface must collect from the user enough information to pick the right interpretation. This is the topic of Sect. 3. Moreover, it may happen that the automatic ranking fails to rank first the interpretation the user has in mind. Thus, as in the first scenario, it is important that the user interface provides non-invasive feedback on the interpretation given to formulae. This is the topic of Sect. 2.

The third scenario is the one where the partition only contains equivalence classes of interpretations that are not well-typed. This means that all possible interpretations contain an error. Our disambiguation algorithm that collects errors in equivalence classes already allows to reduce (usually exponentially) the number of alternative error messages to be presented to the user. Nevertheless, this is not sufficient since the user has a single interpretation in mind and, when presented with multiple errors associated to multiple interpretations, he must first spend time to spot the right interpretation before trying to understand the error. When too many interpretations are listed, this procedure is so annoying that the user stops reading the errors and randomly tries to fix the error ignoring the system provided potentially useful information.

In [10,11] we addressed the problem by ranking equivalence classes of ill-typed terms pruning out spurious errors. A spurious error is an error located in a sub-formula which admits alternative interpretations that assign to the same sub-formula a well-typed interpretation. The idea of a spurious error is that a spurious error is likely to be due to a wrong choice of interpretation, and not to a genuine user error.

Spurious error detection can be efficiently integrated in our efficient disambiguation algorithms and, according to our benchmarks in [11], is effective in reducing the average number of errors to be presented to the user. Nevertheless, we need a light-weight user interface to present the remaining non

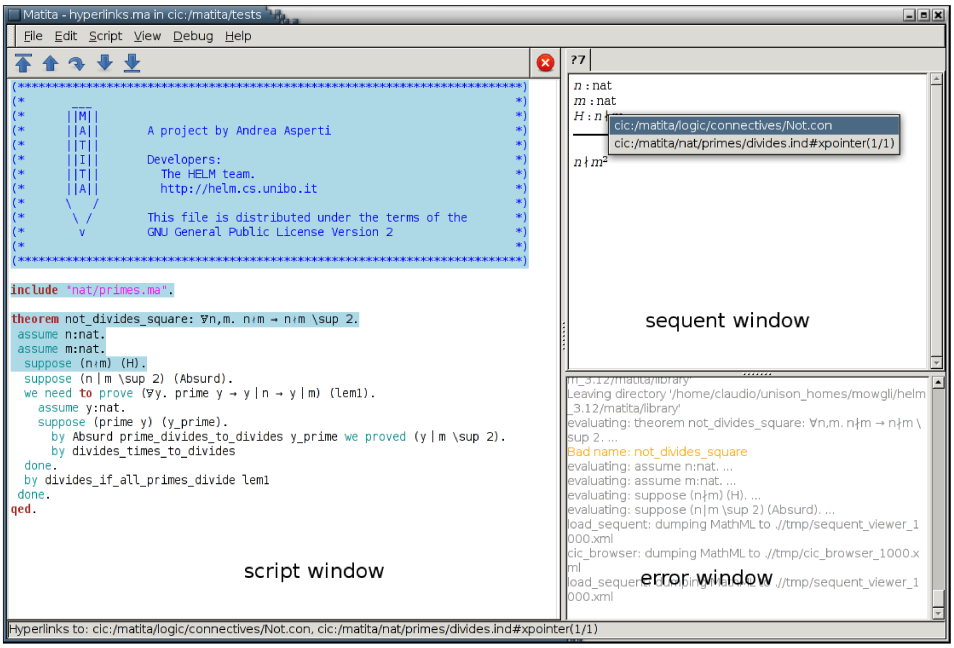


Fig. 2. The user has clicked on the "not divides" symbol in the hypothesis, which hides the formula `Not` (`divides` n m). The status bar lists the hyperlynks for `Not` and `divides` as soon as the mouse is over the symbol. The contextual menu is shown only after left button press on the symbol.

spurious errors to the user, possibly ranked according to passes, and to present on demand also the spurious errors in the rare case of false positives [11]. This is the topic of Sect. 4.

The conclusions of the paper are in Sect. 5.

2 Disambiguation feedback

Since the mathematical notation is overloaded, and since interpretations are automatically chosen by the system among the correct ones, it is important to provide feedback to the user on the way formulae are interpreted. We believe that hyperlinking every symbol, constant and notation to the definition of its semantics already provides on demand enough feedback to the user. In the Matita theorem prover [3] and in the HELM/MoWGLI Web interfaces [1], this is achieved by means of hyperlinks that are followed when the user clicks on a symbol or constant. Moreover, a status line shows the URI of the hyperlink when the mouse is put over the symbol.

Some mathematical notations actually hide more than one symbol, which are independently given a semantics. For these reasons, in Matita it is possible to have hyperlinks to multiple targets, each one identified by its URI. When the user clicks on the hyperlink (see Fig. 2), a contextual menu shows one distinct hyperlink for each URI.

User interfaces as the one of Matita, which are based on the ProofGen-

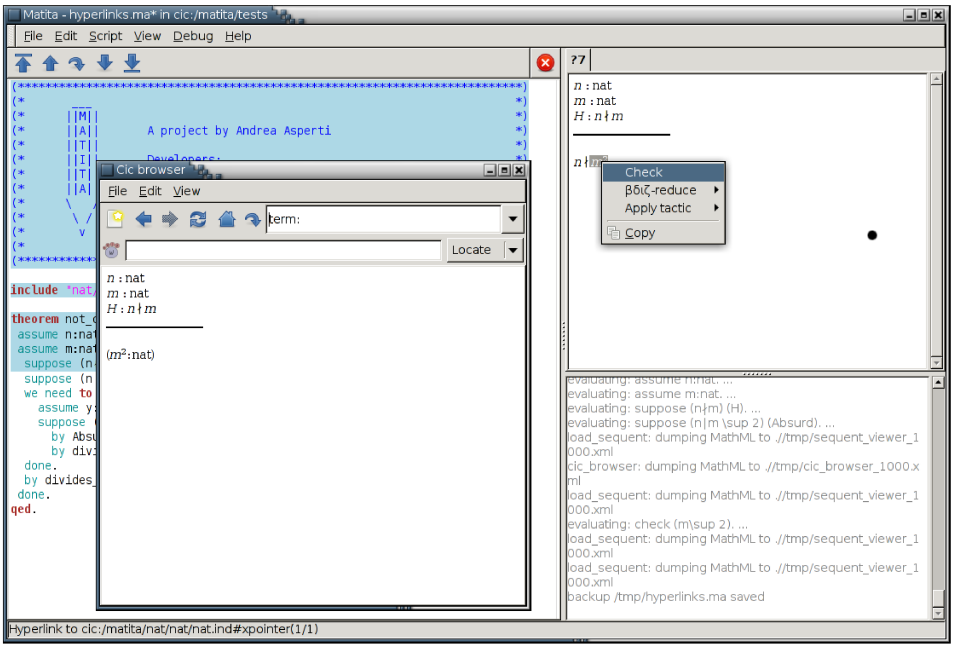


Fig. 3. After semantic selection of the well-formed sub-formula m^2, the user asks to compute its type. The selected term is shown in the CIC browser together with its type and the context it lives in.

eral [4] paradigm (whose origins go back to the CtCoq system [5]), are characterized by an input buffer (script window) and two output buffers (sequent window and error window), indicated in Fig. 2. Matita also has CIC brower windows (in the foreground in Fig. 3), which are non modal floating windows used to browse and search in the library of the proof assistant. In Matita, the sequent window and the CIC browser are actually implemented by a widget for MathML Presentation [8], and formulae are translated from their semantics representation in the Calculus of (Co)Inductive Constructions to MathML Content and then to MathML Presentation according to the transformations described in [1], which are responsible for generating the hyperlinks.

Thanks to the cited technology, hyperlinks are actually provided in Matita for the sequent and CIC browser windows. On the other hand, the script window is implemented by a textual widget that shows user provided text. Traditionally, the upper part of the text, which has already been processed by the system, is read-only and highlighted changing the background color (see again Fig. 2). Formulae contained in the text have already been disambiguated for execution. Moreover, in order to localize error messages, the position of all tokens has been recorded during parsing. If the textual widget supports hyperlinks and contextual menus, it should be easy to add hyperlinks also to the locked part of the text. This is currently planned for Matita, but not implemented yet.

Hyperlinks do not help with subsumption. When subsumption is required to type a formula, the system may add coercions in order to record where

subsumption has been used, and we would like to inform the user about that in a non-invasive way. However, since coercions are not represented visually in any way (to avoid too much noise), there is in general no place where to put an hyperlink. Thus, our current solution is to provide additional feedback allowing the user to semantically select sub-terms in the sequent window and ask to compute their type as in Fig. 3.

Semantic selection [3] is a restriction of selection to well-formed sub-formulae which we provide on top of the MathML widget, which displays documents already represented by XML trees. We do not plan to provide the same functionality on the script window, since semantic selection is not easily supported by textual widgets and since re-computing the type of the sub-formula requires re-disambiguation of the formula under the same conditions the formula was disambiguated in the first time. These are no longer the conditions the system is in.

We believe that asking for the type of a sub-formula is an useful feature anyway, but that is not fully satisfactory to detect the use of subsumption. The reason is that multiple checks can be required to fully understand where coercions have been put. Another strategy we have adopted is to add an option to the `View` menu of Matita to temporarily stop hiding of coercions. This is also not satisfactory since, when hiding is deactivated, the feedback is too invasive. We are currently looking for better solutions.

3 Choosing an interpretation

As already discussed in the introduction, after disambiguation of a formula there could be multiple equally ranked interpretations, that differ on the interpretation of at least one overloaded notation. Fig. 4 shows a very simple example where the user in a new file starts using the infix addition and multiplication notation which are overloaded in the library over integer and natural numbers. The system computes the partition of ranked equivalence classes of interpretations, finding two interpretations with maximal rank. In the first one, all occurrences of the symbols are interpreted over natural numbers, in the second one over integer numbers. Other correct interpretations that receive a lower rank are obtained by considering subsumption between natural and integer numbers. For instance, another possible interpretation is given by $\forall a, b : \texttt{nat}.\forall c, d : \texttt{int}.(a +_{\mathbb{N}} b) *_{\mathbb{Z}} (c +_{\mathbb{Z}} d) = a *_{\mathbb{Z}} c + a *_{\mathbb{Z}} d + b *_{\mathbb{Z}} c + b *_{\mathbb{Z}} d$.

Since the system is unable to decide which maximally ranked interpretation is the one expected by the user, it computes a tree of discriminating questions among interpretations. Each node in the tree is a multiple answer question about the meaning of a symbol, where the possible answers range among the meanings used in the set of correct interpretations. The node has a child for each possible answer. The root of the tree is the question that allows to prune the higher number of interpretations. Its children are computed recursively according to the same criterion applied to the remaining set of

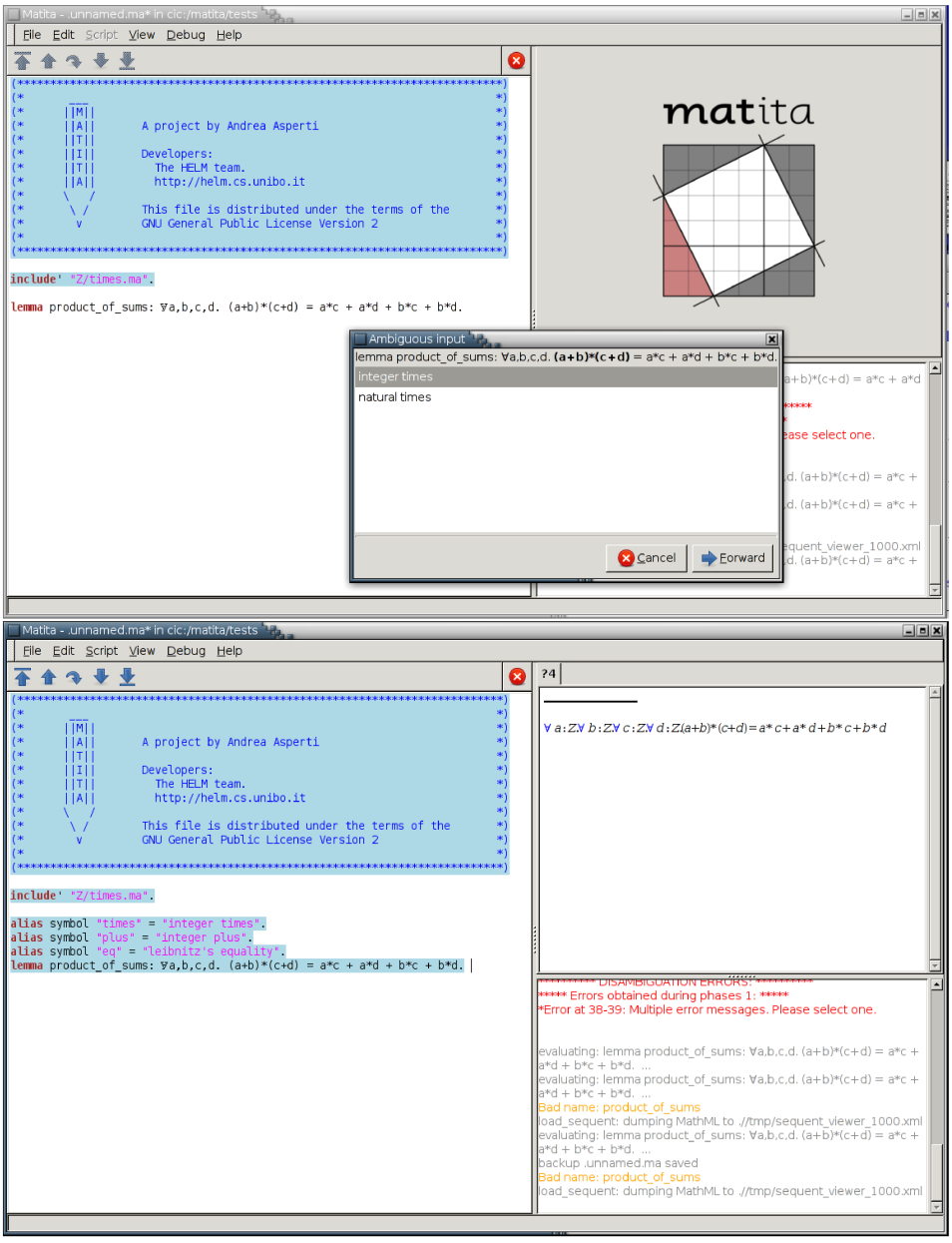


Fig. 4. Ambiguous input. In the upper figure, the user is asked for the interpretation of $*$ in the sub-formula $(a + b) * (c + d)$ (which is highlighted in bold). Since the interpretation for $+$ is constrained by the choice, no further questions need to be asked, and the **Forward** button adds the aliases to the script (in the lower figure). The **include'** command activates the infix $+$ and $*$ notation for integer and natural numbers without pre-loading any alias.

correct interpretations. In our example, we get a degenerate tree with only one node, since one question is sufficient to identify just one maximally ranked interpretation.

Since the system interacts with the user, it is important that the user provided information is recorded somewhere in the script in order to avoid

repeating the interaction the next time the script is processed. This is achieved by automatically adding aliases to the script (see second half of Fig. 4). These are exactly the same kind of aliases we already discussed in Sect. 1. Aliases can be either automatically generated or they can be declared by the user with the same syntax. In combination with passes, they are used to rank interpretations.

The string before the equal sign in an alias declaration is the name of the MathML Content symbol used to give a representation of the notation at the content level. The string after the equal sign was previously associated by the user to a MathML Content to MathML Presentation mapping when declaring the notation. We also associate aliases to identifiers which are represented in MathML Content by themselves. In this case, the syntax becomes

```
alias id "name" = "URI".
```

Aliases look similar to Mizar's environments where the user needs to list, at the beginning of an article, all notations (but also definition and theorems) he wants to use. But the syntactic similarity is (partially) misleading: in Matita all definitions, theorem and, potentially [3], notations are always visible and the user does not need to declare in advance which parts of the library it depends on. On the other hand, like in Mizar, the list of aliases in a script becomes very large when no alias is pre-loaded in advance. To this aim we provide the `include` command that pre-loads all aliases that were active at the end of a previous script. The `include` command looks similar to Coq's `import` or to Isabelle's theory importing machinery and it leads to the same advantages with respect to explicitly listed aliases (see [12], Sect. 4.8 for a short comparison). Even in this case, however, the similarity is only syntactical, since definitions, lemmas and potentially notations can be used anytime in Matita even without including them. The `include` command only pre-loads aliases to set preferences (that can be overriden) on the preferred interpretations for overloaded symbols and notations.

4 Error reporting

As already discussed in the introduction, disambiguation of a formula containing an error results in a partition made of ranked equivalence classes of interpretations characterized by the very same error (one for each equivalence class). This is the most difficult scenario for an user interface, since the user is already making a mistake (and thus he can be confused), and we risk to show errors relative to interpretations he does not mean (increasing the con-

[3] The current implementation of Matita is based on the CamlP5 parser which does not handles GLR grammars. Thus it is currently not possible to pre-load all user notations given in the library. The `include` command of Matita thus performs both pre-loading of user notation and pre-loading of aliases. The `include'` alternative form pre-loads notation alone. We are currently experimenting with alternative GLR grammars for OCaml in order to remove this limitation.

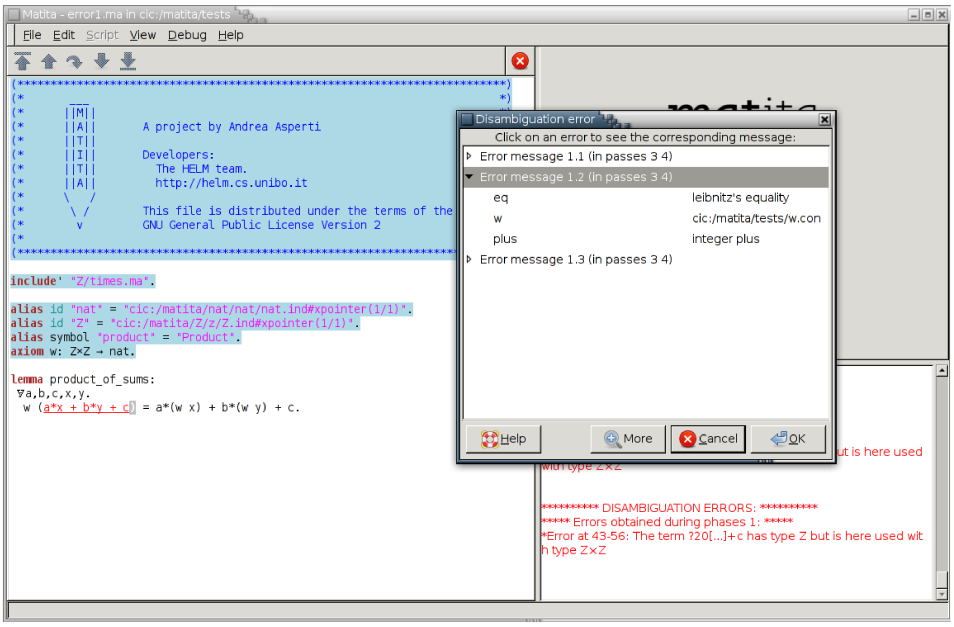


Fig. 5. Error parsing an ambiguous formula. Three partial interpretations are sufficient to represent all possible errors. Moreover, all the errors are located in the same sub-formula $a * x + b * y + c$ which is underlined and highlighted in red. The error message relative to the first interpretation is shown in the error window and a modal window allows to see the other error messages (which are always displayed in turn in the error window). By clicking on the small triangles, the user can also see on demand the aliases that make up the interpretation. The latter feature may be necessary to fully understand the error message. Errors relative to passes 1 and 2 are hidden since they also belong to passes 3 and 4. Errors belonging to pass 5 (which completely ignores aliases) as well as errors classified as spurious are not shown by default, but they are shown by pressing button `More`.

fusion) and to provide too many information (augmenting the confusion and the time to data-mine the information). The natural reaction of the user is to completely ignore the system feedback trying to understand the error by himself without machine help. This was indeed what users were reporting us before the current interface was implemented.

The main observation that allowed us to improve the situation came from Ferruccio Guidi, who told us that, even when programming in OCaml [4], he uses to ignore the actual error message in favour of the error location only. Thus we changed the user interface in order to present to the user a list of error locations and, only on demand, the error messages relative to that location

[4] The most frequently reported error by the OCaml compiler is "This expression has type T_1 but is here used with type T_2" where T_1 and T_2 are inferred by the compiler according to the usage of the bound variables. However, since inference is done globally on the source code, it frequently happens that one of the types is inferred incorrectly, and the error message becomes misleading. On the other hand, the error location is almost always correct. Thus users tend to read the error only if they are unable to immediately spot the problem from the error location.

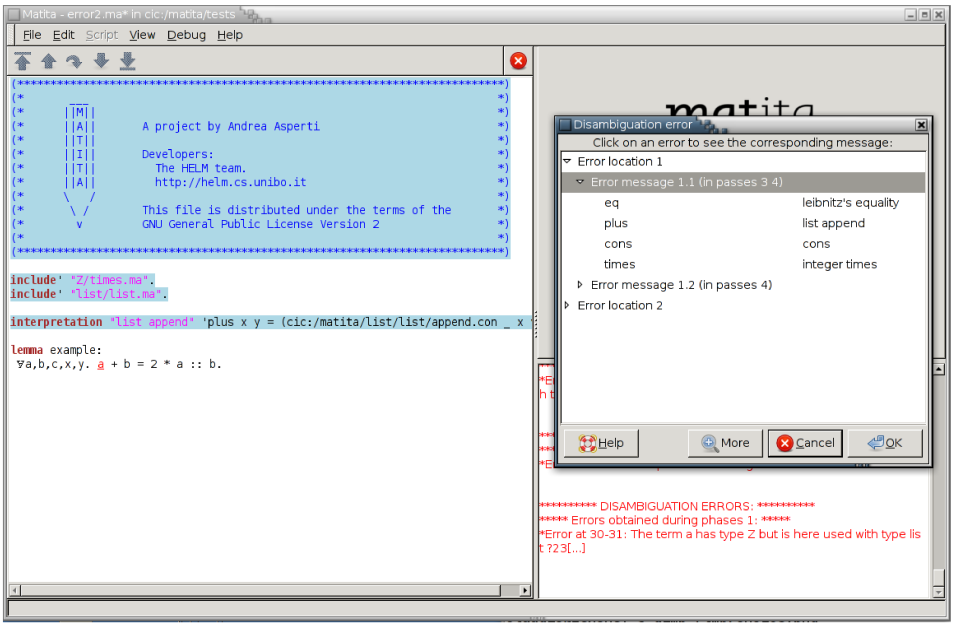


Fig. 6. Error parsing an ambiguous formula. The symbol + has been overloaded also as list append. The symbol :: adds an element (on its left hand side) to a previously existing list (on the right hand side). Since lists are homogeneous, the formula is ill-typed. Interpretations (and corresponding error messages) are categorized by error location first, then by error message. The error location is underlined and highlighted in red when the user clicks on an error location entry in the modal window. Error messages are printed one at a time in the error window when the user clicks on the error message in the modal window. As in Fig. 5, further information over interpretations is displayed only on demand, and the `More` button is used to see spurious errors and errors relative to interpretations that completely ignore aliases.

and, even more lazily, the description of the partial interpretation that is affected by that error.

Fig. 5 and 6 show our user interface both in the degenerate case (but quite frequent, see [11]) of one single error location, and in the general case. As explained in the captions, the user interface hides by default the error messages (and relative locations) for those interpretations that are unlikely to be the one the user has in mind, according to the ranking (which, in turn, depends on the phase used to constraint the interpretation and on the spurious error criterion).

We are currently fully satisfied by this user interface to show multiple error messages. On the other hand, ranking and spurious error detection can probably be always improved to reduce the number of errors to show, at the price of increasing the risk of false negatives.

5 Conclusions

As far as we know, Matita is the only theorem prover that supports arbitrarily overloaded notation and that implements a user interface to cope with ambiguities. We have identified three situations where the user interface plays an important role.

The first one is in providing feedback about the way the system has interpreted symbols. This has been achieved using conventional techniques like hyperlinks and type inference for sub-formulae.

The second situation is user interaction to select one interpretation among those that are deemed equally likely. Our strategy consists in minimizing the interaction by building trees of maximally discriminating questions. An additional challenge consists in the need to record the user choices to avoid repeating the interaction.

The third and most critical situation is that of presenting multiple error messages associated to a wrong formula. Here we designed an interface to progressively provide information to the user on-demand, starting from the less informative (but less confusing) one (i.e. error locations) and moving to the one requiring more effort to be understood by the user.

A preliminary version of the user interface was implemented for the Whelp search engine [2], and re-implemented in Matita, but it was not satisfactory. Matita now implements the new interface described in the paper and we plan to port the new user interface also to Whelp.

Since alternative user interfaces providing the same functionality do not exist, it is difficult to do comparisons. Similarly, we are looking for ideas to collect a quantitative feedback from our users. On the other hand, the ones that were exposed to the previous interface declared to be far more satisfied.

References

[1] Asperti, A., F. Guidi, L. Padovani, C. Sacerdoti Coen and I. Schena, Mathematical knowledge management in HELM, Annals of Mathematics and Artificial Intelligence **38(1-3)** (2003), pp. 27–46.

[2] Asperti, A., F. Guidi, C. Sacerdoti Coen, E. Tassi and S. Zacchiroli, A content based mathematical search engine: Whelp, in: Post-proceedings of the Types 2004 International Conference, Lecture Notes in Computer Science **3839** (2004), pp. 17–32.

[3] Asperti, A., C. Sacerdoti Coen, E. Tassi and S. Zacchiroli, User interaction with the Matita proof assistant, Journal of Automated Reasoning (2007), special Issue on User Interfaces for Theorem Proving. To appear.

[4] Aspinall, D., Proof General: A generic tool for proof development, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, Lecture Notes in Computer Science **1785** (2000).

[5] Bertot, Y., The CtCoq system: Design and architecture, Formal Aspects of Computing **11** (1999), pp. 225–243.

[6] Heering, J., P. Klint and J. Rekers, Incremental generation of parsers, IEEE Trans. Softw. Eng. **16** (1990), pp. 1344–1351.

[7] Luo, Z., Coercive subtyping., J. Logic and Computation **9** (1999), pp. 105–130.

[8] Padovani, L., "MathML Formatting," Ph.D. thesis, University of Bologna (2003), technical Report UBLCS 2003-03.

[9] Sacerdoti Coen, C. and S. Zacchiroli, Efficient ambiguous parsing of mathematical formulae, in: A. Asperti, G. Bancerek and A. Trybulec, editors, Proceedings of Mathematical Knowledge Management 2004, Lecture Notes in Computer Science **3119** (2004), pp. 347–362.

[10] Sacerdoti Coen, C. and S. Zacchiroli, Spurious disambiguation error detection, in: Proceedings of Mathematical Knowledge Management 2007, Lecture Notes in Artificial Intelligence **4573** (2007), pp. 381–392.

[11] Sacerdoti Coen, C. and S. Zacchiroli, Spurious disambiguation errors and how to get rid of them, Submitted to Journal of Mathematics in Computer Science, special issue on Management of Mathematical Knowledge. Available at the first author's home page. (2008).

[12] Wenzel, M. and F. Wiedijk, A comparison of mizar and isar, Journal of Automated Reasoning **29** (2002), pp. 389–411.

Managing Proof Documents for Asynchronous Processing

Holger Gast[1]

Wilhelm-Schickard-Institut für Informatik
University of Tübingen
Tübingen, Germany

Abstract

Asynchronous proof processing is a recent approach at improving the usability and performance of interactive theorem provers. It builds on a simple metaphor: The user edits a proof document while the prover checks its consistency in the background without explicit requests from the user. This paper presents a software architecture for asynchronous proof processing. Its foundation is a novel state model for commands that synchronizes the possibly parallel accesses of the user interface and prover. The state model is complemented by a communication protocol that places minimal requirements on the prover. The model also allows asynchronous processing to be emulated by existing linear-processing proof engines, such that the migration to the new communication protocol is simplified. A prototype implementation that works with the current development version of Isabelle is presented.

Keywords: asynchronous proof processing; usability of interactive provers; software architecture

August 1, 2008Draft: 1 August 2008

1 Introduction

The communication with an interactive prover has traditionally been structured linearly [6,1]: The commands of a proof script are stepped through one-by-one, and the region that has been sent becomes locked to prevent further editing by the user. An undo mechanism built into the prover is used to revert the steps and unlock parts of the region on demand. In this model, the user interface serves as a script buffer that tracks the commands that have been processed by the prover, such that they can be saved to a file for later replay.

The linear processing model is very much centered on the mechanics of proving and it is not flexible enough for greatly improving the usability of future user interfaces. One approach to usability is the direct manipulation of familiar objects [14]. Aspinall et al. [5] have developed a document-centered view in which the user edits a proof document just as a mathematician would edit a pen-and-paper proof. The

[1] Email: gast@informatik.uni-tuebingen.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

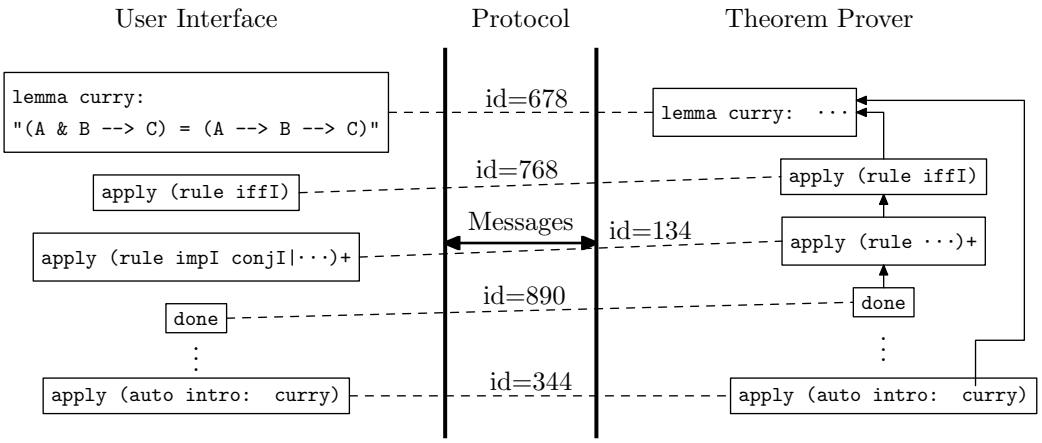


Fig. 1. Commands in the Interface and Prover

prover is used only to verify the consistency of the document. The actual processing of proof commands, however, remains linear in their proposal.

Wenzel [17] has recently pointed out that the linear processing model is far from optimal. The first possible improvement is the use of modern multi-core processors for parallel processing of independent proof commands. In the Isar [19] language, for example, proofs do not influence any of the references to the proven fact. It is therefore possible to postpone the execution of proofs until processing resources become unused, and different proofs can be executed by different processors in parallel. Since proofs take 95% of the overall processing time, the document structure itself can be re-checked almost immediately in response to edits by the user. The second improvement concerns usability. In the Mizar system [12], the prover runs in batch mode and annotates the input proof document with error messages where processing fails, but continues with the next command that does not depend on the erroneous command. The usability of the prover is greatly improved, because the user can work in terms of the metaphor of a proof document. Wenzel proposes to make this kind of response available for interactive proving sessions. The linear processing model is dropped in favor of asynchronous processing of proof documents, where the prover decides when it will process which command.

The purpose of this paper is to explore the demands that asynchronous proof processing poses on the user interface component and the software design of both interface and prover. Our main contribution is a new state model for commands that enables asynchronous processing and a corresponding protocol for the communication between interface and prover. Since the protocol allows the prover to choose the processing order, it can be also be supported by existing, linear-processing provers during a migration phase. We present a concrete implementation of a user interface that works with the current development version Isabelle.

Figure 1 summarizes the overall challenge: The proof document editor on the left holds the textual representation of the commands as they were typed by the user. The prover on the right holds an internal data structure that records the dependencies between commands and allows the commands to be scheduled for processing. The prover and the interface communicate by sending messages through some communication channel. The commands on both sides are linked logically

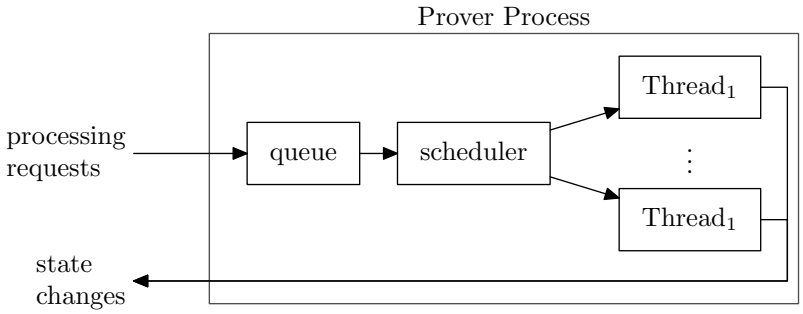


Fig. 2. Model of the Prover based on the Active Object Pattern

through unique IDs. Messages passed between prover and interface communicate changes to specific commands by referring to their IDs.

The remainder of the paper describes our solution to this challenge. Section 2 proposes a state model for commands that delegates the decision about the order of processing entirely to the prover. Section 3 describes a software architecture for the user interface that supports asynchronous proof processing. Section 4 compares our proposal to related work. Section 5 concludes.

2 A Document Model for Asynchronous Processing

Asynchronous processing of proof documents requires a self-contained state model for individual commands: Both the user interface and the prover manipulate the command, possibly at the same time, and the effects and interactions of these manipulations must be well-defined in every possible situation and every possible order. This section develops a state model for the user interface and a protocol for communication with the prover.

2.1 A Model of Asynchronous Processing

Isabelle is currently being extended to support asynchronous processing of commands [18]. To place as few constraints as possible on the software structure of Isabelle, we abstract over the concrete implementation and base our architecture on an abstract model of asynchronous processing. This approach has the additional advantage that the infrastructure and user interface that we develop in Section 3 will work with other provers as well.

The basis of our system model is the Active Object pattern [13], which has proven successful in systems that do asynchronous processing. The core of this pattern is shown in Figure 2. The prover receives processing requests and stores them in a queue until a worker thread becomes idle. At this point, a scheduler examines all requests in the queue, decides which of them is to be executed next and hands it on to the idle thread. When processing finishes, the thread sends the result back to the originator of the request.

It remains to define the messages that contain requests to the prover and notifications to the interface. This protocol can be designed in two ways: By focussing on the interface as the originator of the requests or by focussing on the prover as the component that handles them. We choose to start from the interface for two

reasons: First, asynchronous processing of proof documents can only provide an increase in usability if the metaphor of a "proof document" is presented to the user in a consistent manner, and this is not very likely if technical considerations dominate the protocol design. Still more importantly, designing a protocol also encompasses defining a state model that specifies under which conditions which messages may be sent or received. Since the prover should be free to choose a processing order that suits its existing software structure, it would be unacceptable make any prescriptions here.

In order to design the communication protocol from the interface point of view, we need to design a state model for commands used by the user interface. The messages of the protocol then correspond to the events that label transitions of the state machine. In the subsequent presentation, we use the terminology of the UML [7], including substates (or nested state machines). Events that do have no transition from a state are ignored.

2.2 The State Model for Commands

We view a proof document as a text document that is partitioned into non-overlapping *commands*. Each command is a section of the text that can be sent to the prover individually. The main concern is the problem of serializing accesses to shared resources which occurs in any form of asynchronous or concurrent processing. In the current application, the commands are conceptually shared between the prover and the interface and each component needs to manipulate them according to internal considerations. The conventional model of mutexes to prevent interference is not sufficient, since prover and interface run in separate processes. We therefore introduce an ownership semantics [11]: Instead of sharing some memory object between two threads, each process manipulates those commands that it owns, and there exists a protocol for transferring ownership.

Figure 3 shows the resulting state model for commands. The prover owns the command if and only if the command is in state *sent*; otherwise, the interface owns the command. The user may manipulate commands that are in state *idle*. In particular, only idle commands can be destroyed. The change of ownership occurs by *sending* the command to the prover and by *revoking* the command from the prover. Neglecting the nested state machine in state *sent* for the moment, the events capture just this process: The events *send* and *revoke* are generated by the interface whenever it judges that a command is to be processed by the prover or is to be revoked for further editing. The event *accepted* occurs as soon as the message with the command has been transmitted to the prover via the communication channel. The event *released* is generated by the prover when it has deleted all references to the command from its internal data structures.

The states *to be send* and *to be revoked* are necessary since neither sending nor revoking are synchronous operations. The interface must not stall until the prover answers a particular request, because due to dependencies among commands, both sending and revoking a command may take a noticeable time. The state *to be send* therefore indicates that the interface waits for the prover to accept the command; state *to be revoked* indicates that the interface waits for the prover to release it.

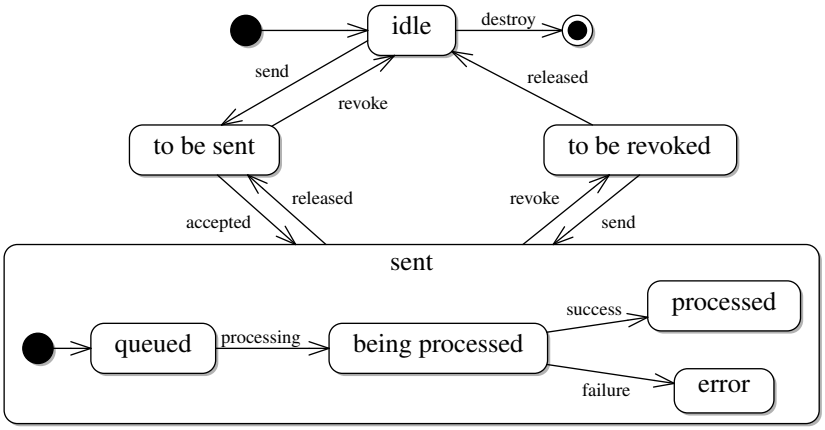


Fig. 3. A new state model

The state *sent* indicates that the command has been received successfully by the prover. The state has four substates which reflect the general execution model from Section 2.1. They are introduced for the benefit of the user who will want to be informed about the progress of proving. The user interface may, for instance, highlight the commands according to the substate. The transitions are labelled with informational messages sent by the prover. If a command ends in state *error*, then the interface may decide revoke the command automatically for further editing.

The transitions in the outer state machine should be clear from the meaning of the events. We point out the following details, because they clarify the intention of asynchronous document processing and delineate the approach from sequential, history-based models.

- Except in the purely informational nested machine, there are no events *success* or *failure*, because their meaning relates to the order of execution, which is considered an internal decision of the prover.
- There is no event *interrupt* which the interface could send to interrupt a particular command. Interruption occurs automatically if the prover receives a *revoke* message for a command that it happens to be processing. In the model of Section 2.1, the scheduler will abort the corresponding working thread.
- The prover may decide to release a command even without a *revoke* request. This may happen due to dependencies known only to the prover. However, from the user's point of view, the command still is to be processed. The transition from *sent* on event *released* is therefore to state *to be sent* rather than *idle*.

  One instance of this behaviour is Isabelle's undo mechanism. When the finishing proof step (`done`,`by`,`qed`) of a theory-level statement is undone, then the entire proof is undone. The above *released* transition ensures that those proof commands that the user has not explicitly requested to be undone will be re-executed automatically.
- Because of the ownership semantics, there is a direct transition from *to be sent* to *idle* on event *revoke*, the transition occurs without the prover being involved. Likewise, the transition from *to be revoked* to *to be sent* on event *send* can occur without the prover being notified.

2.3 Protocol for Prover–Interface Communication

The protocol contains three groups of messages exchanged between interface and prover. The first group consists of the events in the state model described in Section 2.2. They negotiate ownership for individual commands and convey information about their current processing state. Each message contains the ID of the command whose state is modified. It is important to note that no particular sequence of messages is prescribed. By the nature of asynchronous processing, the events that may occur are determined from the states of individual commands alone.

The second group addresses the maintenance of the document structure. Since a batch run must be guaranteed to produce the same results as the interactive work, the textual order of commands in the proof document needs to be known to the prover. The interface therefore sends message `create(`*id*`,`*prev*`)` whenever it creates a new command with ID *id* whose textual predecessor has ID *prev*. It sends `destroy(`*id*`)` when the user edits have destroyed the command with ID *id*. The interface must own the command that it reports as destroyed.

The third group consists of a single message `request(`*id*`)` that the prover sends to the interface if it judges that it cannot proceed with processing without owning command *id*. The interface is, of course, free to disregard this request. The motivation for this request is seen from the following simple situation:

```
lemma "A & B --> A"
apply auto
done
```

When the user decides to send the `done` command, the prover can easily determine that it needs the preceding commands up to the next top-level statement, for processing. If the prover could not request commands, the interface would have to send all preceding commands, because some of them just *may* be necessary.

2.4 Retrofitting Existing Provers

The switch from a synchronous, linear processing model to asynchronous processing and event-based communication requires a major change in the design of the prover. This section shows that it is straightforward to insert an emulator between interface and prover that communicates with the interface by the new asynchronous protocol, while executing commands synchronously in the background using the existing communication channel to a single-threaded prover. As a first step to implementing asynchronous processing, this emulator could also be implemented in the prover.

The emulator follows the model of Figure 1 directly. It maintains a doubly-linked list of commands with unique IDs and a mapping from IDs to command objects. The list is constructed according to the `create` and `destroy` messages received from the interface. The remaining messages from the interface concern the state of individual commands. Since the interface is free to choose any sequence of *send* and *revoke* messages, the emulator must also keep track of the individual commands' states. Figure 4 shows the state model used by the emulator. Its overall structure resembles the inner state machine of state *sent* in Figure 3, but special handling for interrupts and undo is required.

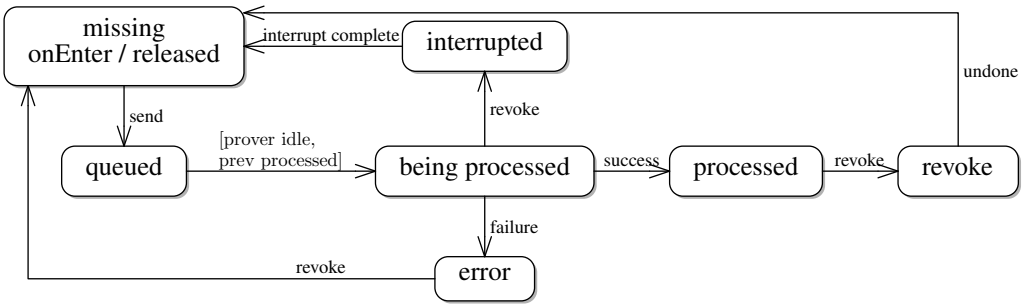


Fig. 4. State Model in the Emulator

Commands start in state *missing*, which indicates that the command is currently owned by the interface. Whenever this state is entered, the prover sends a message *released* to the interface. When the command is sent by the interface, the emulator considers it as *queued*. It is, however, not necessary to create an explicit queue data structure. Instead, the *queued* state has a completion transition [7], which fires spontaneously as soon as the source state can be left. There are two guard conditions: The prover must be idle and the command that precedes the current one in the text must already be processed. The second condition obviously implements a queue-like behaviour. The state *being executed* is left on three events: If the prover reports a *success*, if it reports a *failure*, or if the user interfaces *revokes* the command. If the command is revoked, then the prover needs to be signalled to stop processing the command. The command remains in state *interrupted* until the prover acknowledges by event *interrupt complete* that the execution of the command has been aborted. In this case, the command becomes *missing*, as requested by the interface. When a completely processed command receives message *revoke*, it enters state *revoke*. The emulator sends suitable `undo` commands to the prover, and as soon as they have been executed, the command is released.

3 Software Architecture

This section discusses the software architecture of the user interface that emerges from the considerations of Section 2. Figure 5 gives an overview. The *Host Editor* is a generic text editor that the user employs to enter the proof document. It is extended by a *Display Plugin* that renders the current state of individual commands to the user. Depending on the editor, this functionality may be implemented by special widgets or by markups in the existing display components. The *Infrastructure for Asynchronous Proof Processing* (IAPP) is the core of our system. It implements the mechanisms necessary to support asynchronous proof processing in a reusable, portable manner. Finally, the *Prover Process* communicates with the IAPP using the protocol from Section 2.3. An emulator (Section 2.4) translates the requests to a linear processing model and communicates with the existing Isabelle process.

3.1 Editor Component

Using an existing editor for the text of proof documents has many advantages over a special purpose front-end. The standard features like cut&paste, drag&drop, file management, and syntax highlighting are available without cost, and the user may

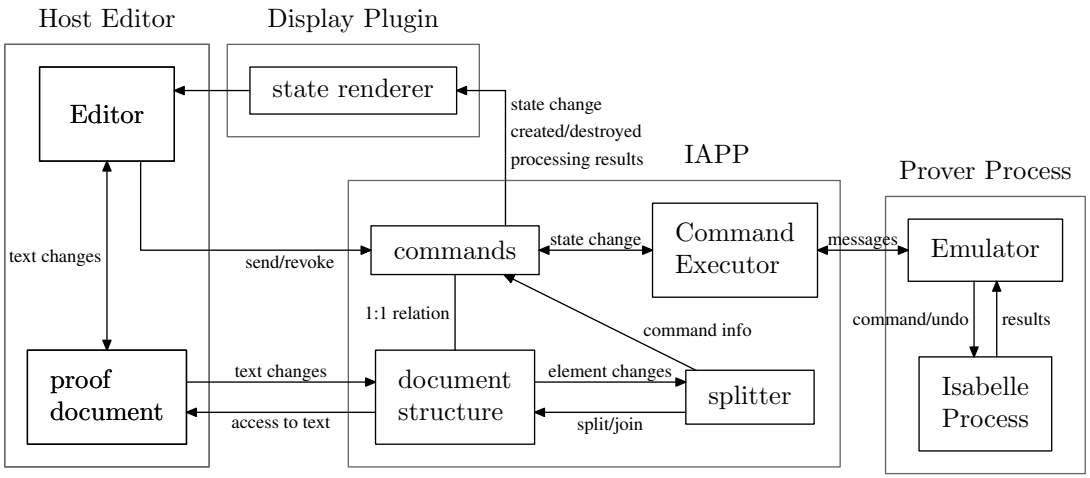


Fig. 5. Software Structure

```
public interface Document {
  void addDocumentListener(DocumentListener l);
  void removeDocumentListener(DocumentListener l);
  String getCharacters(int start, int end) throws DocumentException;
  int getLength();
  char getCharacter(int index) throws DocumentException;
}
public interface DocumentListener {
  void documentChanged(DocumentEvent ev);
}
public class DocumentEvent {
  public Document doc;
  public int start;
  public int removed;
  public int inserted;
}
```

Fig. 6. IAPP Document Abstraction

already be familiar with the handling. The design of the IAPP aims at making minimal assumptions about the editor, in order to allow different alternatives to be evaluated. There are three basic requirements:

- The editor's document content can be accessed.
- The editor's document model implements the Observer [9] pattern.
- The editor can be extended to display new components.

The first two requirements can be made concrete by the Java interfaces that define the expected functionality in the implementation (Figure 6). A `Document` has methods for adding and removing observers [9] of type `DocumentListener`, which are notified about changes to the text. Following the SWT widget set, a change consists of a removing a number of characters and inserting a number of characters. Since the IAPP does not keep a copy of the text, the inserted string is not transmitted.

It is important to point out that the editor does not have to be written in Java. It is also possible to write an adapter that implements the interface but translates the method calls to messages that are sent over some communication channel. The callbacks to the observers take place when the editor process sends a change message.

The editor-specific *state renderer* component displays the progress and result of asynchronous processing. It is notified about all changes to the processing state of the command, and the textual results, for instance error messages, that Isabelle has sent during processing. The design does not specify the exact nature of the display: highlights of commands in the proof document, icons that indicate failure, and a separate display for goals may be suitable. Again, it is possible to write an adapter that translates the method calls into messages and sends them to an external process.

The editor may also generate events *send* and *revoke* (Section 2.2) that change the state of individual commands, and induce the command executor to send them to the prover or have the prover release them. Whether the events are triggered explicitly by the user or a special logic generates them automatically is not specified by the IAPP. We see it as a distinct advantage to be able to experiment with different strategies and evaluate their effect on the usability of the user interface.

3.2 Tracking Document Changes

One of the main challenges in asynchronous proof processing is the maintenance of the document structure as a sequence of commands. Each command is tagged with a unique ID that is used in communication with the prover, such that destroying, creating, and changing commands requires notification of the prover, which due to dependencies may result in extensive and time-consuming proof operations. The textual edits by the user must therefore incur the minimal necessary changes to the document structure. This requirement is in contrast with linear processing, where the splitting of the document can be postponed until the user sends text to the prover. The PGIP architecture [4] contains similar problems, which are approached by letting the prover re-parse the elements affected by an edit. As our analysis shows, it is by no means trivial to decide which elements are affected.

Figure 5 shows a separation of concerns in document maintenance: the syntactic partitioning of the document into *elements* is handled by the *document structure* and *splitter* components. The *command* objects are attached to elements and implement the state model of the IAPP (Section 2.1). The elements of a document are always non-overlapping and cover the complete document. An element offers two operations that maintain this invariant: `splitAt(`*pos*`)` shrinks the target element to end at *pos* and creates a new element that covers the characters from *pos* to the next element. Operation `join()` extends the target element to cover also the subsequent element in the document, and destroys that second element.

The *document structure* and the *splitter* together maintain the partitioning into elements. The document structure is responsible for maintaining the start positions of the elements through deletions and insertions: When text is inserted, the positions of all later elements are increased, when text is removed, they are decreased. The implementation uses a gap-store data structure to make the computation efficient.

The document structure also identifies the elements that are affected by a textual change and reports them to the *splitter*. The splitter has to decide whether the changes lead to splitting or joining elements.

The splitter for Isar proof documents can take advantage of the fact that each command starts with a specific keyword. Whenever a textual change leads to the creation of a keyword, the containing element is split at the position of the keyword. Whenever a change leads to the deletion of a keyword, the element is joined with the previous one. The task is not entirely trivial for two reasons. First, keywords in quoted regions must not lead to a split. There are three kinds of quotes in Isar: Comments (`(*···*)`), inner syntax (`"···"`), and verbatim text (`{*···*}`). The splitter has to maintain for each element those regions that are quoted. The second complication is the interaction with the state of commands: Only *idle* commands can be joined or split, such that the splitter must generate *revoke* events where necessary. Until these requests are acknowledged by *released* events, the splitter cannot proceed. In order to avoid stalling the interface, the splitter itself must work asynchronously. Whenever an element becomes idle, the splitter decides whether it must resume some postponed operation.

We have also considered using a general incremental parsing algorithm (see [10]) to delineate the commands. However, the specialized solution makes it much easier to guarantee that no unnecessary changes to the document structure take place. Also the interaction with the command state cannot be reconciled with existing parsing technology.

Figure 5 shows that the splitter component also attaches information about the recognized keywords to commands. Such information is useful for outline views and for recognizing the category of the command. The effect of undo-operations in Isabelle, for instance, depends on whether the command is a top-level command, a proof command, or a command that finishes a proof (`qed`, `done`, `by`).

3.3 Executing Commands

Executing commands in asynchronous proof processing is more than simply sending selected commands to the prover. It requires negotiating the requests by the user and the prover. The user marks some commands to be ready for processing and reclaims some for further editing; at the same time, the prover may request commands and may release others, guided by the dependencies managed internally. The *command executor* component in Figure 5 reflects this insight: It observes both the state changes of commands and messages from the prover, and decides on the new state of commands and the commands to be sent to the prover.

To make the prover communication more concrete, we have modelled the messages from Section 2.3 as Java method calls between the command executor and the emulator. These classes communicate only through the interfaces defined in Figure 7. The class `CommandID` encapsulates an arbitrary `String`. These interfaces have a second advantage: As soon as Isabelle implements the new protocol natively, we can replace the emulator class with an adapter that implements the interface `AsyncInterface` and translates method calls to message and vice versa.

The logic of the command executor itself is minimal. The command objects

```
public interface AsyncProver {
  void send(CommandID id, String command);
  void revoke(CommandID id);

  void create(CommandID id, CommandID prev);
  void destroy(CommandID id);
}
public interface AsyncInterface {
  void released(CommandID id);
  void request(CommandID id);

  void queued(CommandID id);
  void startProcessing(CommandID id);
  void success(CommandID id);
  void error(CommandID id);
  void result(CommandID id, Result r);
}
```

Fig. 7. Prover/Interface Protocol

from Section 3.2 implement the state model from Section 2.2, i.e. they trigger the appropriate state changes according to the occurring events. The command executor merely handles commands in states *to be send* and *to be revoked* by dispatching messages *send* and *revoke*, respectively, to the `AsyncProver`. Conversely, if the executor receives message *released* from the prover, it triggers event *released* in the command's state machine.

Handling *request* messages touches on questions of usability. In the current implementation, the executor triggers the event *send* on the command, such that in the next step, the executor is informed about the command being ready for sending. As a result, commands that the prover requires for processing are sent automatically. More sophisticated strategies may take the last edits by the user into consideration.

The remaining messages in interface `AsyncInterface` provide information on the processing state of individual commands. The first four messages are explained by the nested state machine in Figure 3. The `Result` object in the last message encapsulates one output element from Isabelle's stream. Among the possible elements are new proof states, and error, warning, and tracing messages. The executor stores this auxiliary information in the objects representing commands, from where it is retrieved by the *state renderer*.

3.4 An Minimal Interface

We have implemented a minimal user interface to evaluate the usability of theorem proving applications build on top of IAPP. Since currently no editor is a clear favorite for a user interface [18], we have chosen to use basic Swing widgets for the prototype. Figure 8 shows the result.

The middle pane shows the text of the proof document. The highlights indicate the processing status of individual commands. Since the emulator (Section 2.4)

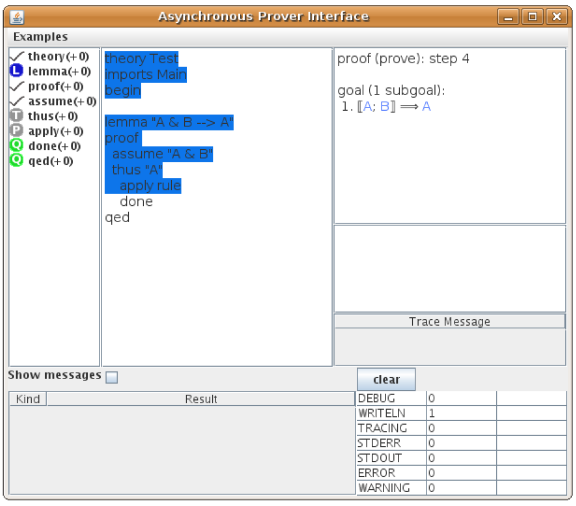


Fig. 8. Screenshot of Example Interface

implements a linear processing strategy, they resemble the locked region in conventional interfaces [1,6]. The left pane shows an outline view that is created in a straightforward manner by observing the document structure and the information attached to commands by the splitter (Section 3.2). The outline immediate reflects edits by the user: When a new keyword is entered, a new item appears in the outline; when a keyword is destroyed, one item disappears.

The right pane resembles the standard output windows of the ProofGeneral [1]. However, its function is very much different. The standard windows follow the command processing by the prover, i.e. they contain the results of the last processed command. Wenzel has pointed out [17,18] that in the context of asynchronous proof processing, this behaviour is not sensible. Instead, the output widgets display the results attached to the command that the caret is currently in. If that command has not been processed, the preceding command is used.

The lower part allows the user to observe the communication between interface and prover. Summary of the counts of message types are shown on the right. It is also possible to limit the number of handled messages, for instance to avoid flooding the interface with an excessive number of tracing outputs.

We are currently exploring several modes of user interaction. The first mode emulates the linear processing behaviour. Key combination Ctrl-N sends the first idle command. Ctrl-U revokes the last non-idle command, which effectively results in a one-step undo. There is, however, an important difference to the ProofGeneral interface: when the final proof command (`qed`,`done`,`by`) of a theory-level statement is undone, then the preceding proof-steps are re-executed, which for the user is a much more consistent behaviour than revoking the entire proof. Ctrl-Enter sends or revokes the command that the caret is in, depending on the command's state. The effect is that the emulator executes or undoes commands until the selected command is reached.

Other strategies for sending commands are possible. In continuous proof processing [17], for instance, the interface would send all commands that the user is not currently editing. When the user hits a key within a sent command, the command is revoked and will not be sent again until the caret leaves it; at that point, it is

sent automatically. If a command is found to contain an error, it would be revoked and left idle until the user has edited it again. Even though some commands would be executed only speculatively, with modern multi-core processors the user would not notice an increased answer time of the interface.

At present, no final answer can be given about the best strategy to increase usability. However, the IAPP simplifies experimentation since the user interface only needs to generate *send*/*revoke* events, while the IAPP carries out the request in the background.

4 Related Work

The PGIP protocol [2,4,3] defines a standard for communication between interactive provers and user interfaces. It is a generalization of the text-based mechanisms of the ProofGeneral [1]. The supporting architecture PGKit is message-based: the prover and display components exchange messages with a central broker. The broker maintains the proof documents currently being edited and negotiates changes with both display components and provers. The proof documents are stored as the textual commands in the provers' native languages, the document structure is represented by XML markups.

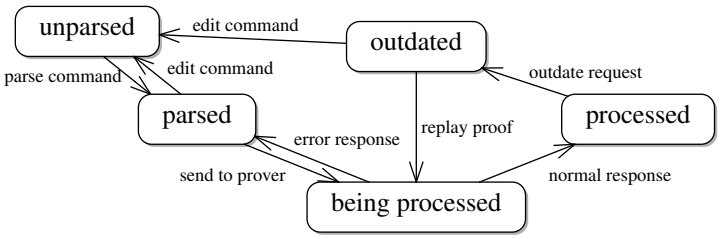


Fig. 9. PGIP Command States

Figure 9 shows the state model for individual commands [3]: Text that has been entered or modified is considered *unparsed*. It is submitted to the broker by the display components; the broker sends unparsed text fragments to the prover and receives the structure in a *parse command* in return. Parsing is expected to be efficient and to occur after a brief delay. The user can induce the broker to send a parsed command to the prover, in which case the command enters state *being processed*. When the prover sends the acknowledgement that the command has been processed successfully, the state changes to *processed*. If an error occurs, the command reverts to state *parsed*. The state *outdated* is used to model undo/redo mechanisms.

The PGKit architecture is thus built around a central broker that takes control of the processing. It also manages dependencies between commands to decide which commands need to be processed and outdated [4, Section 3.2]. The state model for commands implies that the broker decides which commands need to be processed, and it knows which are currently being processed. Observe for comparison that the PGIP model resembles the state model of the emulator (Section 2.3) rather than that of the IAPP itself (Section 2.2). The second distinction from the IAPP is the requirements that the PGIP places on the provers: the prover has to parse

commands and provide dependency lists, both of which may require substantial changes to the software structure of existing provers. The IAPP, on the contrary, aims at assigning minimal responsibilities to the prover. The rationale is that fitting asynchronous processing into existing provers will be much simplified if the implementation can take the existing software structure into account as much as possible. In particular, dependencies and the order of processing remain in the control of the prover.

The document-centric approach to interactive proof has been developped further by Wagner et al. [16,8] into the proof assistance system PlatΩ for authors of mathematical texts. PlatΩ allows users to edit a type-set, printable document that is either annotated [16] or written in a controlled language [8]. From the annotations or syntax tree, respectively, PlatΩ generates a formal representation that is checked by the Ωmega proof system [15]. To avoid unnecessary re-checking, PlatΩ analyses the structural changes to the text caused by user edits and translates them into corresponding changes of the formal representation.

PlatΩ shares with asynchronous proof processing the intention of checking the proof document in the background and re-processing the document incrementally upon user edits. It differs significantly from the IAPP architecture in that the syntactic document structure and the dependencies between its parts are analyzed by PlatΩ, rather than the prover, and it is the PlatΩ system that decides about re-checking proofs; furthermore, the approach is tightly integrated with the Ωmega proof system. The IAPP, by contrast, seeks to provide a minimal infrastructure for a prover to offer asynchronous processing, and it delegates decisions about parsing and presentation to the prover as much as possible.

5 Conclusion

We have presented an infrastructure for asynchronous proof processing, IAPP. It enables user interfaces and provers to communicate in a message-based style and makes minimal assumptions on the processing of individual commands by the prover. In particular, the IAPP does not assume that the prover can parse commands and report dependencies between commands. Provers that wish to support the IAPP protocol can therefore take their decisions according to the existing software structure. In a transition phase, it is simple to support the IAPP protocol by a linear-processing proof engine using a small emulator component that can be implemented in either the user interface or the prover.

IAPP addresses the two main concerns of asynchronous proving: A stable partitioning of the textual proof document into non-overlapping commands and an explicit state model for commands that synchronizes the access to commands between user interface and provers. The state model also defines directly the communication protocol between user interface and prover.

Finally, our design makes the processing within the IAPP entirely independent of the text editor that serves as a front-end. This makes it possible to experiment with different editors, to maintain legacy systems in a transition phase, and to move on to new environments as they emerge. The fundamental capabilities of asynchronous proof processing are equally reliable on any of them.

References

[1] David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, number 1785 in LNCS, 2000.

[2] David Aspinall and Christoph Lüth. ProofGeneral meets IsaWin. In *User Interfaces for Theorem Provers (UITP '03)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2003. (to appear).

[3] David Aspinall, Christoph Lüth, and Ahsan Fayyaz. Proof general in eclipse: System and architecture overview. In *Eclipse Technology Exchange Workshop at OOPSLA 2006*, 2006.

[4] David Aspinall, Christoph Lüth, and Daniel Winterstein. Parsing, editing, proving: The pgip display protocol. In *International Workshop on User Interfaces for Theorem Provers 2005 (UITP'05)*, 2005.

[5] David Aspinall, Christoph Lüth, and Burkhart Wolff. Assisted proof document authoring. In *Mathematical Knowledge Management 2005 (MKM '05)*, number 3863 in Springer LNAI, pages 65–80, 2005.

[6] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symbolic Computation*, 25:161–194, 1998.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[8] Dominik Dietrich, Ewaryst Schulz, and Marc Wagner. Authoring verified documents by interactive proof construction and verification in text-editors. In *Intelligent Computer Mathematics*, volume 5144 of *LNAI*, pages 398–414. Springer, 2008.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[10] Holger Gast. An architecture for extensible Click'n Prove interfaces. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07, August 2007.

[11] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2nd edition, 1999.

[12] P. Rudnicki. An overview of the mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, Bastad, 1992.

[13] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented Software Architecture: Patterns for concurrent and networked objects*, volume 2. Wiley & Sons, 2000.

[14] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3rd edition, 1998.

[15] Jörg H. Siekmann, Christoph Benzmüller, Vladimir Brezhnev, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuel Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Jürgen Zimmer. Proof development with Ωmega. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 144–149, London, UK, 2002. Springer-Verlag.

[16] Marc Wagner, Serge Autexier, and Christoph Benzmüller. PlatΩ: A mediator between text-editors and proof assistance systems. In *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, volume 174(2) of *ENTCS*, pages 87–107. Elsevier, 2007.

[17] Makarius Wenzel. Asynchronous processing of proof documents: Rethinking interactive theorem proving. Talk given at the TYPES topical workshop "Math Wiki", Edinburgh, October 2007.

[18] Makarius Wenzel. personal communication, 2008.

[19] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.

Towards Merging PlatΩ and PGIP

David Aspinall

LFCS, School of Informatics, University of Edinburgh
Edinburgh, U.K. (homepages.inf.ed.ac.uk/da)

Serge Autexier

DFKI GmbH & FR Informatik, Universität des Saarlandes
66123 Saarbrücken, Germany (www.ags.uni-sb.de/~serge)

Christoph Lüth

DFKI GmbH & FB Informatik, Universität Bremen
28359 Bremen, Germany (www.informatik.uni-bremen.de/~cxl)

Marc Wagner

FR Informatik, Universität des Saarlandes
66123 Saarbrücken, Germany (www.ags.uni-sb.de/~marc)

Abstract

The PGIP protocol is a standard, abstract interface protocol to connect theorem provers with user interfaces. Interaction in PGIP is based on ASCII-text input and a single focus point-of-control, which indicates a linear position in the input that has been checked thus far. This fits many interactive theorem provers whose interaction model stems from command-line interpreters. PlatΩ, on the other hand, is a system with a new protocol tailored to transparently integrate theorem provers into text editors like TEXMACS that support semi-structured XML input files and multiple foci of attention. In this paper we extend the PGIP protocol and middleware broker to support the functionalities provided by PlatΩ and beyond. More specifically, we extend PGIP (i) to support multiple foci in provers; (ii) to display semi-structured documents; (iii) to combine prover updates with user edits; (iv) to support context-sensitive service menus, and (v) to allow multiple displays. As well as supporting TEXMACS, the extended PGIP protocol in principle can support other editors such as OpenOffice, Word 2007 and graph viewers; we hope it will also provide guidance for extending provers to handle multiple foci.

Keywords: PlatΩ, Proof General, Mediator, Protocol, PGIP

1 Introduction

Proof General [2,3] is widely used by theorem proving experts for several interactive proof systems. In some cases, there is no alternative interface; in others, the alternatives are little different. Yet the limitations of Proof General are readily apparent and reveal its evolution from simple command line systems. For one thing, the input format is lines of ASCII-text, with the minor refinement of supporting Unicode

or TeX-like markup. The presentation format during interaction is the same. For another thing, the proof-checking process has an overly simple linear progression with a single point-of-focus; this means that the user must explicitly undo and redo to manage changes in different positions in the document, which is quite tedious.

Meanwhile, theorem provers have increased in power, and the ability for workstations to handle multi-threaded applications with ease suggests that it is high time to liberate the single-threaded viewpoint of a user interface synchronised in lockstep to an underlying proof-checking process. Some provers now provide multiple foci of attention, or several prover instances might be run in concert. Text editors, too, have evolved beyond linear ASCII-based layout. The scientific WYSIWYG text editor TEXMACS, for example, allows editing TEX and LATEX-based layout, linked to an underlying interactive mathematical system.

Significant experiments with theorem proving using richer interfaces such as TEXMACS have already been undertaken. In particular, the PLATΩ system [7,4] mediates between TEXMACS and the theorem prover ΩMEGA. While experiments with individual systems bring advances to those specific systems, we believe that many parts of the required technology are generic, and we can benefit from building standard protocols and tools to support provers and interfaces. The aim of this paper, then, is to integrate lessons learned from the PLATΩ system prototype with the mainstream tool Proof General and its underlying protocol PGIP, putting forward ideas for a new standard for theorem prover interfaces, dubbed here *PGIP 2*. Specifically, our contributions are to combine ideas of state-tracking from PGIP with semi-structured document models and menus as in PLATΩ, and to add support for possibly distributed multiple views.

1.1 PG Kit system architecture

The *Proof General Kit* (PG Kit) is a software framework for conducting interactive proof. The framework connects together different kinds of components, exchanging messages using a common protocol called *PGIP* . The main components are interactive provers, displays, and a broker middleware component which manages proof-in-progress and mediates between the components. Fig. 1 shows the system architecture; for details of the framework, we refer to [3].

The PG Kit architecture makes some assumptions and design decisions about the components. Generalising from existing interactive provers (such as Isabelle, COQ, or *Lego*), we assume that provers implement a single-threaded state machine model, with states *toplevel*, *file open*, *theory open* and *proof open*. Displays, on the other hand, are assumed to be nearly stateless. Through the display, the user edits the proof text and triggers prover actions, e.g., by requesting that a part of the proof script is processed. Abstractly, the broker mediates between the nearly stateless display protocol PGIP_D, and the stateful prover protocol PGIP_P; it keeps track of the prover states, and translates display state change

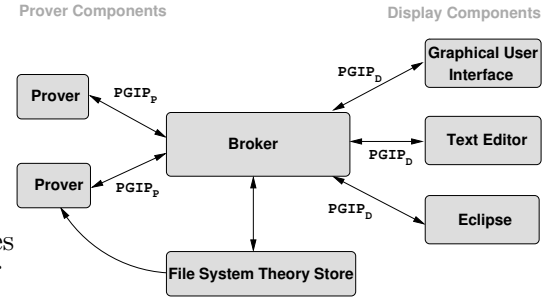


Fig. 1: PG Kit System Architecture

requests into sequences of concrete prover commands, which change the state of the prover as required.

1.2 PLATΩ *system architecture*

The aim of the PLATΩ system is to support the transparent integration of theorem provers into standard scientific text editors. The intention is that the author can write and freely edit a document with high-quality typesetting without fully imposing a restricted, formal language; proof support is provided in the same environment and in the same format. The PLATΩ system is the middleware that mediates between the text editor and the prover and currently connects the text editor TEXMACS and the theorem prover ΩMEGA. For the architecture of the system, see Fig. 2.

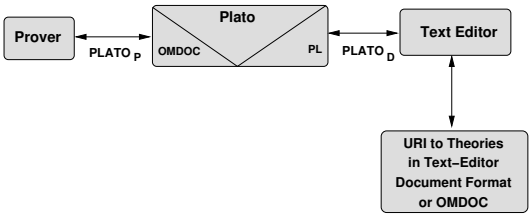


Fig. 2: PLATΩ System Architecture

1.3 Outline

The rest of the paper is structured as follows. In Section 2 we give a scenario for conducting a simple proof, and describe the interaction processes in PLATΩ and in Proof General. Section 3 begins discussion of our proposal to merge the two architectures, explaining how to extend PGIP to support documents with more structure and multiple points of focus. Section 4 describes how to extend PGIP with a menu facility like that provided in PLATΩ, and Section 5 describes how to handle multiple displays, extending what is presently possible in PLATΩ. To complete our proposal, Section 6 explains how we can reconcile semi-structured documents with PGIP flat-structured documents, to connect theorem provers based on classical flat structured procedural proofs with our enhanced middleware for a richer document format. Section 7 discusses related work and future plans.

2 Interaction in PlatΩ and Proof General

We illustrate the overall functionality and workflow of PLATΩ and PG Kit with the following example, in which student Eva wants to prove the commutativity of addition in the standard Peano axiomatisation. Eva is typing this proof in a text editor, TEXMACS or EMACS, and receives assistance from a *theorem prover*, ΩMEGA or Isabelle, for PLATΩ and PG Kit respectively (cf. Fig. 3).
Eva's authoring process splits into the following five phases:

Phase 1. After having specified the theory and the conjecture

$$\forall x, y.x + y = y + x \tag{1}$$

in the text editor the document is passed to the theorem prover.

Phase 2. Eva begins to prove the conjecture. She does an induction on x and gets stuck with the subgoals: (1a) $0 + y = y + 0$ and (1b) $(z + y = y + z) \Rightarrow (s(z) + y = y + s(z))$.

Phase 3. She quickly realises that two lemmas are needed. Hence, she adds the

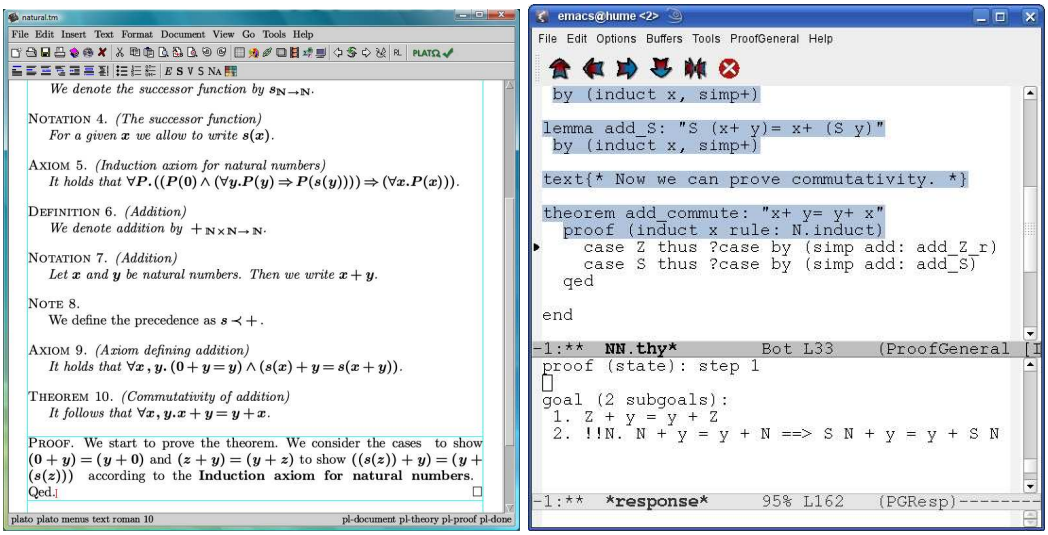


Fig. 3. Formalisation of the example scenario in TEXMACS and PG Kit.

following two lemmas somewhere in the document:

$$\forall x.0 + x = x + 0 \tag{2}$$

$$\forall x, y.(x + y = y + x) \Rightarrow (s(x) + y = y + s(x)) \tag{3}$$

Phase 4. Eva then tackles these lemmas one by one: for each, doing an induction on x and simplifying the cases proves the lemmas.

Phase 5. Eva then continues the proof of (1) by applying both lemmas to (1a) and (1b) respectively, which completes the proof.

2.1 PLATΩ

PLATΩ uses a custom XML document format called PL to connect to the text editor. The PL document contains markup for theories, theory items and linear, text-style proofs, and also notation definitions for defined concepts. Formulas in axioms, lemmas and proofs are in the standard, non-annotated LaTeX-like syntax of TEXMACS. To connect to the theorem prover, PLATΩ uses OMDoc for the hierarchical, axiomatic theories and another custom XML format (TL) for the proofs. [1] PLATΩ holds the representations simultaneously, with a mapping that relates parts of the PL document to parts of the OMDoc(TL) document; a major task of the system is to propagate changes between the documents and maintain the mapping.

The text editor interface protocol ($\mathsf{PLATO}_\mathsf{D}$, see Fig.2) uses XML-RPC, with methods for complete document upload, service requests for specific parts of the PL document, and the execution of specific prover commands. On receiving a new document version, PLATΩ parses the live formulas using the document notations, producing *OpenMath* formulas. If a parse error occurs, an error description is returned to the editor. Otherwise PLATΩ performs an XML-based difference analysis [9] against the old PL document, resulting in a list of XUpdate modifications, [2]

[1] The next version of PLATΩ will use the OMDoc format for proofs, though still with ΩMEGA specific justifications for proof steps.

[2] see xmldb-org.sourceforge.net/xupdate/

which are transformed into XUpdate modifications for the OMDoc(TL) document.

The interface to the theorem prover (PLATO$_P$) also uses XML-RPC, with methods for applying XUpdate modifications, service requests for parts of the OMDoc(TL) document, and executing specific prover commands. Applying an XUpdate modification may result in an error (e.g. a type error) or is simply acknowledged; either response is then relayed by PLATΩ to the display as an answer to the corresponding document upload method call. The result of a service request is a menu description in a custom XML format. That menu is relayed to the display as a reply to the corresponding service request, rendering *OpenMath* formulas in the menu into TEX$_{\text{MACS}}$ syntax using the notation information already used for parsing.

The result of executing a menu action is a list of XUpdates, which can either patch the menu (for lazy computation of sub-menus), or patch the document (for instance, inserting a subproof). PLATΩ transforms these OMDoc(TL) patches into PL patches and renders occurring *OpenMath* formulas into TEX$_{\text{MACS}}$ markup before sending the patch to the text editor.

Semantic Citation. A characteristic of PLATΩ is that everything that can be used comes from a document. Hence, there is a specific mechanism to "semantically" cite other TEX$_{\text{MACS}}$ documents (see Fig. 2); these appear as normal citations in the editor but behind the scenes, are uploaded into PLATΩ, which then passes them to ΩMEGA. As a consequence, PLATΩ does not allow reuse of theories that are predefined in the theorem prover.

We now illustrate PLATΩ by describing the phases of the example scenario.

Phase 1. First, the whole document is passed from TEX$_{\text{MACS}}$ to PLATΩ which extracts the formal content of the document including notational information to parse formulas. From the document, PLATΩ builds up the corresponding OMDoc theories and passes them as an XUpdate to ΩMEGA, which builds up the internal representation of the theory and initialises a proof for the open conjecture.

Phase 2. To start the proof of the theorem, Eva requests a menu from ΩMEGA, which returns a menu that lists the available strategies. Eva selects the strategy `InductThenSimplify`, which applies an induction on x to the open conjecture, simplifies the resulting subgoals terminates with the two open subgoals. This partial proof for Theorem (1) inside ΩMEGA is compiled into patch description and then passed to PLATΩ. PLATΩ transforms it into a patch for TEX$_{\text{MACS}}$ by rearranging the obtained tree-like subproof representation into a linear, text-style proof representation using pseudo-natural language, and rendering the formulas using the memorised notational information.

Phase 3. After the two lemmas are written in the document, the whole document is uploaded and, after parsing, the difference analysis computes the patch to add the two lemmas. This is transformed into a patch description to add their formal counter-parts as open conjectures to the theory and sent to ΩMEGA. ΩMEGA, in turn, triggers the initialisation of two new active proofs.

Phase 4. Eva uses for both lemmas the strategy `InductThenSimplify` (again suggested by ΩMEGA in a menu) which succeeds in proving them. The resulting proof descriptions are again transformed by PLATΩ into proof patches for the document

and both lemmas are immediately available in the ongoing proof of Theorem (1).

Phase 5. ΩMEGA proposes in a menu to apply the lemma (2) to the subgoal (1a) and the lemma (3) to the subgoal (1b). Eva selects these suggestions one by one, which then completes the proof inside ΩMEGA. Subsequently, only the proof patch descriptions are transformed into patches for the TEXMACS document as before.

2.2 Proof General

Unlike OMDOC, PGIP is not a proof format, nor does the PG Kit prescribe one. Instead, PGIP uses proofs written in the prover's native syntax, which are lightly marked up to exhibit existing implicit structure. The mark up divides the text into text spans, corresponding to prover commands which can be executed one-by-one in sequence. Different commands have different mark up, characterising e.g., start of a proof, a proof step, or (un)successful completion of a proof, as in:

```
<opengoal>theorem add_commute: &quot;x+ y= y+ x&quot;</opengoal>
  <proofstep>proof (induct x rule: N.induct)</proofstep>
```

Elements like <opengoal> do not carry an inherent semantics (and they cannot be sent to the prover on their own), they merely make it clear that e.g. the command **theorem** add_commute: "..." starts the proof. Each of these text spans has a state; the main ones are parsed, processed and outdated. Proving a given theorem means to turn the whole proof into the processed state, meaning that the prover has successfully proved it. Returning to the scenario, we discuss the flow of events between the Emacs display, the PG Kit broker and the Isabelle prover.

Phase 1. Eva starts with an initial plain text Isabelle file, giving the definitions for the natural numbers, addition and the conjecture. She requests the file to be loaded, causing the broker to read it and send the contents to Isabelle for parsing. While this happens, the display shows the unparsed text to give immediate feedback. Isabelle returns the parsed file, which is then inserted into the Emacs buffer.

Phase 2. Eva now wants to prove the conjecture. She requests the conjecture to become processed so she can work on the proof (a command <setcmdstatus> is sent to the broker). This triggers sending a series of commands to Isabelle, ending with the conjecture statement. Isabelle answers with the open subgoal, which is then shown on the display.

Eva attempts proof by induction. She writes the appropriate Isabelle commands (proof (induct x rule: N.induct)). The new text is sent to the broker and then on to Isabelle for parsing. Once parsed the broker breaks the text into separately processable spans (here, only one), which is sent back to the display. Now Eva asks for the proof step to be processed, which sends the actual proof text to Isabelle, which answers with two open subgoals.

Phase 3. Realising she needs additional lemmas, and knowing Isabelle's linear visibility, Eva knows she has to insert two lemmas before the main theorem she is trying to prove. Since she cannot edit text which is in state *processed*, she first requests the text to change state to *outdated*. This causes a few undo messages to be sent to the prover to undo the last proof commands, resetting Isabelle's state back to where it has not processed the start of the main proof yet. Eva then inserts

	PlatΩ Display	PG Kit Display	*PGIP 2* Display
Document format	XML	Plain text	XML
Document syntax	TeXmacs	ASCII	Generic
Change protocol	XUpdate	PGIP_D	XUpdate
Change management	Dynamic Notation	Provided by prover	Provided by prover or display
Operations supported	Context-dependent menus	Global menus, typed operations	Context-dependent menus, typed operations

Table 1. Summary of differences between the Display Interfaces of PlatΩ and PG Kit

	PlatΩ Prover	PG Kit Prover	*PGIP 2* Prover
Document format	XML	Plain text	XML
Document syntax	OMDoc	Native prover syntax	Generic
Change protocol	XUpdate	PGIP_P	XUpdate
Change management	Provided by Maya	Provided by Prover	Provided by Prover
Prover support	ΩMEGA	Generic (Coq, Isabelle, etc)	Generic (Coq, Isabelle, ΩMEGA, etc)
Operations supported	Context-dependent menus	Global menus, typed operations	Context-dependent menus, typed operations

Table 2. Summary of differences between the Prover Interfaces of PlatΩ and PG Kit

the needed lemmas in the document, and has them parsed as before.

Phase 4. Eva processes the lemma, and sees a message indicating that the proof worked. She finishes the other lemma similarly.

Phase 5. Eva returns to the main proof, editing the induction proof by inserting the induction base and induction step. Fig. 3 (right) shows the Emacs display at this point: the window is split in two parts, with the proof script in the upper part and the prover responses displayed below. The top portion of the proof script is blue, showing it has been processed, indicating the linear point of focus. After the induction step succeeds, Eva closes the proof with the command `qed`, which registers the theorem with the authorities. By turning the state of this closing command to *processed*, the proof is successfully finished.

3 Semi-Structured Documents

We have now seen how PlatΩ and the PG Kit handle documents. The architecture is similar: a central component handles the actual document, managing communication with the prover on one side and a user-interface component on the other side. The main differences are technical, summarised in the first two columns of Tables 1 and 2. Given the similarity, the question naturally arises: can we overcome these differences and provide a unified framework? This section will tentatively answer in the positive by extending PGIP on the prover side with the necessary new concepts (Sec. 3.1) and multiple foci (Sec. 3.2), and by using XUpdate pervasively on the display side (Sec. 3.3). The right-most columns of Tables 1 and 2 show the technical unification for the proposed *PGIP 2*.

3.1 Document Formats

The two different *document formats* can both be treated as arbitrary XML, with the difference that for PlatΩ and OMDoc, there is deep structure inside the proof

```
<assertion>theorem add_commute: &amp;quot;x+ y= y+ (x::N)&amp;quot;
 <block objtype="proof body">
  <proofstep>proof (induct x rule: N.induct)</proofstep>
  <proofstep>case Z</proofstep><assertion>thus ?case
   <block objtype="proof body"><endproof status="proven">by (simp add: add_Z_r)</endproof>
   </block></assertion>
  <proofstep>case S</proofstep><assertion>thus ?case
   <block objtype="proof body"><endproof status="proven">by (simp add: add_S)</endproof>
   </block></assertion>
  <endproof status="proven">qed</endproof></block></assertion>
```

Fig. 4. Excerpt from the short example proof, marked up with *PGIP 2* (edited slightly for readability).

script (i.e., inside goals, proof steps etc) whereas in the case of PG Kit, there is only a shallow XML structure where the proof script is mainly plain text. To overcome this difference, we allow *PGIP 2* proof scripts to contain arbitrary marked-up XML instead of marked-up plain text, turning the document into a proper XML tree. Here is the present PGIP schema, excerpted and slightly simplified: [3]

```
opentheory  = element opentheory { thyname_attr, parentnames_attr?, plaintext }
closetheory = element closetheory { plaintext }
theoryitem  = element theoryitem { objtype_attr, plaintext }
openblock   = element openblock { objtype_attr, plaintext }
closeblock  = element closeblock { }
opengoal    = element opengoal  { thmname_attr?, plaintext }
proofstep   = element proofstep { plaintext }
closegoal   = element closegoal { plaintext }
```

The proposed *PGIP 2* amends this as follows, again excerpted:

```
theory           = element theory     { thyname_attr, parentnames_attr?, any }
theoryitem       = element theoryitem { objtype_attr, any }
block            = element block      { objtype_attr, xref_attr?, any }
assertion        = element assertion  { thmname_attr?, id_attr?, any }
proofstep        = element proofstep  { xref_attr?, any }
endproof         = element endproof   { xref_attr?, proofstatus_attr?, any }

id_attr          = attribute xml:id
thmname_attr     = attribute thmname { xml:id }
thyname_attr     = attribute thyname { xml:id }
xref_attr        = attribute xref
proofstatus_attr = attribute ("proven"|"assert"|"unproven")

any              = ( text | anyElement ) *
anyElement       = element * { ( attribute * { text } | any ) * }
text             = element text { plaintext }
```

There are two major changes here: (i) arbitrary XML can occur where before only text was allowed; of course, the prover must understand whatever XML syntax is used here (e.g. ΩMEGA can understand OMDOC); (ii) instead of a flat list structure, we now use a proper tree; that is, a theory is not everything between an <`opentheory`> and <`closetheory`> element, but the contents of the <`theory`> element; and similarly, a proof is not everything between <`opengoal`> and <`closegoal`>, but the contents of the <`block`> element of type `proofbody` that belongs to an <`assertion`> element. The <`endproof`> element replaces <`closegoal`> and can be annotated with status information about the proof `proven`, `assert`, or `unproven`. Another extension is the corresponding attributes `xml:id` for the <`assertion`>, and `xref` for the <`block`> elements, which allow assertions to refer to proofs which are elsewhere in the document, and not directly following the assertion. These attributes are optional, and may only appear in the display protocol (i.e., between displays and the broker); we assume that provers always

[3] This XML schema is written in RELAX NG, which can be read much as a BNF grammar, with non-terminals named on the left and `element` and `attribute` introducing terminals; see http://relaxng.org/.

expect proof scripts to be in linear order, and it is the responsibility of the broker to rearrange them if necessary before sending them to be checked.

Furthermore, the broker must be able to divine the structure in an OMDoc proof; e.g., the ΩMEGA prover or a component acting on its behalf must answer parse requests, and return XML documents using these elements. The revised version of our example proof with the *PGIP 2* markup is shown in Fig. 4.

3.2 Multiple Foci

The present PGIP prover protocol imposes an abstract state machine model which the prover is required to implement. ΩMEGA can be made to fit this model, but beyond that provides multiple foci. By this we mean that it can keep track of more than one active proof at a time and switch between them. Ignoring this would lose potential benefits (such as the ability to use a natively multi-threaded implementation of the prover) unnecessarily, and it is easy to accommodate into PGIP: we merely need to add an attribute to the prover commands to identify the focus. Some of these attributes already exist for the display protocol, where files are identified by a unique identifier (`srcid`). By adding unique identifiers also for theories and proofs, the prover can identify which ongoing proof a proof step belongs to, and use the appropriate thread to handle it. To allow fall-back to the simple case, we need a prover configuration setting to declare if multiple foci are available.

3.3 XUpdate

In the PGIP_D protocol, changes in the document are communicated using specialised commands <`createcmd`> and <`editcmd`> from the display to the broker, and <`newcmd`>, <`delcmd`> and <`replacecmd`> from the broker to the display (so the protocol is asymmetric). We can rephrase this in terms of XUpdate; the unique identifier given by the broker to each command contained in the `cmdid` attribute allows use to easily identify an object by the XPath expression `*[cmd=`*c*`]`. The key advantages of XUpdate are that it is standard, symmetric, and allows several changes to be bundled up in one <`xupdate:modifications`> packet that is processed atomically, adding a transaction capability to the display protocol.

Strict conformance to this protocol requires the displays to calculate or track differences, i.e., send only the smallest update. Not all displays (editors) are that sophisticated, and it is unrealistic to expect them to be; a basic design assumption of PG Kit is that the broker should contain the intelligence needed to handle proof documents, and displays should be easy to implement. Hence, displays can send back the whole document as changed, and expect the broker to figure out the actual differences (*whole-document* editing) using the XML difference mechanism from [9] that can take some semantics into account as already used by PLATΩ.

3.4 Protocols

The underlying transport protocol of PGIP is custom designed, because communication with an interactive prover fits no simple standard single-request single-response protocol: the prover asynchronously sends information about the proof as it is progressing, and the ability to send out-of-band interrupts to the prover is crucial.

However, on the display side these reasons do not apply; we might use XML-RPC or even plain HTTP in a REST architecture. REST (representational state transfer [6]) is an architecture style for distributed applications that, in a nutshell, states that an application should provide resources, which are addressed using URIs and manipulated using the four basic operations of creating, reading, updating and deleting ("CRUD"). The resources provided by the broker are as follows:

- The broker itself, with the list of all known provers, all loaded files, a global menu, and global configurations as attributes;
- each prover is a resource, with its status (not running or running, busy, ready, exited) as attributes, preferences for this prover, all identifiers for the prover, messages sent by the prover, its proof state, and prover-specific configurations such as types, icons, help documents, and a menu;
- and each file is a resource, containing the document as a structured text, and the status (saved or modified) as attributes.

Clients affect changes to the document by the XUpdate messages above, and trigger broker actions by changing the attributes. For example, to start a prover, the client will change the status of the prover resource from not running to running. Here, bundling up changes into one XUpdate modification becomes useful, as it allows displays to send several changes to the document resource in one transaction.

In the REST view, changes in the document produce a new version of the document; special links will always point to the latest version of the document, but may require the client to refresh them. This allows multiple displays; we will exploit this in Sec. 5. This REST-style interface is an *alternative* to the stateful protocol using PGIP or XML-RPC; in the long run, the broker will support both.

4 Service Menus

PGIP 2 supports context-sensitive service menus in the display for the interaction with the prover. The user can request a menu for any object in the document; through the broker this triggers menu generation in the prover for the formal counterparts of the selected object. It remains to fix a format for menu descriptions.

Traditionally, menus are fully specified and include *all* submenus and the leafs are *all* actions with *all* possible actual arguments. Executing an action triggers modifications of the document and the menu is closed. For theorem provers, computing all submenus and action instances can be expensive and unduly delay the appearance of the menu. For example, a menu entry for applying a lemma would contain as a submenu all available lemmas, and for each lemma, all possibilities to apply it in the current proof situation. Once the user makes a choice, the other possibilities are discarded. So on-demand computation of submenus is desirable.

The PLATΩ system allows lazy menus, where actions executed in a menu can generate a submenu. The entire menu is modified by replacing the leaf action by the generated submenu. We adapt this model for *PGIP 2* also. However, not all displays are able to incorporate changes to live menus; therefore we do not impose the partial menu representation. Instead, the display specifies in the service request whether it will accept a lazy menu.

The description language for these menus is:

```
menu     = element menu { id,  name, for_attr, ((menu|action)+ | error) }
action   = element action { id, name, argument* }
argument = element argument { id, name, custom }
custom   = element custom { id, alt, any }
error    = element error { id, text }
```

(using the `any` element from above). A menu entry is rendered by its name and an action is rendered by its name and its arguments. Arguments are rendered with the given custom object, e.g., an *OpenMath* formula or some standard TEXMACS markup. The `alt` attribute provides a fallback ASCII representation in case the custom object content cannot be displayed.

When the user chooses an action, it is executed on the specified arguments. The result of the action may be an XUpdate patch to the document. This is sent to the broker and then on to the display, which incorporates the patch and closes the menu. Alternatively it is a patch for the menu only: in this case the action is replaced in the menu by the new submenu. If a submenu is empty, i.e., there are no possibilities to refine the abstract action, then the submenu consists solely of an error that describes the cause, which should be displayed inside the menu.

Example 4.1 We illustrate the interactions when requesting a menu for a display that is able to deal with partial menus.In **Phase 5** of the scenario, Eva requests a menu for the subgoal (1a) $0 + y = y + 0$.

Menu Request: The menu is requested for a specific XPath of the document and the broker maps it to a menu request to the prover for the corresponding formal object, that is, the open goal that corresponds to (1a) $0 + y = y + 0$. The prover generates a top-level menu with the actions "Apply Axiom or Lemma", "Apply Tactic" and returns that to the display via the broker.

Lazy Menu Deployment: Selecting "Apply Axiom or Lemma" triggers computing a submenu containing all available axioms and lemmas. That submenu is sent as an XUpdate patch to the display to replace the action "Apply Axiom or Lemma". Selecting Lemma (2) triggers the prover action that computes the possible ways to apply the lemma on the open goal. In this case the resulting submenu has a few entries for the cases where the lemma is applied from left to right and one case for the application of the lemma from right to left. The submenu is sent as an XUpdate patch to the display to replace the action "Apply Lemma (2)".

Menu Action Execution: The final top level action execution triggers applying the specific instance of the Lemma in the prover, modifying the formal proof. The modification is propagated via the broker to the display, either as an XUpdate patch for the document if the display is able to deal itself with these; otherwise the broker computes the new document version and forwards only the new document. Additionally, a patch description is sent for closing the menu.

5 Multiple Displays

The architecture of our new system inherits from the architecture of PG Kit (Fig. 1), which allows multiple displays to be connected to the broker. One use for this is to allow multiple views on a proof-in-progress, e.g., a display that shows a dependency

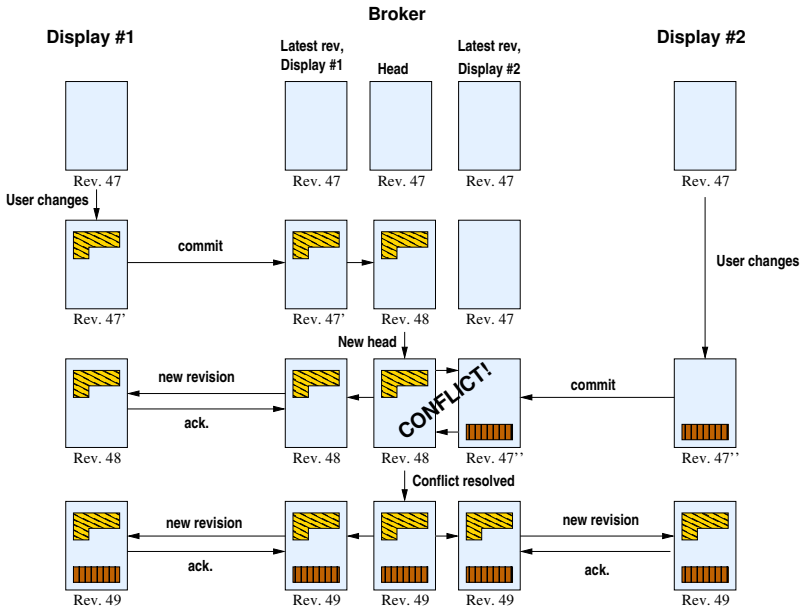


Fig. 5. Example for Editing via Multiple Displays

graph, or a graphical interpretation of a proof (perhaps rendering geometric arguments diagrammatically), alongside the main proof editing display. These displays are prover-specific, but fit smoothly into the general architecture.

Another use for multiple displays is to support more than one display to change the document. For this we need a way to synchronise input from different displays. A way to do this is for the broker to act as a simple kind of source control repository, illustrated by example in Fig. 5. This works as follows:

- The broker maintains the latest revision (the head) of a document, and for each display, a copy of the latest revision acknowledged by that display. In Fig. 5, the head is Rev. 47.
- When Display 1 sends a change (Rev. 47'), the change is committed to the new head (Rev. 48), and the new revision broadcast to all displays.
- Display 1 acknowledges the new revision. However, Display 2 has been changed meanwhile, so it does not acknowledge, instead attempting to commit its changes (Rev. 47"). The broker spots a potential conflict, and (in this case) merges the disjoint changes between 48 and 47" with respect to 47 into the current head revision without trouble. The merged document becomes the new head (Rev. 49), and is broadcast to all displays. Since no further changes have been made in the display, the both acknowledge.

If a conflict that cannot be merged occurs, the broker sends the merged document including conflict descriptions back to the display (using an extension to XUpdate to markup the conflicts, as in [9, Sect. 7.1.3]). The display (or the user) then needs to resolve the conflicts, and send in new changes.

This strategy is simple and flexible: displays could always send in changes to the whole document, and only acknowledge changes sent from the broker if the user has not edited the document at all. Alternatively, since this may create extensive conflicts without realising, displays might block between commit and acknowledge, or attempt to merge eagerly with new revisions sent by the broker.

6 Supporting Multiple Document Formats

So far the document format used with the display and the prover are essentially the same: for instance, in the classical PGIP with Isabelle, the document on the display is an Isabelle input file with additional markup. With the extension for arbitrary XML document formats in Section 3, we could connect a display and prover that both use OMDoc. But we cannot yet connect two different formats, say, connecting the display based on a document format D, with a prover that works on a different format P. This is the final missing piece of the architecture for emulating PlatΩ, which connects D =PL in the $\mathsf{PLATO_D}$ protocol to TEXMACS through to P =OMDoc format as used in the $\mathsf{PLATO_P}$ protocol to ΩMEGA.

To support multiple document formats at once, we propose to use a central structured document format B in the PG Kit broker that is annotated by PGIP markup. The broker does not need to know the semantics of the format B. Instead, dedicated translators are required for each target document format, translating $D \rightleftharpoons B$ and $B \rightleftharpoons P$. Each translator maintains a document representation mapping, and converts XUpdate-patches in either direction, much as the PlatΩ system does between the PL representation and the OMDoc representation as described in Sec. 2.1. The advantage of using the central format B is that provers do not need to be adapted to the document format of every display.

Experience with PlatΩ suggest the main difficulty lies in translating patch descriptions between the different document formats. Suppose we connect structured TEXMACS documents with plain text Isabelle proof scripts, and choose OMDoc as the broker's central document format. On the display side we have a translator component that mediates between TEXMACS documents and OMDoc. Prover side, a translator mediates between OMDoc and Isabelle ASCII text. We encode ASCII documents in XML as `<document><text>...</text>...<text>...</text></document>`, where text nodes are whitespace preserving.

Consider now the interactions when uploading and patching a document.Menu interactions are basically passed unchanged, but document patches must be translated. Since PlatΩ can already mediate between the TEXMACS and OMDoc formats, we need only one new translator for OMDoc and Isabelle, implementing:

XUpdate flattening going from OMDoc to ASCII, the structured XML representation must be transformed into a linearised text representation. A mapping must be setup between XML ranges and text ranges, i.e., the start XPath maps to the start text position (relative to the last range) and the end XPath maps to the end text position (relative to the last range). Start and end XPaths have the same parent XPath by definition. To flatten patches, the affected XML ranges must be recomputed and the mapping adapted; additions in the patch are flattened similarly.

XUpdate lifting: going from ASCII to OMDoc, the text spans must be lifted to the XML representation. Generally, this is done by mapping text spans to the corresponding sequence of adjacent XML ranges. As an invariant it must be checked whether the resulting sequence can be expressed by start and end XPaths with the same parent XPath. Similar to flattening, the mapping has to be adapted between text ranges and XML ranges.

Of course, the devil lies in the detail: OMDoc allows some embedding of legacy formats, but to usefully translate to and from Isabelle, we must accurately interpret a subset of syntax that reflects theory structure, and have some confidence about the correctness of the interpretation.

On the other side, we can now provide translators for further displays with advanced layout possibilities, such as Word 2007. The translator component must abstract the display document format to simplify it for the broker: e.g. in Word 2007, the document body is extracted and information about fonts, colors and spacing is stripped. On the way back, annotations are extracted from the patches coming from the broker, which guide heuristics for layout of new or modified text.

7 Related Work, Conclusion and Next Steps

Many user interfaces to theorem provers are similar to the Proof General style of line-by-line and single focus interaction using ASCII input files in native theorem prover format. Often, a custom interaction protocol is used. The main novelties for *PGIP 2* proposed here are: (i) to handle semi-structured XML documents as input formats; (ii) to allow the user to work on different parts of a document in parallel by using multiple foci; (iii) to allow the theorem prover to change parts of the input document, possibly using menus, and (iv) to have multiple views and editing of the same document in different displays.

With respect to (i) the *MathsTiles* system [5] also allows to map semi-structured documents towards several special reasoning systems. However, the mapping is only unidirectional from the display to the reasoners and also does not support multiple displays and conjunctive editing.

With respect to (ii) to our knowledge, the ΩMEGA system is the only prover that currently supports semi-structured document input and multiple foci. State information describing which parts of the document have been checked by ΩMEGA is managed in an ad hoc style; making this explicit in the multi-threaded state machine model in *PGIP 2* markup is a clear improvement. Further, it gives guidance on how to migrate a single-threaded theorem prover to a multi-threaded mode. There is some ongoing work for the Isabelle system to support multiple foci, but it is not in the official release to date.

Multiple views have been used in various forms in different systems, but not in a clearly distributed way that also allows editing, as in *PGIP 2*. In LΩUI [10] the display was split into a graph view on the proof and a display of the actual proof goals: those were based on pretty-printing and graph-visualisation tools built into the same display component. MATITA's user interface [1] has one proof script buffer and a display for the actual proof goal: the latter uses GtkMathview based on MathML representation of formulas that is generated from the internal representation of MATITA. GEOPROOF [8] allows one to generate COQ proofs from its internal, geometric representation which can be viewed in COQIDE [11]: this comes close to what we propose with multiple displays, except that currently there is no way back from COQ into GEOPROOF.[4] The infrastructure of *PGIP 2* and a (par-

[4] This could, of course, only be a partial mapping since not all COQ-proofs are geometric proofs.

tial) mapping from Coq into GeoProof would allow for simultaneously working in GeoProof and CoqIDE. Away from proof assistant systems, multiple views are familiar in IDEs for programming languages such as Eclipse and NetBeans: there the same file may be presented in different ways in different windows (e.g., code and model), and either updated dynamically in step, or at clearly defined points in the interaction (e.g., window activation).

The ability to extend the input document by incorporating information from the prover has also been supported in various ways before. An example besides the general change mechanism of PlatΩ/ΩMEGA is that of Matita, which can generate a *tinycal* proof script from the GUI interactions on goals, and include it into the overall document. We hope that a generic infrastructure would allow functionality like this to be reused between systems. The facility to include information from the prover together with the multiple foci provide a good basis to use PG Kit for provers like Mizar, PVS and Agda that have different, non-linear interaction styles. The details of adapting to further prover interaction styles is left to future work.

The main next step is to implement our planned *PGIP 2* and to rebuild PlatΩ's functionality on that basis. Future work will also be devoted to use Word 2007 and OpenOffice as displays and especially to build bi-directional transformers between prover-specific textual input files and corresponding OMDoc representations. We hope this will lead to a rich family of improved prover user interfaces.

References

[1] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007. Special Issue on User Interfaces in Theorem Proving.

[2] David Aspinall. Proof General: A generic tool for proof development. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785, pages 38–42. Springer, 2000.

[3] David Aspinall, Christoph Lüth, and Daniel Winterstein. A framework for interactive proof. In *Mathematical Knowledge Management MKM 2007*, volume 4573 of *LNAI*, pages 161– 175. Springer, 2007.

[4] Serge Autexier, Armin Fiedler, Thomas Neumann, and Marc Wagner. Supporting user-defined notations when integrating scientific text-editors with proof assistance. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, LNAI. Springer, june 2007.

[5] William Billingsley and Peter Robinson. Student Proof Exercises using MathsTiles and Isabelle/HOL in an Intelligent Book. *Journal of Automated Reasoning*, 39(2):181–218, August 2007.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Archite ctures.* PhD thesis, University of California, Irvine, 2000.

[7] Christoph Benzmüller Marc Wagner, Serge Autexier. Plato: A mediator between text-editors and proof assistance systems. In Christoph Benzmüller Serge Autexier, editor, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, volume 174(2) of *Electronic Notes on Theoretical Computer Science*, pages 87–107. Elsevier, april 2007.

[8] Julien Narboux. A graphical user interface for formal proofs in geometry. *Journal of Automated Reasoning*, 39(2):161–180, 2007. Special Issue on User Interfaces in Theorem Proving.

[9] Svetlana Radzevich. Semantic-based diff, patch and merge for XML documents. Master thesis, Saarland University, Saarbrücken, Germany, April 2006.

[10] J. Siekmann, S. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, and V. Sorge. $\mathcal{LOUI}$: $\mathcal{L}$ovely ΩMEGA $\mathcal{U}$ser $\mathcal{I}$nterface. *Formal Aspects of Computing*, 11:326–342, 1999.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.1.* INRIA, http://coq.inria.fr/doc-eng.html, 2008.

Replace this file with `prentcsmacro.sty` for your meeting, or with `entcsmacro.sty` for your meeting. Both can be found at the ENTCS Macro Home Page.

A Lightweight Theorem Prover Interface for Eclipse

Julien Charles [1]

Everest Group
INRIA Sophia Antipolis
2004 Route des Lucioles - BP 93
FR-06902 Sophia Antipolis, France

Joseph R. Kiniry [2]

Systems Research Group
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

Abstract

A major deliverable of the EU FP6 FET program MOBIUS project is the development of an Integrated Verification Environment (IVE)—the synthesis of a programming-centric Integrated Development Environment (IDE) with a proving-centric Interactive Theorem Prover (ITP). This IVE focuses on Java verification. Therefore, Eclipse was chosen as the IDE in which to integrate the system.

In this paper we present ProverEditor, a system used to interact with theorem provers from within Eclipse. It is similar to the Proof General Toolkit for Eclipse, except that it has a much more lightweight architecture, and consequently less features and more flexibility. In this paper we summarize its main functionality, as well as the plugin for the initial and primary prover that is well-supported, Coq. We also summarize the system's architecture and discuss our work on integrating other ITPs, PVS in particular.

Key words: interactive theorem prover, Eclipse, IDE, higher-order, interface, editor

[1] Email:julien.charles@sophia.inria.fr
[2] Email:joseph.kiniry@ucd.ie

©2008 Published by Elsevier Science B. V.

1 Introduction

Developing proofs using proof assistants can be a peculiarly difficult task. Even using current modern tools, formulating a moderately complex proof is sometime not easy. This difficulty is particularly noticeable in modern software verification efforts that regularly use theories with hundreds of, and sometimes (many) thousands of, definitions and theorems.

For instance, the Coq proof assistant has three main user interfaces [3] :

- CoqTop, which is just a LISP-like command-line top-level. While somewhat useful, it is a bit awkward to use and very limited in functionality.
- CoqIDE, which supports editing and evaluating a specific Coq proof script. While this interface offers some facilities for automatically constructing proof scripts and finding help, it is still quite difficult to manage the aforementioned complex theories and, e.g., navigate through hierarchies of proof files.
- Proof General, a major mode for Emacs, which has many features and is a very rich interface, leveraging the power and flexibility of Emacs. Ironically, its dependence on Emacs is one of its main drawbacks, as, in recent years, IDEs like Eclipse are replacing older tools like Emacs as the standard environments for developing software.

Ideally, one would just make Proof General a part of Eclipse. The Proof General Toolkit represents one such effort. Unfortunately, at this time Proof General Toolkit only supports the Isabelle theorem prover. Thus far, no other prover is supported by Proof General Toolkit within Eclipse, mainly because the inclusion of a new proof assistant is difficult due to the complexity of the Proof General Toolkit architecture and its communication protocol(s). (An attempt was made with Coq, but it seems to be discontinued.)

In this paper we present ProverEditor, a multi-prover interface for Eclipse. The architectural approach chosen here turns the Proof General Toolkit architecture on its head—our architecture is very *lightweight* and only a *simple interface* must be implemented to support the integration of a new proof assistant. Not only is proof script creation and editing supported, but interactions with the prover are enabled via a formally specified API that support communication with integrated and automatic provers via Java. ProverEditor also has some more advanced features like the Eclipse's outline view and a completion system.

The paper is divided as follows: first we provide a detailed overview of some existing tools and environments. Second, we review the design and architecture of our system. Third, we describe the ProverEditor plugins and features currently available, and summarize how they are similar to, and different from, the interfaces familiar to those that use theorem provers daily. Finally, we conclude with reflections on ongoing development work and next steps in integrating IDEs and ITPs.

[3] There is actually two others, PCoq [3], that is written in Java, and CtCoq, but they are not maintained anymore.

2 User Interfaces for Theorem Proving

There are several different canonical interfaces to interactive theorem provers. We first summarize these interfaces to (a) put ProverEditor in context, (b) compare and contract it with other interfaces, and (c) identify the interfaces and feature interactive provers have in common so as to drive our own architecture design.

2.1 Command-line Interfaces

The provers we are targeting (e.g., Coq, PVS, and Isabelle) have command-line interfaces as some top-level. These top-levels have many similarities. Each allows one to send commands to the prover and receive its answers using ASCII byte sequences and simple syntaxes. Usually the standard output file descriptor is used for the dialog and the standard error file descriptor is used to show the prompt.

A typical "raw" interaction at a top-level is to open a text file (e.g., in a text editor) where one stores all the command steps involved in a given interaction and cut-and-paste its content to the top-level and await results from the prover. This "user-active" kind of interaction is quite unnatural and ungainly. Therefore these low-level interactions are often wrapped in a richer environment like Emacs, or in the case of Coq, its own IDE, CoqIDE.

2.2 Web interface

ProofWeb [9] is a multi-prover web interface. It handles several provers notably Coq, Isabelle and Matita. It is mainly targeted towards students, hence it permits to interact with it without any local installation. It proposes several views to view proofs. It has also a system to access courses and predefined Coq files containing exercises.

Its architecture is similar in some ways to the one of ProverEditor as well as Proof General. The way it handles provers is plugin based, and the interaction is mostly handled plugin side.

This tool is really good for small sized projects and for teaching-oriented use of theorem provers, but it has several flaws, notably for handling large projects and for library forging. It only permits the edition of one file at a time, it does not handle directory, and it does not allow rich client type interaction. For instance, if a user wants to develop a proof of a program, he will not be able to edit the program and edit the proof obligation in the same environment. He will have to switch context back and forth from the code editor to the proof script editor. It will be worst if the user want to modify the program, and re-generate the proof obligation. The user will have to load the file by hand afterward through the interface. These kind of problems are partially solved using Emacs as the host for the multi-prover interface, and it is totally solved if using Eclipse [4].

[4] An exemple of rich client type interaction can be found in the tool MOBIUS' DirectVCGen on the MOBIUS Trac server [15]

2.3 Emacs

Emacs a relatively popular tool for computer scientists and programmers. It is now a bit overshadowed by more recent tools like Eclipse, and it is, in part, because of this "popularity with the masses," that we are currently targeting Eclipse as an integration platform.

The core features of modern IDEs that we retain for ProverEditor are the outline/summary view (which is not built-in to Emacs, though is available via the use of the Speedbar and/or CEDET tools), as well as the quite useful "tagging" (completion) system. Especially for development in C/C++ or Java, tagging is a very valuable feature. Eclipse replaces tagging by more semantically-aware form of definition searching and completion akin to Microsoft's "Intellisense" (Microsoft's implementation of autocompletion [5]).

2.4 Eclipse

The Eclipse platform is used like Emacs as a front-end for an Integrated Development Environment or, more generally, as a Rich Client Platform [13]. At first Eclipse focused on Java development. Now, it is used for C/C++ and Python development, as well as a front-end for revision control tools like CVS and Subversion and writing papers in LaTeX.

As Eclipse represents a "modern" development tool, several standard concepts are used: files are grouped in projects, it has an outline/summary view, and navigating source code is simplified via implicit definition retrieving. We believe that, to gain mind-share with today's programmers, it is good idea to be hosted in Eclipse, rather than in Emacs, both for these key features, as well as more social reasons.

2.5 Proof General

Proof General [4] is the *de facto* multi-prover environment. It uses Emacs as its graphical interface. It is multi-prover in the sense that it allows one to interact with Isabelle and Coq as well as other provers. It does not have a lightweight approach: each theorem prover-specific part uses its own communication engine, and each prover language has some of its semantic aspects encoded into this engine.

The new version of Proof General is called Proof General Toolkit [8], and it is an Eclipse feature/plugin. It bases its interactions with provers on the interaction it already supports with Isabelle. This interaction protocol is called PGIP, which uses an XML-based language called PGML.

By using XML, command streams are well-delimited and easy to parse in the Java context using a standard XML parser. This permits to add some of the prover language semantics inside the tags. A typical example of such an addition is the use of a vernacular-like Show command within a proof script—the XML around

[5] http://msdn2.microsoft.com/en-us/library/hcw1s69b(vs.71).aspx

this command is independent of the command and the proof script, which makes for a more script-independent protocol.

An interesting application of this PGML might be to support a new feature, like prover-centric refactoring, e.g., scope-aware variable renaming. A problem with realizing such a new feature is that the PGIP-based approach is, in some sense, too heavyweight to be generic. In order to communicate with the prover, the interface must maintain some semantics, in particular, it must have some theory and proof context information.

Since Proof General Eclipse uses the PGIP protocol to communicate with provers, supporting a new prover means realizing this protocol for that prover. Unfortunately, implementing this protocol seems to be not such an easy task for anyone other than the Proof General Toolkit developers, especially for system not written with these kind of interactions in mind. To have the full output and a good view of the state of Coq one must gather two informations: the ones collected from the standard output and the ones from the error output. In common cases these outputs do not interleave but in some specific situations the interleaving is unexpected. Hence it is difficult to gather to a single output and reordering informations for the PGIP protocol. In the same vein, PGIP does the hypothesis that the undo stack is infinite, which is not the case for Coq. One could argue to use instead the PCoq output of Coq which is supposed to offer all the facilities missing, but this output is not properly maintained, so using Coq standard interaction output is safer.

All these problems for adapting Coq to the PGIP protocol have led us to use a simpler protocol much similar to what has been done in Proof General for Emacs but in a lighter and more generic fashion.

3 Analysis and Design

ProverEditor is part of the MOBIUS Program Verification Environment (PVE). The MOBIUS PVE is an integrated environment that supports the specification, implementation, and verification of Java programs. Because the Mobius PVE focuses on Java, the Eclipse platform was a natural choice for a programming environment in which to host the Mobius PVE.

This integration has several aspects:

- Eclipse has a well-defined plugin architecture, so extending the system with rich functionality and interfaces is relatively straightforward, and
- Mobius PVE subsystems can extend and interact easily with Eclipse's Java programming components.

In this section, we present the integration of ProverEditor inside of Eclipse and then summarize its main features.

3.1 Plugin Architecture

ProverEditor's architecture is based on the Eclipse plugin architecture. Fundamentally, ProverEditor is an Eclipse plugin that hosts several other plugins which are specific to the provers with which it interacts.

3.1.1 Plugins in Eclipse

Eclipse's plugin architecture's power and flexibility principally lies in its *extension points* concept. An *extension point* is a facet of a component's interface, thus a plugin can either provide an extension point or extend an extension point of another plugin. Extension points are often used to implement a specific behaviour or, as is often the case in Eclipse, to support the proper integration of a plugin into Eclipse's graphical interface. For instance, if one wants a plugin to be integrated to the preferences menu, an extension point must be used to add a the plugin's preferences page to Eclipse's preferences menu.

This compositional modularity is very useful in combining several facilities provided by other plugins, like synthesizing a generic interface to two similar systems, like the CVS and Subversion revision control systems. In our case, it simplifies the adoption and adaptation of modern IDE features in our interface, as initially advocated by Kiniry [11]. While such adoption and adaption is the hallmark of extensions to Emacs, due to the richer foundations (and consequent greater use of resources) of Eclipse and Java, Eclipse plugins can be more powerful and attractive than their Emacs-based counterparts.

Extension points are defined by a unique identifier that links to a definition of an XML tag. Programmers that wish to extend this extension point must write an XML file called `plugin.xml` (typically with the assistance of a specialized plugin editor that comes with Eclipse) as well as provide the correct parameters to the tag in question. The plugin providing the extension point inspects, at runtime, what was specified with the XML tag. The name of a class to classload is typically provided, and consequently instantiated, otherwise the name of a resource or a simple `String` may be provided.

Eclipse provides many extension points: e.g., editors for specific file types, binding a file type to a specific icon, and syntax highlighting. One can also add specialized widgets to the interface, like the standard "outline" view that so many IDEs provide. A drawback of using extension points is fundamental to their design: they are, in essence, static and global. In addition, as stated above, extension points can only be provided through the addition of a new plugin with a plugin specification file. Thus, in general, Eclipse does not permit any mechanism for (dynamic) extension.

3.1.2 Plugins Used

ProverEditor is based on different plugins which define different subsystems: the editor, the outline, the proof view and the preference page.

The editor is based on Eclipse's integrated editor plugin. The standard Eclipse

file editor is extended to view and edit proof scripts by (a) adding prover-specific syntax highlighting, and (b) adding a means by which subsets of the text can be dynamically highlighted so as to show which parts of a proof script have already been evaluated and may not be modified.

The outline shows the detailed structure of a file being edited via a tree-based view. The basic outline view of ProverEditor is initially empty: it only contains the name of a file opened in an editor and which has the focus. The prover plugins augment this implementation by providing an outline of the proof script files. For example, in Coq, the outline view summarizes the type hierarchy found in the current file. The outline always represents the whole file, much like what is done for the Java outline in Eclipse. This view is especially useful when inspecting a library file, to jump easily to a specific definition.

Another plugin is the *proof view*, which shows the user the result of each interaction with the prover. It is essentially a log of the user interactions with the plugin and needs no input. It is integrated at the same fashion as the outline window in the Eclipse workspace. This interaction can only target a single file at a time. A file has the interaction focus only if the user decides to step through it.

The last extension from Eclipse that is used is the *PreferencePage* extension. It handles the preferences necessary for a particular *proof view*, like whether to expect output to use the Unicode character set.

Currently there are four plugins. The base plugin (ProverEditor) handles all generic (non-prover-specific) interaction. There are plugins to support the Coq and PVS higher-order provers. There is also a plugin that supports interaction with the top-level. It provides a high-level Java API to control a Coq top-level, and is called the CoqSugar plugin.

3.2 ProverEditor

ProverEditor is formed of four parts, as seen in Figure 1: the editor, the top-level view, the outline view, and the toolbar. ProverEditor also understands a number of keyboard shortcuts to trigger toolbar and menubar actions.

The actions associated with the buttons seen in Figure 2 are (from left to right):

- start to evaluate the file from the beginning
- take a step in the current file
- undo a step
- progress to the end of the file
- undo to the beginning of the file
- cancel an action

These actions are fairly standard in proof assistants like Coq, PVS, and Isabelle. Of course, they are also similar to the actions implemented in Proof General and CoqIDE. In fact, the general organization of the view for ProverEditor was inspired by the look-and-feel of those two tools. Still, there are some enhance-

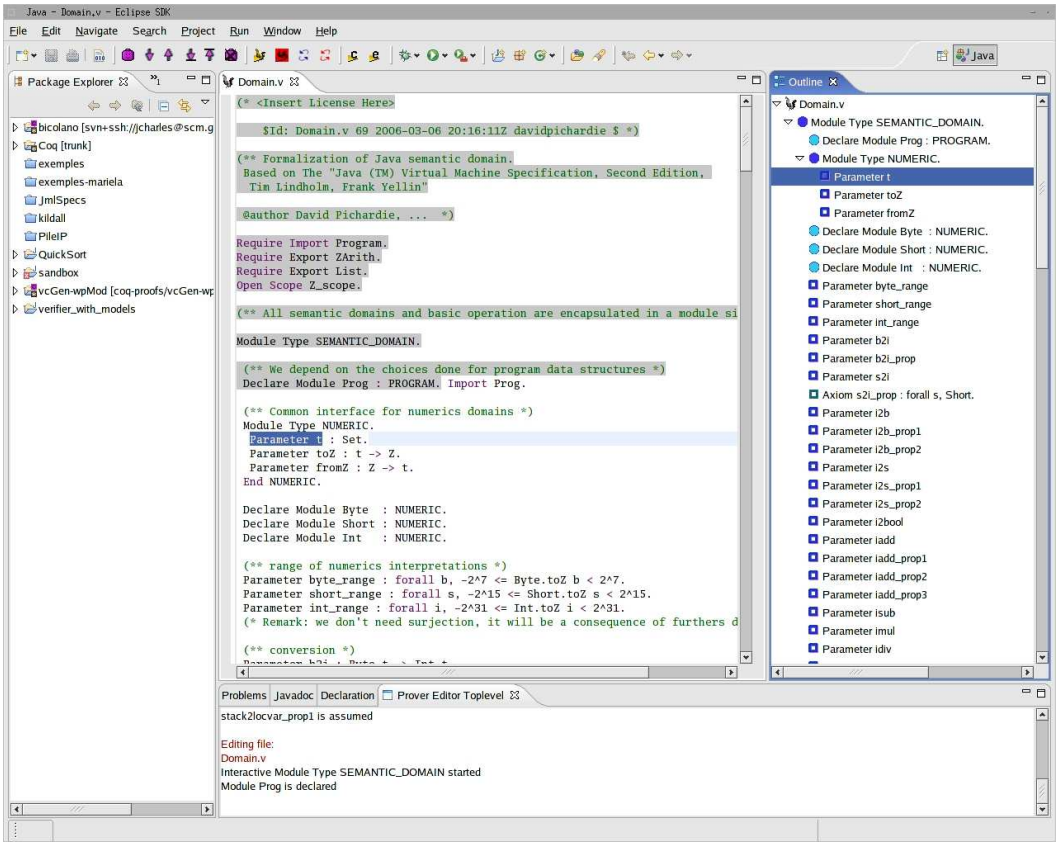


Fig. 1. A Typical ProverEditor Interface

Fig. 2. The ProverEditor Toolbar

ments, as ProverEditor adds syntax highlighting to the top-level output and has a very lightweight interface that reflects its lightweight architecture.

More concretely, extending ProverEditor for new provers is simpler than extending Proof General. In order to provide the base extensions, syntax highlighting, and prover input/output communication for a new prover, less than a hundred lines of Java code is necessary. The base implementation provides library functions to communicate with provers as well, so adding support for further new provers should necessitate even less code.

3.2.1 Core features

First we will discuss the main features provided by the base ProverEditor plugin. This plugin provides some low-level handling of a target prover top-level as well as the basic UI building blocks, to edit a file from the prover, highlight its syntax, show an outline of the file, and step through a proof.

The interaction subsystem, found in the `prover.exec` package, manages in-

teractions like sending a stream to, and receiving data from, the top-level via pipes, typically its standard output and standard error. The main class in this subsystem is the `prover.exec.toplevel.TopLevel` class, which implements the interface `prover.exec.ITopLevel`.

Interaction with the top-level is based on calls to a method called `sendCommand`. In a typical `sendCommand` scenario, the method `sendCommand(String)` of the `TopLevel` class is called. The command passed is supposed to be atomic. The primitive `ITopLevel.sendToProver(String)` is called afterward with a prover specific code.

`sendCommand` is a gateway for other methods to undo commands. For example, the `undo()` method triggers an undo command to the top-level. This method calls a prover-specific command that undos one step from whichever context in which the user is working. In most of the cases, this command reduces to a call to `sendCommand()` with the correct parameter. For instance, in the case of Coq, the undo command is translated into sending one of three Coq top-level commands: `Back` command, the `Undo` command, or the `Abort` command and for this behaviour the top-level state is used.

These interactions are generic and simple, but they permit accurate communication with each prover. They rely on a simple parsing of the inspected file. These base feature can be used in more advanced interactions like the tagging system.

3.2.2 Tagging

A feature that differentiates ProverEditor from other similar systems is its support for tags and tagging. Tags are a standard way of indexing source code contained in libraries for interactive front-ends. One of the main implementations of tagging is found in the program `ctags`. Tagging is used in the vi editor as well as in Emacs. Here we chose to implement a tagging system compatible with the etags [2], which is used with Emacs, as nearly all higher-order theorem provers “native” interfaces are based upon Emacs.

To tag a file, one uses regular expressions to match identifiers with their definitions. Once identifiers and their definitions are all gathered, a system can search through this index at user request. The basic search method is triggered when the user select an identifier (with, say, the mouse pointer), and asks to open its definition, or something that is considered as its definition.

In ProverEditor, definition search is triggered by the standard Eclipse keystroke ‘`F3`’. This keystroke only works if the current Eclipse project is a ProverEditor project, like a Coq project for instance. This restriction is due to the fact that the tags are built incrementally, and in Eclipse this incremental construction feature is intimately linked to the project nature.

Tags are stored using the standard `etags` format [1]. We find it useful to have a compatible storage format because sometimes one works with Proof General Emacs and ProverEditor at the same time. While this is not the most common situation, it is a valuable feature to those who must use both interfaces, either simultaneously or alternatively.

The tagging system used in our tool is quite efficient for several reasons:

- Eclipse optimizes file search, so definition search is efficient and fast,
- definition search is incremental—each time a file is added, removed, or saved to the disk, its tags are calculated and saved, and
- user interaction identical to Eclipse's Java code browsing functionality. When the user presses 'F3' over an identifier, the system opens a file containing the corresponding definition and highlights it. This interaction is very familiar to even the beginner Eclipse developer. Since our approach is lightweight and involves no context information, several definitions may be found for a single identifier. Thus, pressing 'F3' again jumps to the next definition.

To our knowledge, no other tagging system has been implemented for Eclipse. A potential future development is spawning off an independent tagging plugin from this code. Such a plugin would be useful as a separate component for Proof General Toolkit Eclipse, as well as ProverEditor or any other Eclipse plugin.

3.2.3 Extending ProverEditor with New Provers

The key differentiating feature of ProverEditor is the lightweight way in which one integrates new provers. While the programmatic interface is simple, new provers must have certain key properties in order to seamlessly integrate them. In particular, the prover must have a top-level and they must have two modes of interaction: a "definition mode" and a "proof mode". This separation is useful for minimal interaction as presented in this paper, and especially for Coq. To have a richer interaction the way would be to create a plugin that could be extended implementing a specific protocol. This could be useful for better integration of some provers, like PVS.

PVS integration hasn't been an easy task, because interaction with the prover is more complex than in Coq. The main problem is that the two modes tends to interleave in different way than seen Coq or Isabelle. Still, due to ProverEditor's lightweight nature, the current prototype supporting PVS is only around a hundred lines of code as well [6].

The main aim is to keep extensions simple and easy. To add a new prover one must write an Eclipse plugin in the standard way and extend two extension points. Additionally, one must implement at least two classes: one to help handle top-level interactions, and the other, which is more prover-oriented, implements GUI related functionality.

The first extension point to extend is called `org.eclipse.ui.editors`. It is the extension point used to add a new file format handling. The editor to edit the file is provided by ProverEditor is the class `prover.gui.editor.ProverEditor`. This extension point has to be extended, because Eclipse permits to add specific editors only statically, through this extension point. We expect in the future

[6] This plugin is currently experimental, but its source code can be downloaded from the ProverEditor main site [14], as well as the MOBIUS Trac server [15]

developments of Eclipse we will be able to automatically add it like what is done currently for the *PreferencePage*.

The second extension point that must be realized is specific to ProverEditor. Its name is `prover.editor.prover`. This extension point connects the two aforementioned classes that must be implemented to support a prover to the generic plugin. Two classes have to be added, one that extends the abstract class `prover.plugins.AProverTranslator` and the other implementing the interface `prover.plugins.IProverTopLevel`. The latter providing some of the top-level functionalities.

By these two simple steps one can add a prover plugin to ProverEditor. Currently two plugins are using directly these functionalities, the CoqPlugin and the PvsPlugin.

4 Current Plugins

The main motivation to make the ProverEditor was to manage Coq interaction inside of Eclipse. That is why the finished plugins concern Coq. There is also a PVS plugin which is in developpement. ProverEditor plugins are used in 2 tools: JACK [5] (the Java Applet Correctness Kit), and MOBIUS' DirectVCGen [7]. Both tools being Eclipse plugins to do static program verification on Java programs annotated with JML.

Right now the plugins available are the following:

- the core plugin, containing all the basic features which has been described in Subsection 3.2.
- the Coq plugin: it handle the interactions with Coq, basically it send parts of a file to Coq, give an outline of the current edited file and do tagging. This plugin is used with the DirectVCGen and JACK.
- the Coq Sugar plugin, which is an API to interact with the Coq top-level. It is the plugin used in JACK

4.1 The Coq Plugin

This plugin is the genuine plugin for ProverEditor. It contains all the features that were mentionned previously:

This plugin is made to do interaction with Coq. The core features are inside 3 classes. The mandatory plugin class for Eclipse (Eclipse needs a plugin class in order to consider it has a plugin). This class is generated automatically by Eclipse. The two other classes are the one needed to add the extension to ProverEditor: the one used to handle the top-level and the other one containing mainly the highlighting parts.

There is another part which is less mandatory: the one to handle the outline. This adds about 10 classes to represents the leafs and nodes of the tree-view. There are 2 kinds of handling: the leaf kind, containing only constructs like

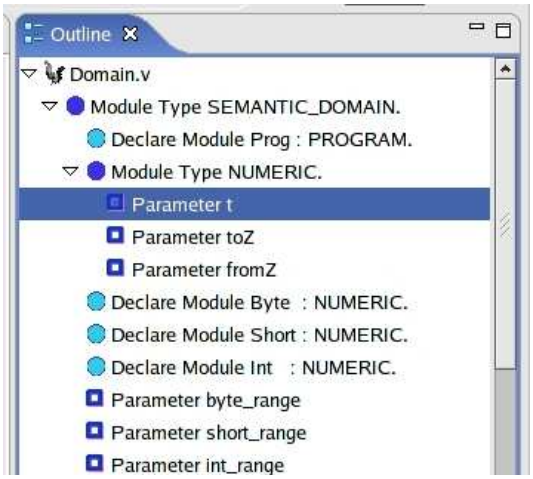


Fig. 3. The outline for Coq

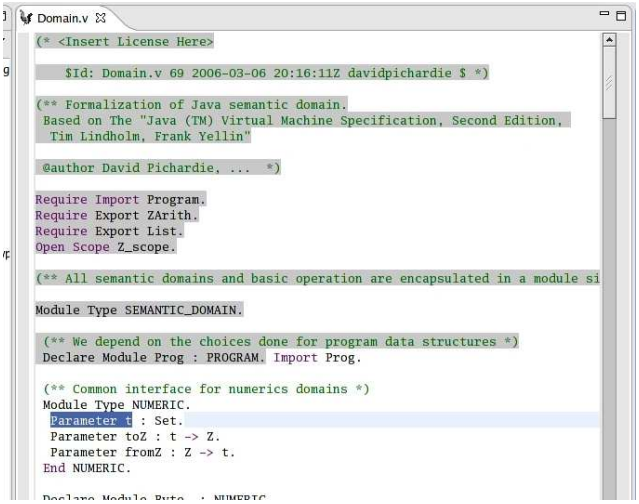


Fig. 4. The editor customized for Coq

`Definition`, `Parameter` and other single non-imbricated definitions. There is another kind of construct, the imbricated ones. In Coq these are the section and the module. These constructs are of a binary kind, with one to begin them (`Module m.` or `Section s.`) and one to end them (`End m.` or `End s.`).

4.2 The Coq Sugar plugin

We have made another plugin which add lots of some superfuous features to the basic plugin. The main purpose of this plugin being to ease the use of Coq within ProverEditor.

We have mainly added a more complete API to handle Coq. It has more high-level 'macros'. It subclasses the prover.exec.toplevel.TopLevel to have a real top-level targeted at Coq. It adds some facilities like methods to declare lemmas, do a

particular standard command or parse the output from Coq to give a parsed result of the command instead of the standard output. For instance, in Coq there is a command `Show Intros` which is used to know which variable name Coq would use with its intros command. Here the method gives an array of String with the different variable names. In Jack we mainly use this API to pretty print the proof obligations with Coq in order to have more user-readable proof obligations.

5 Conclusion

We have implemented a lightweight theorem prover within Eclipse, based on the extensions facilities provided by Eclipse. To have done it lightweight and minimal in its mandatory features has allowed us to extend it quite fast for the theorem provers like Coq or PVS.

What will follow is the extension of these basic features toward more prover specific features. One of the main idea would be to extend it in a component based approach. Instead of having as in the Proof General Toolkit one single mediator communicating through lighweight protocols to other plugins, we have a main plugin, which handle base interaction, that can delegate specific protocol handling to plugins and their extensions.

5.1 Next Steps

The next step will be to have real API plugins for provers. It has already begun through Coq's Sugar plugin which only aim is to provide this kind of API. It will be done later on for PVS.

We plan to include Isabelle/HOL in a near future. The inclusion should be simple as Isabelle can generate simple tagged output.

Other features that should be added as separate plugins are the projects and file wizard to creat new prover specific files or projects dedicated to a single prover.

Tagging has been included in for Coq. This approach should be completed with hints in a similar way as for Java. Like for tagging one of the difficulty is to keep it generic and simple enough. One of the pattern that could be used is to associate each identified tag with the nearest identified documentation. This is the pattern used in Java parser to keep the Javadoc in the bytecode [7] . Now we use the outline to get an idea of the file structures for Coq and PVS. We plan to do an enhanced outline that could give a representation of the proof tree. This extended outline could permit to manipulate the definitions as objects, which would be more akin of the Proof by Pointing [6].

We plan also to integrate it more thoroughly in the Mobius PVE [10]. Especially the look and feel which shall become more uniform with the other plugins part of the Mobius PVE.

[7] Although no real documentation is available, it can be seen in the sourcecode of the OpenJDK[12]

Acknowledgement

Benjamin Grégoire, Gilles Barthe and Yves Bertot. This article has been partially funded by The European Project MOBIUS within the frame of IST 6th Framework.

References

[1] *ETags*, http://www.gnu.org/software/emacs/emacs-lisp-intro/html_node/emacs.html#Tag-Syntax/.

[2] *Exuberant CTags*, http://ctags.sourceforge.net/.

[3] Amerkad, A., Y. Bertot, L. Rideau and L. Pottier, *Mathematics and proof presentation in Pcoq*, in: *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, Sienna, Italy, 2001.

[4] Aspinall, D., *Proof general: A generic tool for proof development*, in: *Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000*, number 1785 in LNCS, 200.

[5] Barthe, G., L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova and A. Requet, *JACK: a tool for validation of security and behaviour of Java applications*, in: *Formal Methods for Components and Objects*, Lecture Notes in Computer Science **4709** (2007), pp. 152–174.

[6] Bertot, Y., G. Kahn and L. Théry, *Proof by Pointing*, in: M. Hagiya and J. C. Mitchell, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Softward* (1994), pp. 141–160, http://citeseer.ist.psu.edu/bertot94proof.html.

[7] MOBIUS Consortium, *Deliverable 4.3: Intermediate report on proof-transforming compiler* (2007), available online from http://mobius.inria.fr.

[8] David Aspinall, C. L. and D. Winterstein, *A Framework for Interactive Proof*, in: *Towards Mechanized Mathematical Assistants* (2007), pp. 161–175.

[9] Kaliszyk, C., F. Wiedijk, M. Hendriks and F. van Raamsdonk, *Teaching logic using a state-of-the-art proof assistant*, in: H. Geuvers and P. Courtieu, editors, *International Workshop on Proof Assistants and Types in Education (PATE'07)*, 2007.

[10] Kiniry, J., *Formalizing the user's context to support user interfaces for integrated mathematical environments*, Electronic Notes in Theoretical Computer Science **103** (2004).

[11] Kiniry, J. and S. Owre, *Improving the PVS user interface*, in: *User Interfaces for Theorem Proving*, 2003.

[12] Sun Microsystems Inc., *OpenJDK* (2007-2008), http://openjdk.java.net/.

[13] The Eclipse Consortium, *Rich Client Platform*, http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[14] The MOBIUS Project, *ProverEditor* (2007-2008), http://provereditor.gforge.inria.fr.

[15] The MOBIUS Project, *The Mobius Trac* (2007-2008), http://mobius.ucd.ie.

Visualizing Proof Search for Theorem Prover Development [1]

John Byrnes[2]

HNC Software
Fair Isaac Research
San Diego, California
johnbyrnes@fairisaac.com

Michael Buchanan Michael Ernst Philip Miller Chris Roberts Robert Keller[3]

Department of Computer Science
Harvey Mudd College
Claremont, California
{mbuchanan,mernst,pmiller,croberts,keller}@cs.hmc.edu

Abstract

We describe an interactive visualization tool for large natural deduction proof searches. The tool permits the display of a search as it progresses, as well as the proof tree itself. We discuss the feature set and architecture of the tool, including aspects of extensibility and the interface for interaction with other user-provided analysis and visualization code.

Keywords: Natural deduction, automated theorem prover, intercalation search, proof search visualization

1 Introduction

There are two main reasons why automated theorem proving in natural deduction particularly benefits from visualization. First, natural deduction is generally considered to be easier to read than most other logics (4; 15; 19); one motivation for doing theorem proving in natural deduction (ND) is to produce easily-comprehensible proofs, but in order to be understood they must first be put in an accessible form.

[1] The authors are very grateful to our referees for supplying insightful questions, comments, and references which greatly improved the quality of this paper.

[2] This work was funded in part by the Intelligence Advanced Research Projects Activity (IARPA) Collaboration and Analyst System Effectiveness (CASE) Program, contract FA8750-06-C-0194 issued by Air Force Research Laboratory (AFRL). HNC Software is a subcontractor of the University of South Carolina in this contract. The views and conclusions are those of the authors, not of the US Government or its agencies.

[3] This project was conducted under the auspices of the Harvey Mudd College Clinic Program, with the sponsorship of Fair Isaac Corporation.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Second, automated theorem proving (ATP) in natural deduction is still in early stages of study, and visualization provides a much-needed tool to assist in understanding the operation of experimental algorithms. By proving and visualizing in natural deduction, a single tool can display both human-readable proofs and be faithful enough to the data structures used by the reasoner to allow easy development of theorem proving algorithms.

Many ideas have been developed to aid in the comprehension of automatically generated proofs. Some we adopt for our purposes, such as graphical display (23). Others are unnecessary, such as converting proofs to natural deduction (15). Still others may aid proof comprehension but would do so at the cost of obscuring the function of the underlying theorem prover, such as conversion to natural language (6), and so we eschew them. We did not expect HTML-based browsers such as IWBrowser (of the Inference Web project, (13)) and SigmaKEE (of the SUMO project (17)) to facilitate high-level inspection of large proofs well. Interactive DAG viewers, such as the Interactive Derivation Viewer (25), are more likely to succeed at such a task.

In this paper we describe **ViPrS** (an acronym for **Visualizing Proof Search**), a tool for visualizing and interacting with proofs and partial proof search structures. It was designed with particular attention to use as an aid for proof search algorithm development and has several features to facilitate that use, including an extremely flexible programmatic user interface, the ability to interface directly with a theorem prover, and the ability to interact with the theorem prover as the search evolves step by step.

ViPrS was built specifically for the SILK theorem proving project, which we describe in section 2. The design is intended to be decoupled from SILK as much as possible, but effort was not spent in this initial version on allowing ViPrS to interact with arbitrary reasoning engines. SILK reads in proofs specified in a fairly simple XML format (described in (26)). Section 3 describes ViPrS, including its interface, implementation, and architecture. In section 4 we discuss experience with ViPrS to date, and possible future work is discussed in section 5.

2 The SILK Reasoning Project

Automated theorem provers have traditionally been directed toward problem solving in the domain of mathematics (24; 22). Although they have been successfully adapted to other formal domains, such as circuit verification, adaptation to informal domains of human knowledge has proven much more challenging. Large ontologies (18; 17) have been constructed to formalize reasoning in many domains, and the semantic web (2) offers the promise of ever growing amounts of formal knowledge over which software will attempt to reason. Traditional approaches to automated reasoning suffer from the combinatorial explosion of the search space that follows from the enormous number of concepts and axioms in large knowledgebases.

A number of extensions to traditional reasoning have been suggested. Proof planning (16) and strict segmentation of knowledge into microtheories (18) have shown promise. The approach of SILK, or "Soft Inference for Large Knowledgebases", is to adapt machine-learning techniques designed for unsupervised organiza-

tion of unstructured data (primarily text) to the problem of structured knowledge. Patterns in the co-occurrence statistics of formal systems are exploited to automatically *compress* knowledge into a smaller vocabulary of higher-level concepts. Reasoning can occur much more efficiently at the higher level, and the resulting proofs can be used to guide proof search in the original vocabulary. In this regard, the high-level proof can be seen as a *plan*, and the approach is somewhat like proof planning with the exception that plans are discovered automatically in any domain rather than being coded from expert knowledge.

SILK also addresses the problem of reasoning under inconsistent assumptions. Large knowledgebases, especially those that grow organically such as the semantic web, are certain to occasionally introduce contradiction. Techniques which extract information from various data sources will introduce contradictions both due to extraction errors and due to the existence of inconsistent claims or erroneous entries in data sources. Because knowledge compression can also introduce contradiction, SILK needs to be especially robust. Classical theorem provers trivially reach arbitrary conclusions from contradictory assumptions, but SILK has the ability to prove relevant results without making arbitrary conclusions from inconsistency. This is achieved by using minimal logic via natural deduction proof search (this is an additional benefit to ND search over classical resolution search beyond the improved readability described in section 1). The reduced set of attainable conclusions is expected to be sufficient for many expected applications of "real world reasoning" (as opposed to theoretical mathematical reasoning), but this result needs to be established through usage. Of course SILK also has the option merely to prefer minimal proofs, permitting classical inferences sparingly and perhaps notifying the user when doing so.

The technique of *intercalation* (21) has been adapted to create direct natural deduction search which is provably as efficient as search in the sequent calculus (4). SILK reasons directly in IKL (9), a dialect of Common Logic (CL) (5), which provides a very convenient and powerful syntax which looks higher-order but has a strictly first-order semantics over which it is sound and complete (8) [4]. KIF (the Knowledge Interchange Format (7)) is the most well known variant of CL. The typical approach to reasoning over knowledge represented in KIF by a first-order theorem prover is to use the "holds" translation into first-order logic, leading to computational and complexity difficulties (11). SILK attempts to overcome these difficulties through use of a natural deduction calculus in which reasoning is done directly in IKL without translation.

The intercalation theory underlying natural deduction search has been proven sound but has not received the large-scale implementation attention that has been given to resolution and other standard automated theorem proving techniques (12; 1). Reasoning directly in CL is novel, and the practical complexities are yet to be discovered. Reasoning in large knowledgebases of course yields very large search spaces and often large proofs as well, as inherited properties of classes must be established by reasoning through the subclass hierarchy. For all of these reasons, development of a practical, usable system such as SILK is likely to encounter many

[4] Actually, this is true for the fragment of IKL implemented in SILK, which only allows finite expansion of row variables

unforeseen obstacles which may be difficult to understand and overcome without tools such as ViPrS, which provide insights into the search patterns and inefficiencies that arise in practical application. SILK is in very early stages of development. It has not yet been reported on and is not available for public access.

3 ViPrS

SILK's data structure is particularly well-suited to graphical visualization. As described below, the structure is inherently non-linear, which makes displaying it textually unproductive. Instead, we visualize the search DAGs graphically. The objects to be displayed are also large enough that the interface must allow both inspection of the overall structure and of finer details, a challenge we address in several ways.

The search space for a proof is represented by a directed acyclic graph (DAG) rooted at the proposition which SILK is attempting to prove. *Proof search graphs* contain two distinct types of nodes. *Line nodes* represent logical formulas, while *rule nodes* represent rule applications. A rule node will have one line node conclusion and any number of line node premises. It is considered *proved* if all of its premises are proved. Likewise, a line node will have one or more *applications*, rule nodes for which it is a premise (except the root, which has none), and zero or more *justifications*, rule nodes for which it is the conclusion. A line node is considered *proved* if any of its justifications is proved or if it is an axiom or assumption.

A complete proof, then, consists of the proposition to be proved at the root, various internal nodes, and axioms and assumptions as leaves. A proof search graph is similar, except that its leaves may include formulas whose truth is as-yet undetermined. As such, proofs are merely a special class of proof search graphs, namely, completed ones. For conciseness we will henceforth refer to the object of visualization as a *proof search*.

3.1 Interface

A typical screenshot of ViPrS in use can be seen in figure 1. Here, we describe its salient features.

3.1.1 Viewing Pane

The main part of the ViPrS interface is the viewing pane. Here, the proof search is displayed in a traditional graphical format, with nodes represented as boxes and their relationships as lines between them. By default rule nodes have text indicating their type (which rule is being applied), while line nodes are blank (due to the lengthy propositions of most interesting proofs). The contents of a line node are viewable in a tooltip, or can be displayed using the command-line interface, as described in section 3.1.2.

The viewing pane allows various types of mouse-based interaction. Scrolling the mouse wheel zooms in and out of the proof, allowing inspection of the fine detail of proofs that are too large to easily fit on screen. Clicking and dragging pans the display. Clicking on a node selects it, causing it to visually increase in size and

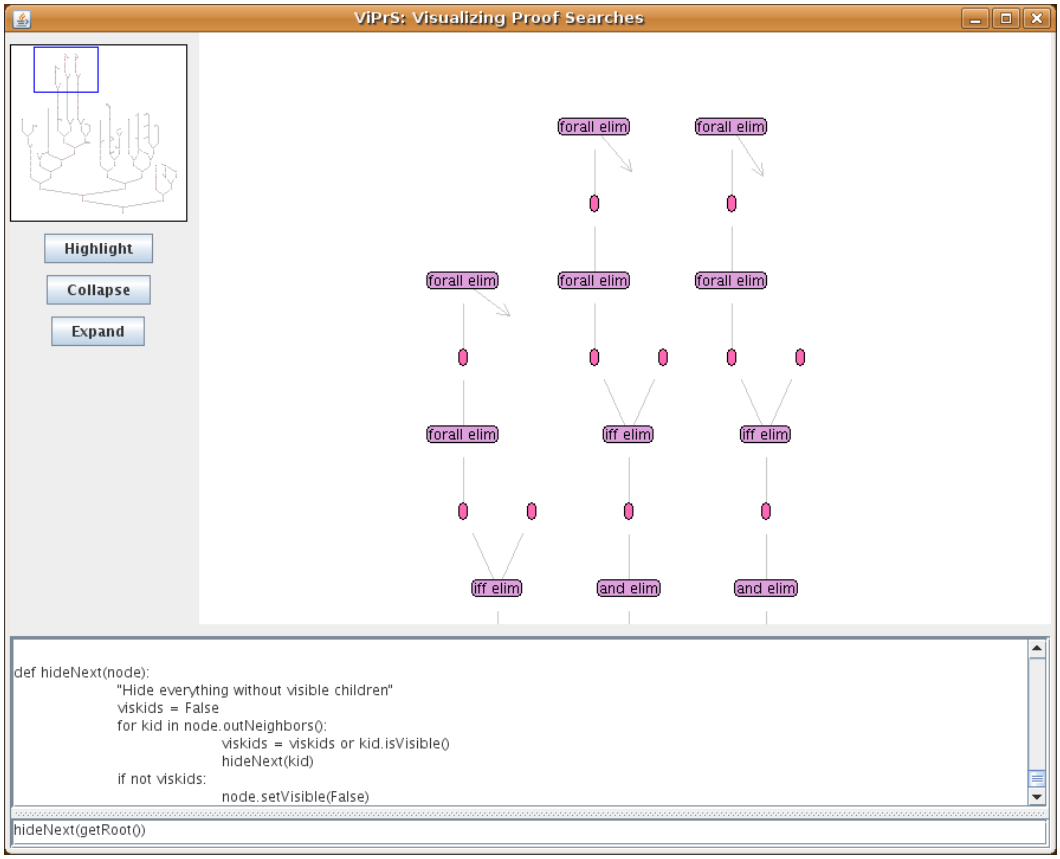


Fig. 1. A screenshot of the complete ViPrS system, with the main viewing pane in the center, the minimap and customizable buttons along the left, and the command-line interface at the bottom.

making it the object of future button presses. Hovering over a line node displays a tool-tip that contains its formula and the formulas that make up its context.

DAG layouts frequently attempt to minimize edge lengths and crossings by placing linked nodes near each other, but occasional long crossing edges are unavoidable. ViPrS simplifies the layout by treating the DAG in a very tree-like manner. The *primary parent* of each node is the parent closest to the root (note that the goal of the search provides a unique root to the search space), or leftmost in case of a tie. All other parents of a node connect to it via *crossing edges*, which are the short pairs of arrows displayed in figure 3. Clicking on one end of a crossing edge automatically pans the display to the far end of the edge and darkens both ends of the edge. The panning is most important when the far end of the edge is offscreen; the highlighting is especially useful when multiple crossing edge endpoints are visible.

3.1.2 Command Line Interface

Sited below the main viewing pane is the Command Line Interface (CLI), the primary means for the user to manipulate the proof search. The CLI lets the user interact programmatically with the visualization and the reasoning engine. Specifically, the user can enter arbitrary Python code and have it interpreted. Through predefined library functions and specially exposed variables, this code can interact with the visualization. This means that the user can query and manipulate every

property of the proof search without the need for any explicit prior implementation of the particular interaction. The exposed variables link not only to the visualization layer but into the data structures of the search algorithm as well. This means that any detail of the state of the algorithm can be inspected or adjusted at any step of search, allowing the user to understand and alter decisions made about backtracking, goal selection, etc. As the system developer changes the Java code in the reasoning engine, the new structures become exposed in ViPrS without any changes required to ViPrS so long as the original data structures can access the new structure.

The python library functions are loaded from a file at start up. The user can add arbitrary new function definitions to this file, expanding the library simply in python without editing any java code and without recompiling. The CLI also provides a *command history*. The command history makes it easy for the user to re-run previous commands, with or without modification. Our implementation lets the user both scroll through the history sequentially and search through it. The history persists between runs in a simple text file, allowing a user to return to commands from previous sessions. Function definitions entered during a session can be re-executed through the history mechanism or can be manually copied from the history file into the library file.

3.1.3 Dynamic Buttons

To the left of the viewing pane are a number of buttons to provide easy mouse-based manipulation of the proof search. As currently implemented, buttons act on the selected node(s). In the example configuration seen in figure 1, ViPrS has three buttons: one highlights a node by changing its color, and the others hide and display the sub-DAG rooted at the selected node.

The exciting aspect of the buttons is their easy customizability. Rather than running compiled-in code, each button is associated with a piece of Python code to run when it is clicked. A new button can be added by invoking a simple function, `addButton`, through the CLI. The buttons appearing on start up are defined similarly in the user's initialization script. Buttons can also be removed through an analogous `removeButton` function.

This sort of easy tool-building should be appreciated by and comfortable for our target audience of ATP developers and advanced users. In addition to saving us from having to anticipate the most frequently useful commands, dynamic buttons also hold distinctive advantages for users over buttons with fixed functionality. The user can create appropriate buttons to avoid switching back and forth between navigating a proof search with the mouse and manipulating it through the CLI. Making the buttons completely dynamic also saves users from having to recompile and restart, or even reload some configuration file, to extend the functionality of ViPrS.

3.1.4 Minimap

Since proof searches can be so large, and the user may be focusing on only a small area at a time, we provide a summary view, or "minimap" of the proof search in a small box to the side. The minimap is a small, less-detailed view of the entire proof

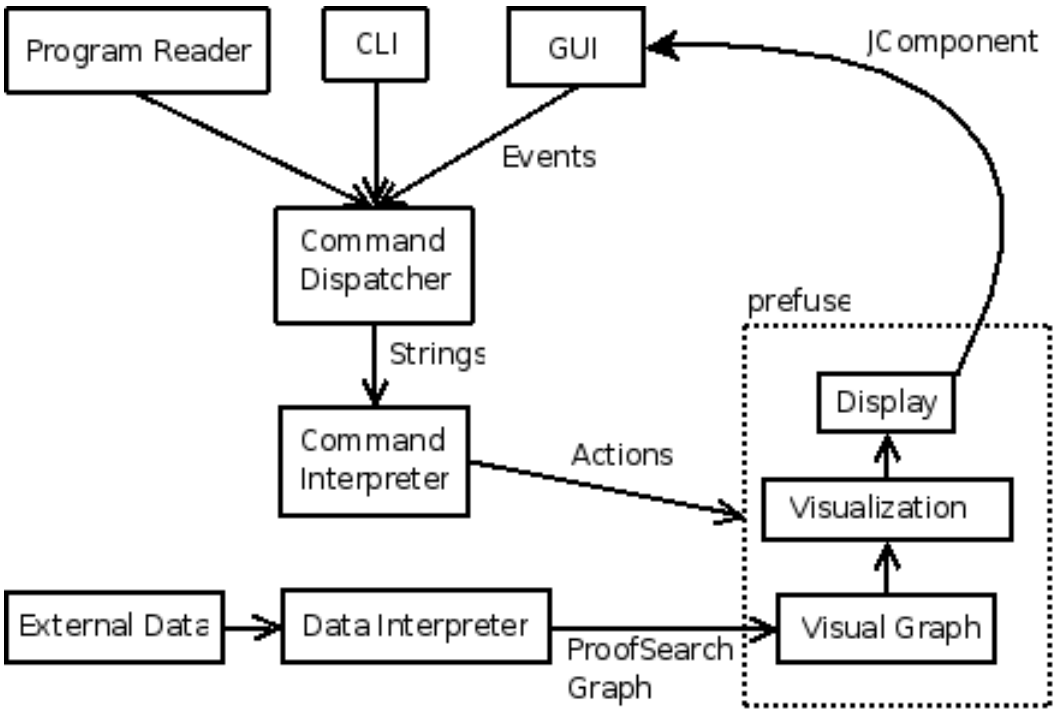


Fig. 2. The architecture of ViPrS, showing the data-flow relationships between the various components.

search. Its main utility is in showing what part of the proof search is currently in the viewing pane, by means of a rectangle surrounding that area. The minimap also allows the user easy navigation over large distances in the proof search by clicking on the area to be examined more closely.

3.2 Architecture

To support these features, the design of ViPrS needs to strike a balance between core features and the flexibility of a programmable system. Our architecture can been seen in figure 2. Extensibility was a driving factor in this arrangement of the functionality we have developed.

At the foundation layer, SILK provides the collection of nodes constituting a proof search to be visualized. On top of this source data, the visualization component maintains an internal abstract model of the proof search graph.

From that abstract model, we derive a concrete visualization of the graph, with fields for all of the values salient to a visual display, including position, text, color, size, shape, and so forth. The main display and minimap are each a view onto this visualization. [5] These two displays, along with the other interface elements, are uniformly represented as Java Swing `JComponent` objects.

The command dispatcher responds to button presses and entries in the command line window by passing the code to be run to the command interpreter. The command interpreter, which holds references to the relevant objects of interest,

[5] This is a slight simplification–to improve the utility of the minimap, it actually keeps a separately derived visualization in which nodes are always displayed without text and at a fixed size.

evaluates the given code, and then asks the visualization system to update itself to reflect possible changes.

3.3 Implementation

3.3.1 Jython

Jython (3) provides the Python interpreter used as the back end of the CLI. It is a Python interpreter written entirely in Java, and allows interaction (such as function invocation and object reference) between compiled Java code and Python code in the interpreter.

3.3.2 Online Update

We use an observer pattern to let the visualization system register its interest in the changes presented by telling SILK to continue its search. This maintains the loose coupling between SILK and ViPrS. The observer object queues the changes presented by SILK, and then applies them to the visualizer's model of the search space at controlled points, where such changes won't upset the visualization. The user determines the number of search steps between updates, and can change this dynamically during a run.

The tree-like treatment of the DAG described in section 3.1.1 greatly simplifies this process. The primary parent completely determines the position of each node, so adding nodes to the DAG only spreads the display in the same way that adding nodes to a tree would do. Deletion of a node's primary parent without deletion of the node itself causes the primary parent to change, moving a subgraph of the display to a different region of the DAG. This is potentially more disruptive to the overall layout than the addition of nodes, but it still does not cause significant repositioning of many parts of the tree.

3.3.3 prefuse

The prefuse visualization toolkit [6] (10) is the open source software package used to drive the visual component of ViPrS. It is a software toolkit specifically designed for the visualization of graphs. It provides classes to model a graph with various visual characteristics, and then renders that model to a Java Swing `JComponent` which is embedded in the GUI.

Data being visualized by prefuse goes through a sequence of transformations, taking it from its raw, external form, through an internal abstract model, to an extension of the model to include concrete visual details, and finally rendering that visual model on a display. The SILK proof search data structures constitute the external form. From the graph implicit in this collection of objects, we create an explicit graph in an extension of prefuse's provided `Graph` class. We then augment this in a `VisualGraph` object that adds in details like layout, size, and color. Each rendered `Display` is a window onto that fully elaborated visualization of the graph structure, to which various interaction controls (e.g. pan, zoom) can be attached.

While prefuse provides various implementations of each of these transformation steps and interaction controls, they are often not precisely suited to the purposes

[6] Per the conventions of its developers, prefuse's name is written in lower case

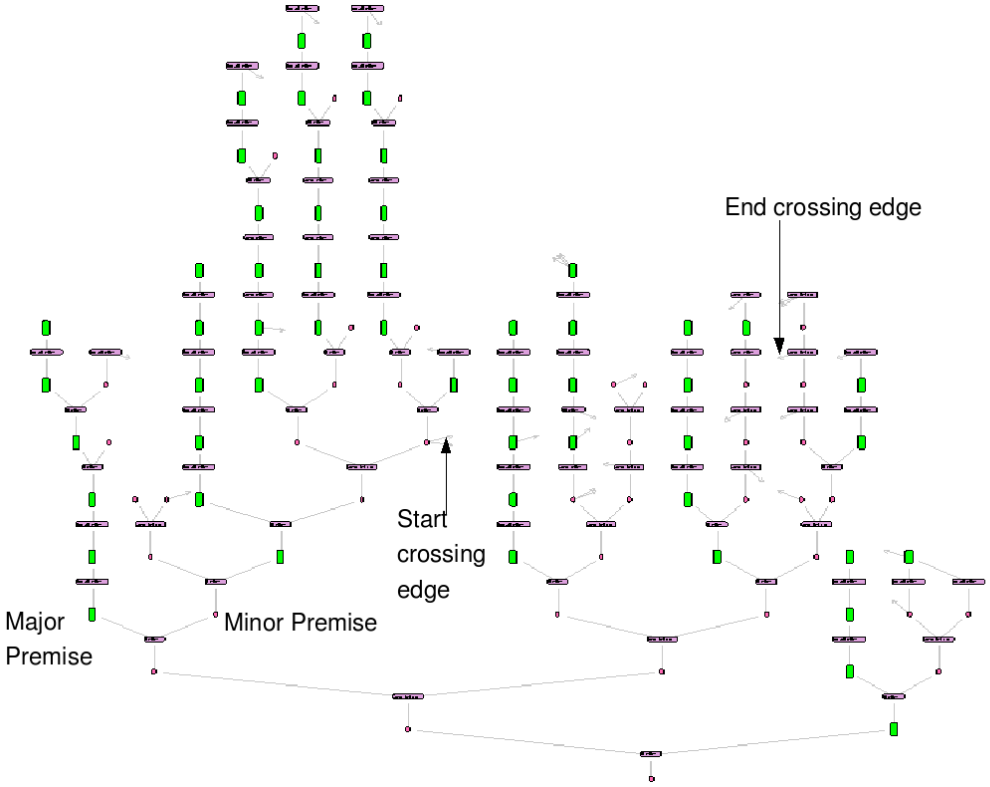


Fig. 3. Display of a simple proof. The darkest, horizontal nodes represent rule applications. The gray nodes are formula occurrences, and in this image the major premises have been highlighted. The restriction of major premises to the upper portion of each branch is a property of *normal* natural deductions (20; 4).

of this visualization application. Thus, many of the implementations had to be tailored to our purposes. For example, the included tree-like layout didn't take account of its input being a rooted, directed graph.

4 Discussion

The ViPrS tool has proven itself to be very useful, even while in its developmental stage. First, in reporting research to sponsors, it was valuable to be able to display a visualization of a proof in order to explain its structure and some of the difficulties involved in proof search. Figure 3 is a ViPrS screen shot displaying a SILK proof constructed from an IKL translation of SUMO (the Suggested Upper Merged Ontology, (17)).

The tool has also been very useful for debugging of SILK. During proof search, certain heuristics depend on the height of given nodes. When part of the space did not seem to be getting explored, work in the debugger revealed a node for which the height was incorrectly recorded. Finding the root of this problem would have been extremely tedious through a standard debugger or logger, but a snapshot of the search space represented in ViPrS identified the roots of all mislabeled subtrees immediately. This was done without any change to the visualization source code. A simple recursive Python command was defined through the CLI which colored all nodes having height one green (simulated in figure 4). The nodes with the incorrect

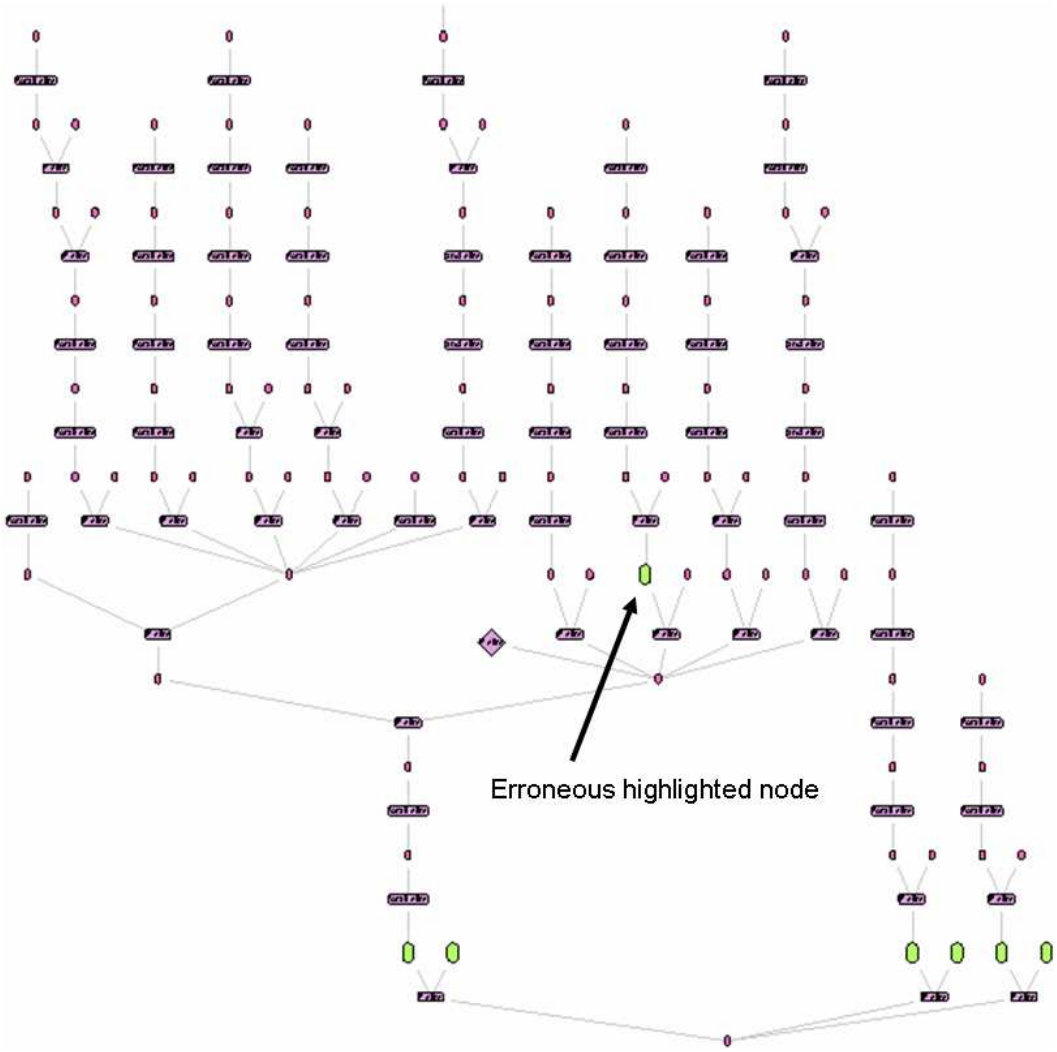


Fig. 4. The tree is searched and all nodes reporting height one are highlighted. The node with erroneous height stands out immediately. Traditional use of a debugger or text output would have required tedious comparison and manual inspection.

heights were immediately visible, and the state of each could be queried directly through the CLI. The visualization layer has no direct reference to the reasoning engine's node height values. Rather, the effect was generated by applying the ability described in section 3.1.2 of the CLI to access SILK data structures directly.

Beyond detection of bugs (coding errors), the real goal of the system is to understand the structure of the search space in order to improve search efficiency. The full search space can potentially contain redundant subtrees. These are recognized during search and treated specially so that redundant search does not occur. However, the use of Skolem functions and Herbrand terms during search introduces the possibility of parts of the search space which are redundant without being identical (rather, they are identical up to variable renaming). Proper treatment of these redundancies is best handled theoretically, but one often alternates between empirical algorithm exploration and theoretical development. When a particular proof search failed to yield a proof, the large scale visualization in figure 5 of part of the search

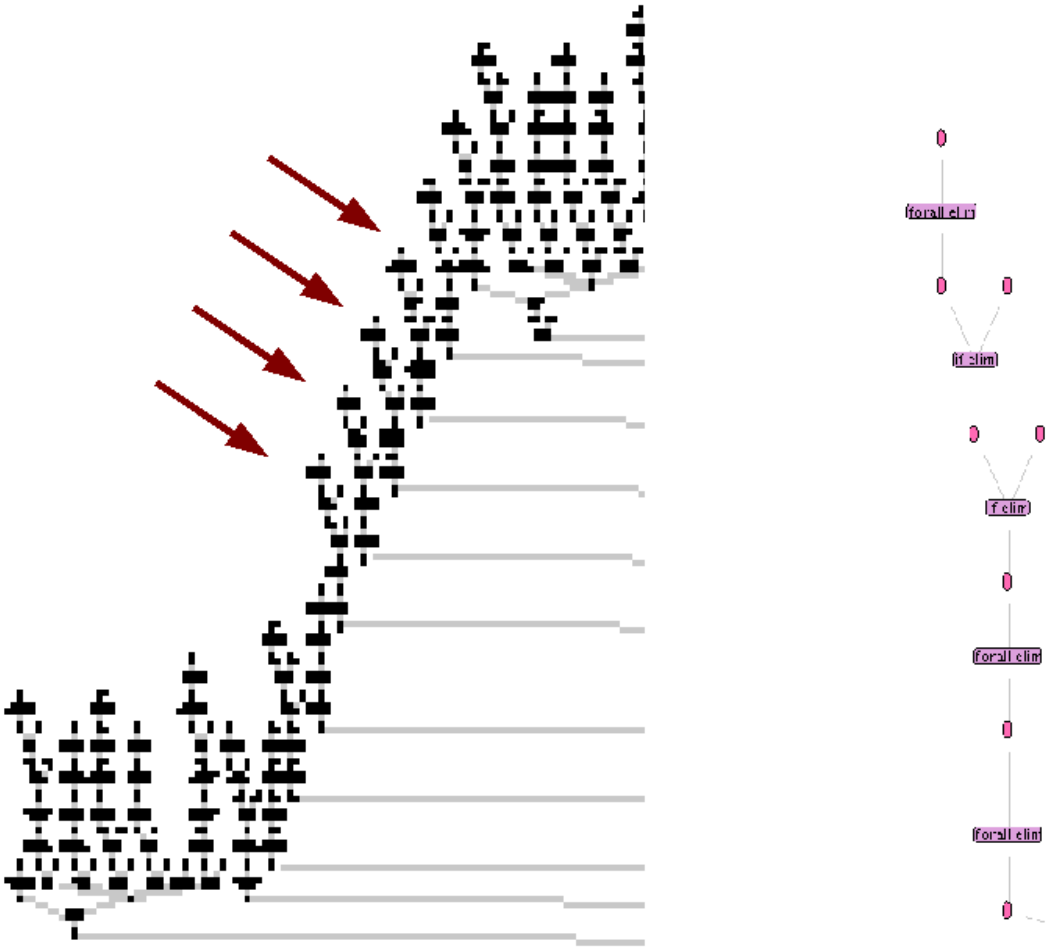


Fig. 5. The leftmost side of a large search tree after fifty steps of search. The close-up structure on the right appears at each of the subtrees indicated by the arrows. The tool-tip feature (described in section 3.1.1) immediately allows us to see that the root of each of these subtrees is identical up to choice of a newly introduced free variable.

space quickly suggested that redundant trees were being searched. Finer inspection then provided the determination that the redundancies seen were due to variable renaming and not due to coding errors with respect to the more straightforward type of redundancy.

5 Future Work

Extensibility was the primary consideration in the design of the ViPrS architecture. The simplest type of extension, as described above, is the addition of Python functions that can be called from the CLI. Colorings which highlight structural properties of the search space have been written already, such as those which highlight nodes that have been successfully proven, or those which are major premises (as the ND search is driven by restriction to *normal* deductions).

One important accomplishment is the decoupling of the minimap from the primary display. Although the two diagrams currently represent the same view at different scales, the minimap can render a different view entirely. One possibility is embedding nodes into points on the plane without rendering individual nodes or

edges. The points of the plane could be colored to represent properties of nodes, such as a heat map representing the ages of nodes to indicate the order in which parts of a tree were visited. Other maps might represent the number of free variables occurring in each node, or the number of instances of members a given set of formal symbols (names) from the knowledgebase that might be of particular interest to a user.

Collapsing of nodes is crucial to readability of the graphs, but the appropriate mechanisms for collapsing DAGs are not exactly clear. The current implementation hides as much as possible, meaning that all nodes "above" a collapsed node are hidden, even if they are above through crossing edges (the arrows) rather than direct edges; as a result collapsing may hide distant nodes unexpectedly. Convenient techniques for allowing the user to control this functionality and for indicating points which have been collapsed remotely need to be developed.

One of the motivations of ND theorem proving is to provide a more human-readable proof. The fact that intercalation always finds normal proofs can be exploited to automatically collapse parts of the visualization based on minimal nodes and the branch structure of normal derivations. The plans which SILK generates to provide guidance for proof search should give guidance for collapsing as well, and future work should provide the ability to present a plan which expands to the underlying proof.

Further extensions of SILK will likely also lead to extensions of the visualization. SILK is currently being extended to interoperate with BRUSE, a Bayesian network software system which provides soft evidential updating (26; 27). SILK proofs are being converted into network fragments as a means of automating network construction, and the ViPrS system is likely to be extended to provide visualization of the resulting Bayesian networks.

An appealing direction for ViPrS not originally considered is to allow arbitrary reasoning engines to make use of ViPrS for visualization. The current system with SILK could be used to read in arbitrary proofs and proof search DAGs specified in SILK's XML format, so an easy extension that might be useful is simply to let SILK read other standard formats such as TPTP (24) and PML (14). Of potentially greater utility is development of a Java API which developers of reasoning engines could use in order to provide interactivity with any algorithms being developed. This would allow, for example, different reasoners (possibly using different logical calculi) to be run in parallel in order to compare their approaches to various problems of interest.

References

[1] Beckert, B. and J. Posegga, *leanTAP: Lean tableau-based deduction*, Journal of Automated Reasoning **15** (1995), pp. 339–358.
URL http://citeseer.ist.psu.edu/beckert95leantap.html

[2] Berners-Lee, T., J. Hendler and O. Lassila, *The semantic web*, Scientific American (2001).

[3] Bock, F., *Jython project* (2007).
URL http://www.jython.org/

[4] Byrnes, J., "Proof Search and Normal Forms in Natural Deduction," Ph.D. thesis, Department of Philosophy, Carnegie Mellon University (1999).

[5] Delugach, H., *Common logic - a framework for a family of logic-based languages*, International Standards Organization Final Committee Draft (2005).
URL http://cl.tamu.edu/docs/cl/32N1377T-FCD24707.pdf

[6] Fiedler, A., *P.rex: An interactive proof explainer*, in: A. L. R. Gore and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artifical Intelligence (2001), pp. 416–420.

[7] Genesereth, M. R. et al., *Knowledge interchange format*, draft American National Standard (1998).
URL http://www.ksl.stanford.edu/knowledge-sharing/kif/

[8] Hayes, P. and C. Menzel, *A semantics for the knowledge interchange format*, in: *Proc of IJCAI 2001 Workshop on the IEEE Upper Ontology*, 2001.
URL http://reliant.teknowledge.com/IJCAI01/HayesMenzel-SKIF-IJCAI2001.pdf

[9] Hayes, P. and C. Menzel, *IKL specification document* (2006).
URL http://www.ihmc.us/users/phayes/IKL/SPEC/SPEC.html

[10] Heer, J., S. K. Card and J. A. Landay, *prefuse: a toolkit for interactive information visualization*, in: *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems* (2005), pp. 421–430.

[11] Horrocks, I. and A. Voronkov, *Reasoning support for expressive ontology languages using a theorem prover*, in: *Foundations of Information and Knowledge Systems (FoIKS)*, 2006.

[12] McCune, W., *Prover9* (2008).
URL http://www.cs.unm.edu/~mccune/prover9/

[13] McGuinness, D. L. and P. P. da Silva, *Explaining answers from the semantic web: The inference web approach*, Journal of Web Semantics **1** (2004), pp. 397–413.
URL http://browser.inference-web.org/

[14] McGuinness, D. L., L. Ding, P. P. da Silva and C. Chang, *PML 2: A modular explanation interlingua*, in: *AAAI 2007 Workshop on Explanation-aware Computing*, Vancouver, British Columbia, Canada, 2007.

[15] Meier, A., *System description: Tramp - transformation of machine-found proofs into natural deduction proofs at the assertion level*, in: D. McAllester, editor, *Proceedings of the 17th Internal Conference on Automated Deduction*, number 1831 in Lecture Notes in Artifical Intelligence (2000), pp. 460–464.

[16] Melis, E. and A. Bundy, *Planning and proof planning*, in: S. Biundo, editor, *ECAI-96 Workshop on Cross-Fertilization in Planning*, Budapest, 1996, pp. 37–40.

[17] Niles, I. and A. Pease, *Towards a standard upper ontology*, in: *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems* (2001), pp. 2–9.

[18] Panton, K., C. Matuszek, D. Lenat, D. Schneider, M. Witbrock, N. Siegel and B. Shepard., *Common sense reasoning—from cyc to intelligent assistant*, in: Y. Cai and J. Abascal, editors, *Ambient Intelligence in Everyday Life*, number 3864 in LNAI, Springer, 2006 pp. 1–31.

[19] Pfenning, F., "Proof Transformations in Higher-Order Logic," Ph.D. thesis, Carnegie Mellon University, Pittsburgh (1987).

[20] Prawitz, D., "Natural Deduction: A Proof-Theoretic Study," Dover Publications, 2006.

[21] Sieg, W. and R. Scheines, *Searching for proofs (in sentential logic)*, in: L. Burkholder, editor, *Philosophy and the Computer*, Westview Press, Boulder, 1992 pp. 137–159.

[22] Siekmann, J., C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C. Wirth and J. Zimmer, *Proof development with omega*, in: A. Voronkov, editor, *Proc. CADE-18*, number 2392 in LNAI (2002).

[23] Steel, G., *Visualising first-order proof search*, in: D. L. C. Aspinall, editor, *Proceedings of User Interfaces for Theorem Provers 2005*, 2005, pp. 179–189.

[24] Sutcliffe, G. and C. Suttner, *The tptp problem library: Cnf release v1.2.1*, Journal of Automated Reasoning **21** (1998), pp. 177–203.

[25] Trac, S., Y. Puzis and G. Sutcliffe, *An interactive derivation viewer*, , **174**, 2007, pp. 109–123.

[26] Valtorta, M., J. Byrnes and M. Huhns, *Logical and probabilistic reasoning to support information analysis in uncertain domains*, in: *Proceedings of the Third Workshop on Combining Probabilty and Logic (Progic-07)*, Canterbury, England, 2007.

[27] Valtorta, M., Y.-G. Kim and J. Vomlel, *Soft evidential update for probabilistic multiagent systems*, International Journal of Approximate Reasoning **29** (2002), pp. 71–106.

Panoptes

An Exploration Tool for Formal Proofs

William M. Farmer[1,2] Orlin G. Grigorov[1,3]

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

Abstract

Proof assistants aid the user in proving mathematical theorems by taking care of low-level reasoning details. Their user interfaces often present proof information as text, which becomes increasingly difficult to comprehend as it grows in size. Panoptes is a software tool that enables users to explore graphical representations of the formal proofs produced by the IMPS Interactive Mathematical Proof System. Panoptes automatically displays an IMPS deduction graph as a visual graph that can be easily manipulated by the user. Its facilities include target zooming, floating information boxes, node relabeling, and proper substructure collapsing.

Keywords: IMPS, deduction graph, proof tree, theorem prover, graph visualization, OCaml, OpenGL.

1 Introduction

A proof assistant is a software system for developing formal proofs. The user guides the development of an attempt to prove a conjecture, while many of the low-level details are done automatically by the proof assistant. Proof assistants are usually not equipped with sophisticated tools for exploring the "tree" of formulas that is produced by a proof attempt. However, the proof structure created in proving a conjecture can sometimes grow to a large size involving hundreds of formulas and inferences. In this case, the user can easily lose his or her way when exploring the proof and can miss seeing different parts of the proof with similar structure that could be merged if identified.

Panoptes, named after the all-seeing giant of Greek mythology, is a software system for exploring the proof structures produced by the IMPS Interactive Mathematical Proof System [3,4,5]. The proof structures that IMPS creates are certain kinds of graphs called deduction graphs. Panoptes automatically displays an IMPS

[1] This research was supported by NSERC.

[2] Email: wmfarmer@mcmaster.ca

[3] Email: ogrigorov@gmail.com

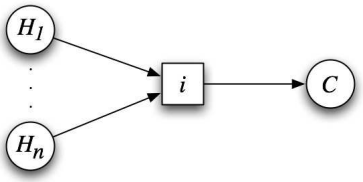
This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

deduction graph as a visual graph that can be manipulated by the user. Although Panoptes is designed to work with IMPS, it focuses on facilities that would be useful to many other proof assistants. This paper describes the facilities that Panoptes provides and gives an overview of its implementation.

2 Deduction Graphs in IMPS

A *deduction graph* [3] in IMPS is a directed bipartite graph used to represent a proof or proof attempt. It contains two types of nodes and arrows that connect a node of one type to a node of the other type. A *sequent node* represents a sequent consisting of a single formula called the *assertion* and a finite set of assumptions called the *context*. An *inference node* represents an inference from a finite set of sequents (the hypotheses) to a single sequent (the conclusion). An inference node has arrows pointing to it from the sequent nodes representing its hypotheses and an arrow pointing from it to a sequent node representing its conclusion. The *root node* of a deduction graph is a distinguished sequent node in the graph that represents the sequent to be proved.

For example, the figure



is a small deduction graph consisting of $n + 1$ sequent nodes and 1 inference node. This deduction graph represents the inference of the conclusion held by the sequent node C from the hypotheses held by the sequent nodes $H_1, \ldots, H_n$.

Since a sequent node can have more than one arrow into it, any number of alternate strategies can be represented in the deduction graph for proving a given sequent. Thus a deduction graph generally does not represent a single proof attempt, but rather a set of intertwined proof attempts. Deduction graphs may contain cycles and may not be connected. A sequent node is said to be *grounded* if it is known to be valid. A deduction graph is a *proof* if its root node is grounded. A deduction graph that is a proof does not necessarily represent a proof tree; it may contain garbage, i.e., parts of the deduction graph that represent unneeded or unfinished alternate proof attempts.

3 Description of System

Panoptes serves as an add-on application, which runs concurrently with IMPS. It provides a graphical visualization of the deduction graph, which is synchronized with the internal representation of the deduction graph upon request by the user. The tool provides a large set of functions to ease the process of exploring the structure of the graph and understanding the logical development of the proof. The main design goal was to make graph manipulation easy, and the result is a program that

provides an almost playful way of exploring the deduction graph. Another design goal was to make it easy to port to other theorem provers by encapsulating into a separate module the part that processes the input from the theorem prover.

3.1 Functionality

Apart from the graphical visualization of the deduction graph on the screen, the system provides a range of useful functionality to the user. The following are some of the major options available.

- **Target zooming.** The user can zoom in and out on parts of the graph by just pointing with the mouse and holding down a button. This is quite different from the standard way of zooming first and then scrolling to reach the point of interest, which can easily lead to confusion and disorientation of the user.
- **Collapsing.** Parts of the graph can be collapsed into special inference and sequent nodes. For instance, if the validity of a sequent node is reduced to the validity of one or more other sequent nodes through a number of proof steps, the user has the option to collapse all these steps into a special inference node, which consolidates the reasoning that reduces the goal to the subgoals. Similarly, cycles of equivalent sequent nodes can be collapsed into a special sequent node. Collapsing is very important when dealing with large deduction graphs since it enables secondary information to be hidden without compromising the semantic integrity of the deduction graph representation.
- **Labeling.** The user can freely label nodes and parts of the graph so that these components can be identified with names that are more meaningful than the names generated by the system.
- **Floating information boxes.** Each node, regardless of its type, contains some information. In the case of a sequent node, this information comprises the sequent represented by the node. An inference node contains the inference rule that generated the represented inference, and a collapsed inference node contains the (possibly large) part of the graph that is hidden by the collapsing. Each node in the deduction graph has an information box that contains the information associated with it. These information boxes can be toggled between visible and nonvisible states. Additionally, when visible, an information box has a direct visible link to its associated node, which further enhances the efficiency of presenting the information to the user. Of course, these information boxes can be scaled, repositioned, and manipulated in many ways by merely pressing a button or dragging the mouse.
- **History of operations.** A comprehensive history of the operations applied to a deduction graph is kept at all times, so that the user can easily revert back to an earlier arrangement of the graph on the screen. Also, this function is important for preserving the effort invested into rearranging the graph between proof steps, which is possible due to the fact that IMPS only adds new nodes, but never removes nodes from the deduction graph.
- **Automatic layout and manual rearranging.** Upon startup, Panoptes provides an initial layout of the deduction graph. This allows Panoptes to fit the

whole graph in the screen space provided by the system and also to minimize the crossing of edges as much as possible. In addition, the user is able to drag and drop components of the graph to either improve or modify the layout according to his or her preference, while the program automatically protects the connections (the arrows) between the nodes.

Additionally, appropriate automatic labeling and numbering of repetitions (in the case of inference nodes representing applications of the same inference rule) is automatically performed by the system. The power of color is also utilized: grounded nodes are colored in green, repeated nodes (i.e., nodes that complete a cycle or merge proof directions) in brown, collapsed inference nodes in purple, etc.

3.2 Implementation

A fully functional prototype of the proposed system has been developed in Objective Caml (OCaml) [9] using the LablGL library [6] that implements an interface to OpenGL [7] in OCaml.

The choice of OCaml as the programming language was made on the basis of its features, such as its support for modular design, as well as its automatic garbage collection system, type inferencing, and allowance for both imperative and functional programming styles. All of this adds up to a versatile language that is suitable for developing large projects with a reduced chance for programming errors and an increased runtime stability. Furthermore, OCaml is available for many operating systems including Linux and Mac OS X, which makes the tool portable to all systems that currently support IMPS.

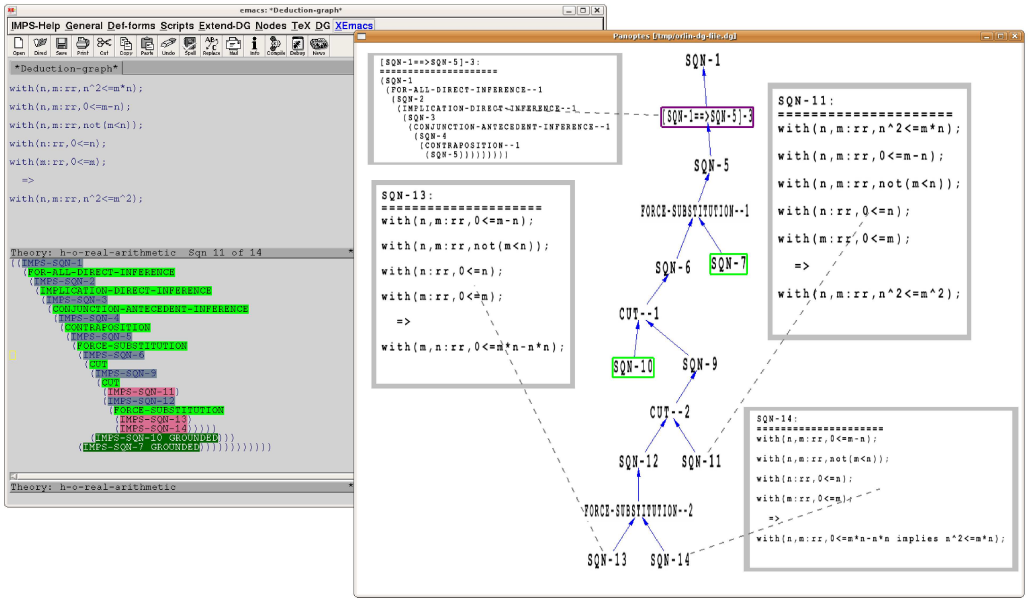
As for the graphical library, OpenGL is usually associated with three-dimensional graphical visualizations, but the system uses these capabilities for implementing different techniques. For instance, moving the graph or selected components of the graph closer or further away from the viewer creates the effect of zooming in contrast to the usual method of merely scaling the image. The advantage lies in OpenGL being a direct API to the 3D instruction set of the graphical hardware, and as such it provides a performance unmatched by the standard way of drawing graphics on the screen. The result is an application, which delegates all graphical computations and manipulations to the GPU, rather than to the CPU of the host machine, leaving the latter fully available for other work (such as that done by the IMPS reasoning engine). Consequently, a computer system equipped with a reasonably modern graphics card would be capable of running the prototype smoothly without burdening the user with unnecessary lags and delays.

3.3 Availability and Screenshots

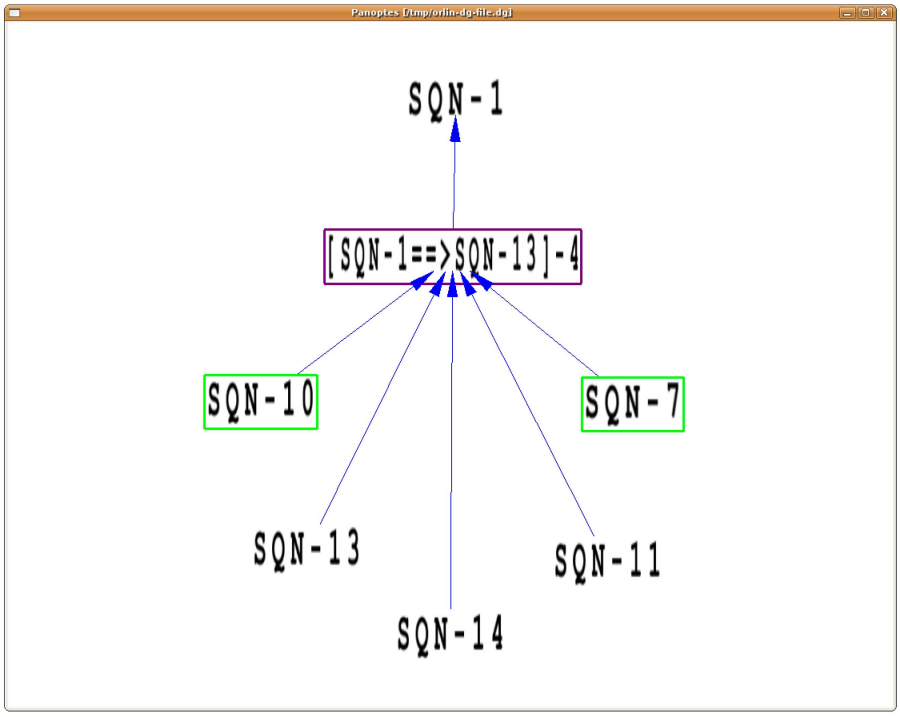
The source code and instructions for compiling and running the system are available at the Panoptes home page: http://imps.mcmaster.ca/ogrigorov/panoptes/. The home page also provides access to a demo of the system, as well as detailed documentation of the requirements, design, and implementation of the system [8].

A few screenshots are displayed below, although the complete functionality, features, and performance of Panoptes cannot be demonstrated by static pictures:

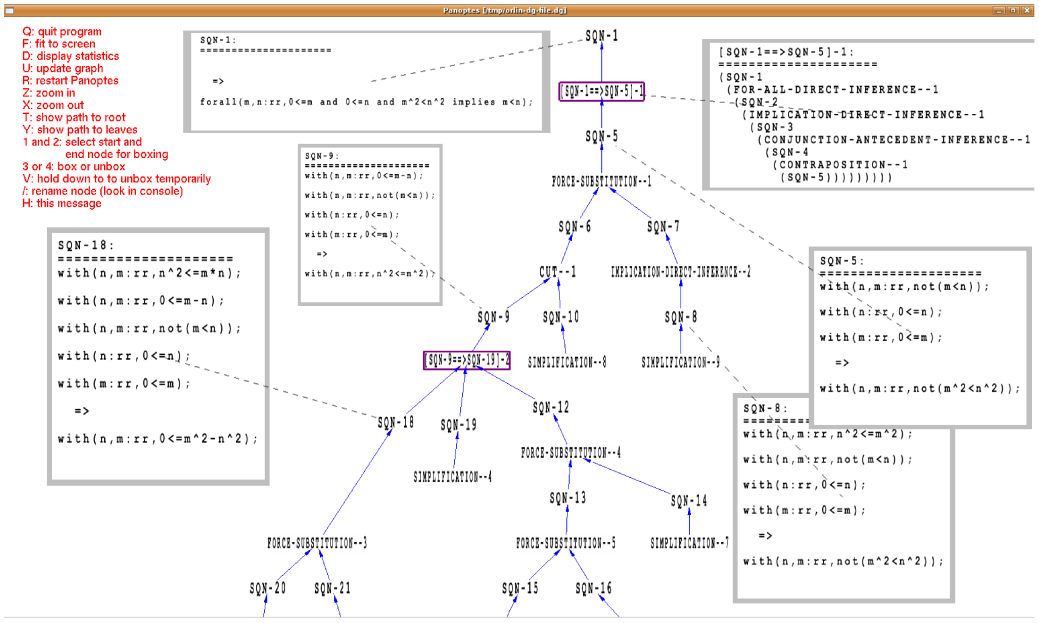
- **Screenshot 1.** IMPS and Panoptes working side by side.



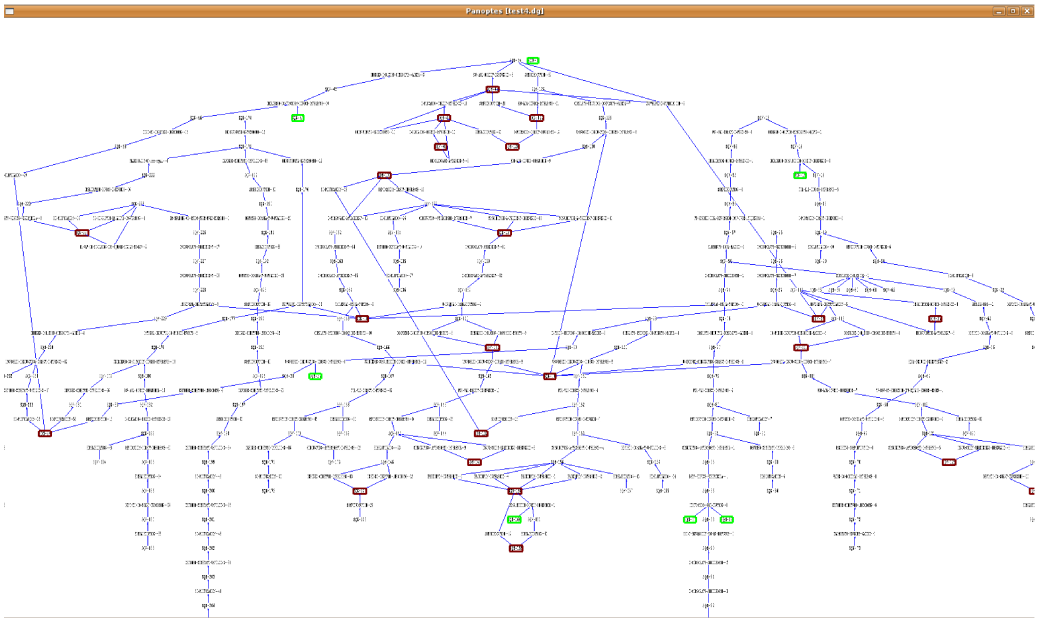
- **Screenshot 2.** The deduction graph from the previous screenshot is fully collapsed.



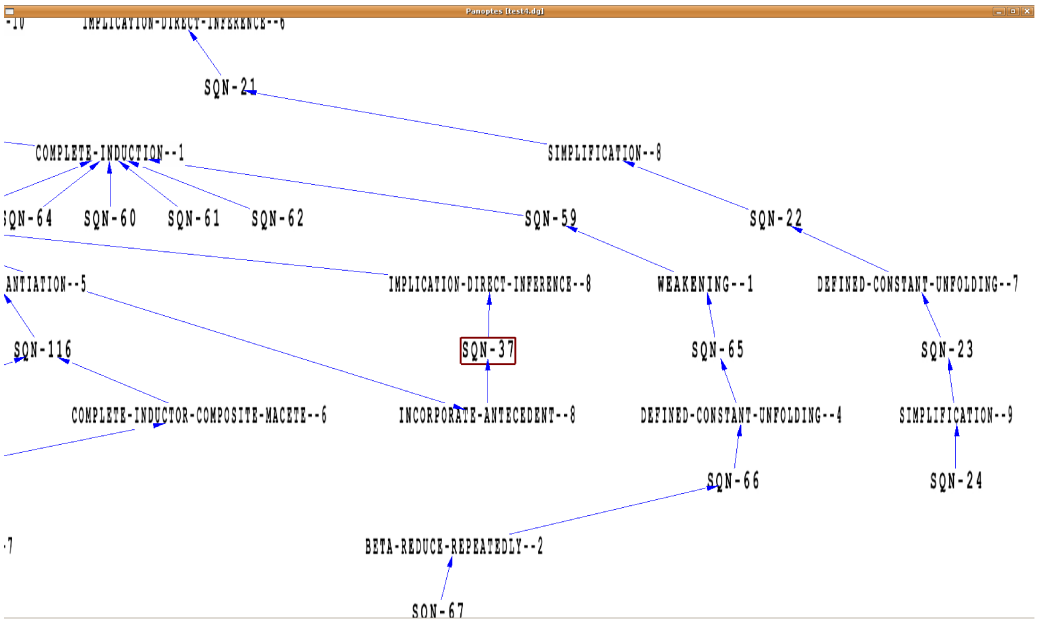
- **Screenshot 3.** A rendering of a larger graph. A help screen with the available commands is visible, and a few subgraphs are collapsed. Some information boxes for certain nodes are visible.



- **Screenshot 4.** A very large graph consisting of hundreds of nodes.



- **Screenshot 5.** Zoomed section of the large graph from the previous screenshot.



4 Related Work

A number of people have invested time and effort in improving the user experience with theorem provers. The work that deals with matter similar to the ideas behind Panoptes is described below.

Developed in Java, the *Interactive Symbolic Visualization of Semi-Automatic Theorem Proving* system [1] presents formal proofs produced by the ACL2 theorem prover [10] in the form of cone-shaped three-dimensional graphs on the screen. The user can rotate the visualization in order to look at all angles, as well as to open detached information windows with information about the nodes. Rather than labeling, colors are used to differentiate between different nodes.

Another proof assistant, PVS [11], offers graphical display of proof trees for users with Tcl/Tk (http://www.tcl.tk/) installed on their systems. The visualization appears to have limited functionality for the manipulation of the tree display, although it too offers opening of windows with details about the nodes.

The Interactive Derivation Viewer [15] renders derivations that are written in the TPTP [14] language [13], and provides an interface that allows one to quickly explore different features of the derivation. The display is very customizable and the user has access to functions like zooming, fit to height or width of the drawing pane, etc. The user can also see the information associated with each node, although it appears difficult to have such information visible simultaneously for more than one node.

LOUI (Lovely ΩMEGA User Interface) [12] offers a graphical representation of the logical proofs created by the ΩMEGA system [2] in the form of a graphical structure that is a proper tree. It divides the nodes into different categories and differentiates them visually from one another by assigning different shapes and

colors to each category. Similarly to Panoptes, the user has different facilities to manipulate the appearance of the proof tree, such as zooming, scrolling, focusing, and other functionalities.

5 Future Work

Future work can take different directions.

New features, such as a facility to syntactically or semantically compare and calculate a degree of similarity between sequent nodes will further enhance the effectiveness of Panoptes. Also, exploiting the 3D capabilities provided by OpenGL can result in the ability to stack different proof attempts of a particular goal perpendicularly to the screen plane. The user can then use commands to spin through the different proof attempts or even look at the graph from a different angle for obtaining different perspective and understanding.

Even though Panoptes was successfully tested and performed without noticeable lags on a system equipped with two 30" Apple Cinema HD^{TM} displays, each capable of 2560×1600 pixels resolution, it is yet to be tested on a system connected to a large wall of screens (i.e., 4×3 units with combined resolution of 10,240×4,800 pixels). Since the current design and implementation concentrate on optimizing the program for better runtime performance, it will prove beneficial if the tool is running smoothly on such large screen systems.

Since the dataflow between IMPS and Panoptes is currently happening only in one direction (data can travel from IMPS to Panoptes, but Panoptes cannot send messages to IMPS), expanding Panoptes into a standalone user interface to completely replace the existing Emacs-based user interface of IMPS is the most ambitious plan of all. This is due to the enormous amount of details that need to be accounted for, although given sufficient time and dedication, it is completely achievable.

6 Conclusion

The users of proof assistants require effective tools for exploring the proof structures they create. Panoptes demonstrates the kind of functionality that these tools need to provide. Its implementation utilizes the powerful features offered by today's computer graphics technology. The ideas used in Panoptes for exploring IMPS deduction graphs can be readily applied to other proof assistants. Moreover, Panoptes has been designed so that the code itself can be ported to other proof assistants as well.

References

[1] Bajaj, C., S. Khandelwal, J. Moore and V. Siddavanahalli, *Interactive symbolic visualization of semi-automatic theorem proving*, Technical Report TR-03-37, University of Texas at Austin (2003).

[2] Benzmller, C., L. Cheikhrouhou, D. Fehrer, A. Fiedler, Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann and V. Sorge, *Omega: Towards a mathematical assistant*, in: W. McCune, editor, *Automated Deduction—CADE-14*, Lecture Notes in Computer Science **1249** (1997), pp. 252–255.

[3] Farmer, W. M., J. D. Guttman and F. J. Thayer, *IMPS: An Interactive Mathematical Proof System*, Journal of Automated Reasoning **11** (1993), pp. 213–248.

[4] Farmer, W. M., J. D. Guttman and F. J. Thayer, *The IMPS user's manual*, Technical Report M-93B138, The MITRE Corporation (1993), online at http://imps.mcmaster.ca/.

[5] Farmer, W. M., J. D. Guttman and F. J. Thayer Fábrega, *IMPS: An updated system description*, in: M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, Lecture Notes in Computer Science **1104** (1996), pp. 298–302.

[6] Garrigue, J., *An Objective Caml interface to OpenGL* (2007), online at http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgl.html.

[7] Gold Standard Group and SGI, *OpenGL—The industry standard for high performance graphics*.

[8] Grigorov, O. G., "Panoptes: An Exploration Tool for Formal Proofs," M.Sc. in Computer Science thesis, McMaster University (2008), online at http://imps.mcmaster.ca/ogrigorov/panoptes.

[9] INRIA, *The Caml language* (2008), online at http://caml.inria.fr.

[10] Kaufmann, M. and J. Moore, *An industrial strength theorem prover for a logic based on common lisp*, Software Engineering **23** (1997), pp. 203–213.

[11] Owre, S., N. Shankar, J. M. Rushby and D. W. J. Stringer-Calvert, "PVS System Guide, Version 2.4," SRI International, Menlo Park, CA, USA, November 2001.

[12] Siekmann, J., S. Hess, C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis and V. Sorge, *ℒOUI: A distributed graphical user interface for the interactive proof system* ΩMEGA, in: R. C. Backhouse, editor, *User Interfaces for Theorem Provers*, number 98-08 in Computing Science Reports, Department of Mathematics and Computing Science, Eindhoven Technical University, 1998, pp. 130–138.

[13] Sutcliffe, G., S. Schulz, K. Claessen and A. V. Gelder, *Using the TPTP language for writing derivations and finite interpretations.*, in: U. Furbach and N. Shankar, editors, *IJCAR*, Lecture Notes in Computer Science **4130** (2006), pp. 67–81.

[14] Sutcliffe, G. and C. Suttner, *The TPTP problem library*, J of Automated Reasoning **21** (1998), pp. 177–203.

[15] Trac, S., Y. Puzis and G. Sutcliffe, *An interactive derivation viewer*, Electronic Notes in Theoretical Computer Science **174** (2007), pp. 109–123.

User Interfaces for Portable Proofs

Paulo Pinheiro da Silva, Nicholas Del Rio

Department of Computer Science
University of Texas at El Paso
El Paso, TX, USA

Deborah L. McGuinness, Li Ding, Cynthia Chang

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, USA

Geoff Sutcliffe

Department of Computer Science
University of Miami
Coral Gables, FL, USA

Abstract

Portable proofs are a new and interesting way of integrating theorem provers into distributed environments like the web. This article reports on user interface's challenges and opportunities for theorem provers in such environments. In particular, this article reports on the design of user interfaces used for searching, browsing and inspecting TSTP problems when published as portable proofs.

Keywords: PML, TPTP, Inference Web, distributed proofs, user interfaces.

1 Introduction

The integration of theorem provers into hybrid distributed environments offers a new set of challenges and opportunities for providing explanations of system results. Distributed and portable proofs can be more interesting than stand alone proofs for a number of reasons: they may be deployed, stored and reused outside of environment in which they were generated; portions of the proof (e.g., individual inference steps or combinations of inference steps) may be named, annotated, and reused; support for portions of the proofs may be provided by other portions of the system (or even found by searching the web); axioms may have multiple lines of support; axioms can be asserted by multiple sources; and supporting evidence can be provided by multiple sources (instead of only the one source used in the original proof). We use the term *portable proofs* to refer to artifacts with these properties.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

In order for portable proofs to realize their full potential, innovative user interfaces are required. For example, consider the following tasks:

- Searching for proofs and proof fragments
- Searching for proof annotations for reuse
- Browsing proof annotations

For example, what are the design requirements for the query interface of a proof-aware search engine? Also, what are the design requirements for the presentation of the search results? Considering that the results can be entire proofs or just proof fragments, how can a user interface show the exact part of the proof is represented by the search result? Moreover, how can the user intuitively ask for more details of the results, whether the additional results are related to in-depth disclosure of proof details or to a better understanding of the proof structure? The browsing of proof annotation can become particularly challenging considering the amount of details that can be incorporated into proofs.

In addition to the design issues above, we see that traditional challenges related to proof presentation remain for portable distributed proofs:

- Conclusion presentation
- Complex proof presentation
- Browsing techniques that incorporate evidence and sources

In this paper, we address the challenges above. The rest of this paper is organized as follows. Section 2 describes a typical use of our Inference Web explanation environment tool suite in a theorem prover setting. Section 3 revisits Inference Web's Proof Markup Language (PML) used to encode portable proofs. Section 4 explains how portable proofs are extracted from PML documents. Section 5 describes how Inference Web's Search (IWSearch) can be used to search for both proofs and proof metadata on the web. Section 6 introduces ProbeIt - a tool that supports proof inspection. Section 8 summarizes the main results for this paper.

2 Motivating Use Case

We leverage the TPTP collection of problems and proofs as the setting for our use case. Consider a simple scenario where a user is interested in solving one of the problems and investigating a particular theorem prover's solution. (Later we will expand to investigating multiple prover's solutions for the same problem). Our initial use case is the "Aunt Agatha" problem `PUZ001+1` in the TPTP collection [11], and consider the SNARK system's [10] solution of the problem.

Proofs generated by theorem provers can be published on the web. However, a typical proof output by a theorem prover is not annotated with meta information such as generator, time, and context. In fact, a proof's content is typically restricted to a raw identification of derivations plus a brief mention of the name of the inference rule used in each derivation. Without annotations, proofs may be used to debug the reasoning within theorem provers, but may be of limited use when trying to identify many other important properties of proofs such as the authors of the provers or a proper description of the inference rules used.

In this case of SNARK solving the Aunt Agatha problem, we may want to annotate that the proof was generated by SNARK. To be more specific, considering the possibility that the proof steps can be distributed on the web, we may want to annotate that SNARK was responsible for each step of the proof for Agatha. Further, we want the annotation to say that SNARK was implemented by Mark Stickel who is affiliated with SRI International. More generically, metadata should be able to be added to explain every single aspect of a proof, including the theorem provers responsible for generating the proofs, the version of the implementation, inference rules used by the theorem provers, axioms used in each proof, etc. More interestingly, metadata is expected to be reused at proof generation time. For example, an inference rule may be used multiple times in a proof as well as to be reused in multiple proofs from. In this case, one should be able to create and identifier, i.e., a URIref, and to publish the metadata about the rule. With this identifier in place, the metadata can be reused as needed.

We consider the following issues with relation to user interfaces for distributed proofs.

(i) How to search for proof-related metadata on the web, e.g., how to search for SNARK metadata?

(ii) How to verify that proof metadata correctly corresponds to the object of concern, e.g., that SNARK metadata is about the theorem prover from SRI International and implemented by Mark Stickel?

(iii) How to understand the structure of a distributed proof?

(iv) How to visualize a richly annotated proof?

The use of PML and (more) IW tools on the full TSTP solution library is also described in [8]. In the rest of the paper, we further describe the interface to the tools we use to create a demonstration environment for distributed proofs.

3 Proof Markup Language

In our environment, we encode distributed proofs in the Proof Markup Language (PML) [4,7]. We do this in the setting of the Inference Web [5] explanation infrastructure, which includes a number of PML-literate tools and services such as proof browsers, e.g., ProbeIt [1] and the IW Local View, and search services e.g., IWSearch. Inference Web also includes the PML ontologies and and references a collection of PML documents already available on the Web. We have generated a collection of PML proofs for the TPTP problems [8] and made the collection available on the Web.

Different than other markup languages for mathematical documents such as OMDoc [3], PML focus is on the creation and handling of graphs used to represent information manipulation traces created by agents (i.e., humans or machines) to infer conclusions. These graphs may be used to encode a formal proof but they may also be used to encode incomplete information on how conclusions were inferred. Moreover, a single graph may include a single justification for a given conclusion but it may include many alternate justifications for the same conclusion. Moreover, PML can be used to encode any kind of conclusion while OMDoc prescribes a

precise way of encoding conclusions as formal logical sentences. Because of these characteristics, OMDoc is expected to have a better support for handling conclusions than PML since the conclusions need to conform to the OMDoc syntax. On the other hand, PML can be used to encode any kind of proof, including the proofs that can be encoded in OMDoc and informal proofs such as information extraction based on natural language processing [6].

In PML, NodeSet [1] and InferenceStep are the main constructs of portable proofs and web explanations.

A NodeSet represents a step in a proof whose conclusion is justified by any of a set of inference steps associated with the NodeSet. PML adopts the term "node set" since each instance of NodeSet can be viewed as a set of nodes gathered from one or more proof trees having the same conclusion.

- The `URIref` [2] of a node set is the unique identifier of the node set. Every node set has one well-formed URIref.
- The `hasConclusion` of a node set represents the expression concluded by the proof step. Every node set has one conclusion, and a conclusion of a node set is of type Information.
- The expression language of a node set is the value of the property `hasLanguage` of the node set in which the conclusion is represented. Every node set has one expression language, and that expression language is of type Language.
- Each inference step of a node set represents an application of an inference rule that justifies the node set's conclusion. A node set can have any number of inference steps, including none, and each inference step of a node set is of type InferenceStep. The inference steps are members of a collection that is the value of the property `isConsequentOf` of the node set. A node set without inference steps is of a special kind identifying an unproven goal in a reasoning process.

An InferenceStep represents a justification for the conclusion of a node set. Inference steps are anonymous OWL classes defined within node sets. For this reason, it is assumed that applications handling PML proofs are able to identify the node set of a inference step. Also for this reason, inference steps have no URIs.

- The rule of an inference step, which is the value of the property `hasRule` of the inference step, is the rule that was applied to produce the conclusion. Every inference step has one rule, and that rule is of type InferenceRule. Rules are in general specified by theorem prover developers. However, PML specifies three special instances of rules: *Assumption*, *DirectAssertion*, and *UnregisteredRule*. When specified in an inference step, the *Assumption* rule says that the conclusion in the node set is an explicit assumption. The *DirectAssertion* rule says that the conclusion of the node was provided by the sources associated with the inference step. The *UnregistredRule* says that the conclusion in the node set was derived by some unidentified rule. *UnregisteredRule*s allow the generation of proofs-like structures applying undocumented, unnamed rules.
- The antecedents of an inference step is a sequence of node sets each of whose con-

[1] PML concept names are typed in sans serif style and PML attribute names are typed in `courier` style.
[2] http://www.ietf.org/rfc/rfc2396.txt

clusions is a *premise* of the application of the inference step's rule. The sequence can contain any number of node sets including none. The sequence is the value of the property `hasAntecedent` of the inference step. The fact that the premises are ordered may be relevant for some rules such as *ordered resolution* [9] that use the order to match premises with the schema of the associated rule. For other rules such as modus ponens, the order of the premises is irrelevant. In this case, antecedents can be viewed as a set of premises.

- Each binding of an inference step is a mapping from a variable to a term specifying the substitutions performed on the premises before the application of the step's rule. For instance, substitutions may be required to unify terms in premises in order to perform resolution. An inference step can have any number of bindings including none, and each binding is of type VariableBinding. The bindings are members of a collection that is the value of the property `hasVariableMapping` of the inference step.
- Each discharged assumption of an inference step is an expression that is discharged as an assumption by application of the step's rule. An inference step can have any number of discharged assumptions including none, and each discharged assumption is of type Information. The discharged assumptions are members of a collection that is the value of the property `hasDischargeAssumption` of the inference step. This property supports the application of rules requiring the discharging of assumptions such as natural deduction's *implication introduction*. An assumption that is discharged at an inference step can be used as an assumption in the proof of an antecedent of the inference step without making the proof be conditional on that assumption.
- The engine of an inference step, which is the value of the property `hasInferenceEngine` of the inference step, represents the theorem prover that produced the inference step. Each inference step has one engine, which is of type InferenceEngine.
- The timestamp of an inference step, which is the value of property `hasTimeStamp` of the inference step, is the date when the inference step was produced. Time stamp is of the primitive type dateTime. Every inference step has one time stamp.

An inference step is said to be well-formed if:

(i) Its node set conclusion is an instance of the conclusion schema specified by its rule;

(ii) The expressions resulting from applying its bindings to its premise schemas are instances of its rule's premise schemas;

(iii) It has the same number of premises as its rule's premise schemas; and

(iv) If it is an application of the *DirectAssertion* rule, than it has at least one source, else it has no sources.

PML node set schemas and PML inference step schemas are defined as follows. A PML **node set schema** is a PML node set which has a conclusion that is either a sentence schema [3] or a sentence; which has a set of variable bindings that map

[3] A sentence schema is a sentence optionally containing free variables. An instance of a sentence schema S

free variables in the conclusion to constants; which has zero of more inference steps; and whose inference steps are either inference steps or **inference step schemas**. An inference step schema is an inference set of a node set schema whose antecedents are node set schemas.

4 Portable Proofs

Since a PML node set can have multiple inference steps and each antecedent of each of those inference steps can have multiple inference steps, a PML node set N and the node sets recursively linked to N as antecedents of inference steps represent a graph of alternative proofs of N's conclusion. In this section, we describe how to extract individual proofs of N's conclusion from that graph of alternative proofs. We shall call each such extracted proof a "proof from N".

We begin by defining a *proof* as a sequence of "proof steps", where each proof step consists of a conclusion, a justification for that conclusion, and a set of assumptions discharged by the step. "A proof of C" is defined to be a proof whose last step has conclusion C. A proof of C is conditional on an assumption A if and only if there is a step in the proof that has A as its conclusion and "assumption" as its justification, and A is not discharged by a later step in the proof. An unconditional proof of C is a proof of C that is not conditional on any assumptions. (Note that assumptions can be made in an unconditional proof, but each such assumption must be discharged by a later step in the proof.) Finally, proof $P1$ is said to be a *subproof* of $P2$ if and only if the sequence of proof steps that is $P1$ is a subsequence of the proof steps that is $P2$.

Given these definitions, we can now define the proofs that are extractable from a PML node set as follows: for any PML node set N, P is a "proof from N" if and only if:

(i) The conclusion of the last step of P is the conclusion of N;

(ii) The justification of the last step of P is one of N's inference steps S; and

(iii) For each antecedent A_i of S, exactly one proof from A_i is a subproof of P.

If N is a node set with conclusion C, then a proof from N is a proof of C.

5 Searching for Proofs and Proof Metadata

IWSearch is the search tool for the Inference Web Infrastructure. IWSearch was developed to overcome a number of limitations related to metadata management found in our past practice: (i) IWBase, Inference Web's registry-based metadata management system, provides limited mechanisms for accessing metadata entries – a user can only browse the type hierarchy of those entries to find entries; and (ii) no service is available to find and reuse PML provenance metadata published on the web. IWSearch searches over PML proofs and proofs' metadata that has published on the web, and thus focuses on providing access to proof elements that have already been registered in the database registry.

---

is a sentence that is S with each free variable replaced by a constant.

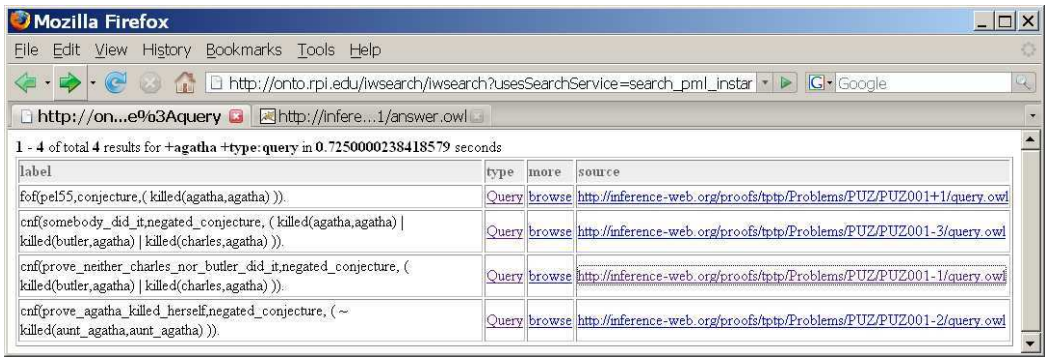


Fig. 1. IWSearch results for inference engine SNARK.

IWSearch is modeled off of SWOOGLE[2], which can be viewed as a search tool that "understands" RDF. Similarly, IWSearch can be viewed as a search tool that "understands" PML and OWL. IWSearch has an indexing phase that indexes terms, and also looks for particular terms using its knowledge of PML. Some metadata that IWSearch looks for includes:

- uri: Each PML object is identified by a unique identifier, i.e., a URI.
- type: Each PML object has one most-specific type, and IWSearch additionally indexes the other general types of a PML object. For example, an instance of inference engine metadata may also be considered as an instance of agent metadata.
- label: Each PML object has one label indicating its name. In the absence of name, the raw string content of the object is used. For example, an inference engine name is "SNARK 20070805r043", but for a conclusion, its label is its raw string content -" ? [X] : (lives(X) & killed(X,agatha))".
- source: Each PML object is extracted from one PML document, and the URL of the PML document is deemed as the source.

With the above metadata, IWSearch can provide much more than keyword search. By searching for `+SNARK +type:inferenceengine`, we can restrict the query and return only PML objects in the specified type. This is particularly useful if we want SNARK-generated proofs in PML to be annotated with the information that the proofs were generated by SNARK. Figure 1 shows the result of such a search. When querying for SNARK, one may find multiple metadata entries that are identified as SNARK. There are multiple reasons for this: more than one theorem prover is called SNARK; multiple versions of a single theorem prover; multiple metadata statements about the same theorem prover; or any combination of the previous reasons. By browsing the metadata, as in Figure 2, one may be able to verify multiple properties of the engine metadata such as authors, author's affiliation, engine's website as well as the creator of the metadata. By browsing the metadata, the user should be able to decide whether to reuse some existing metadata or even to create new metadata.

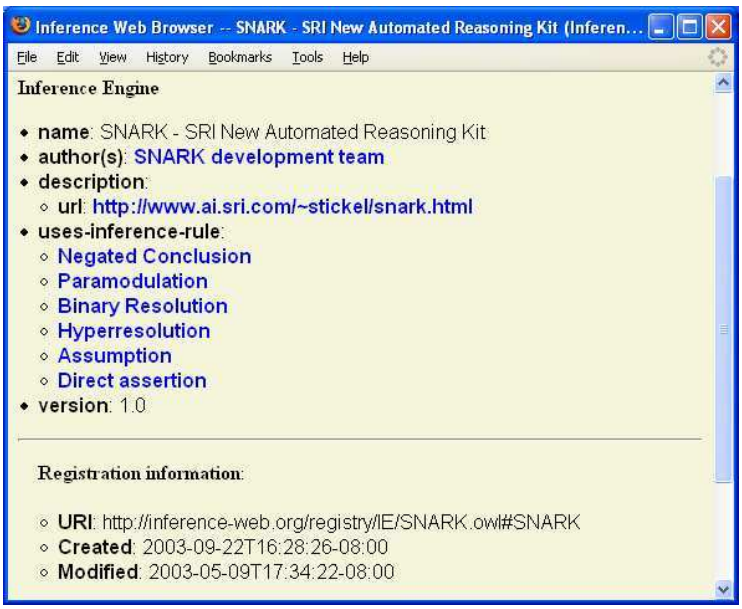


Fig. 2. Browsing metadata about an inference engine called SNARK.

6 Browsing Proofs

Probe-It! consists of three primary views to accommodate the different kinds of proof information: queries, proofs (or justifications), and provenance (or metadata).

The `query view` shows the links between a given problem and possible solutions for the problem. Upon accessing one of the solutions in the query view, Probe-It! switches over to the `global view` associated with that particular solution. All views are accessible by a menu tab, allowing users to navigate back to the query view from any other view.

The `global view` graphically shows the reasoning associated with a given solution. Probe-It! renders this information either as a directed acyclic graph (DAG) or as a tree. The example of a tree view of the SNARK's solution for the Agatha problem is shown in Figure 3. In this view, users can visually see the conclusions of each node as well as some essential metadata.

The `local view` provides a comprehensive view of proof information available mainly at the level of a single proof step. For example, in Figure 4, one of the intermediate conclusions of the proof is that the butler hates himself ("hates(butler,butler"). The conclusion itself is encoded in `TPTP-CNF` language, and the view shows how the conclusion was derived: `SNARK 20070805r043` was the theorem prover responsible for deriving the conclusion by applying the rule `SNARK HYPERRESOLVE` to the antecedents also listed in the view. One of the main benefits of the global view is that it provides a good insight about the structure of the proof. For example, for the intermediate conclusion we can see that it was derived from three antecedents and that one of antecedents was itself derived from other statements. Further, the edge leaving the intermediate conclusion is evidence that it is not final (i.e., the intermediate conclusion is not an answer for the problem being solved by the inference engine.

The local view is structured to be a textual description of the main properties

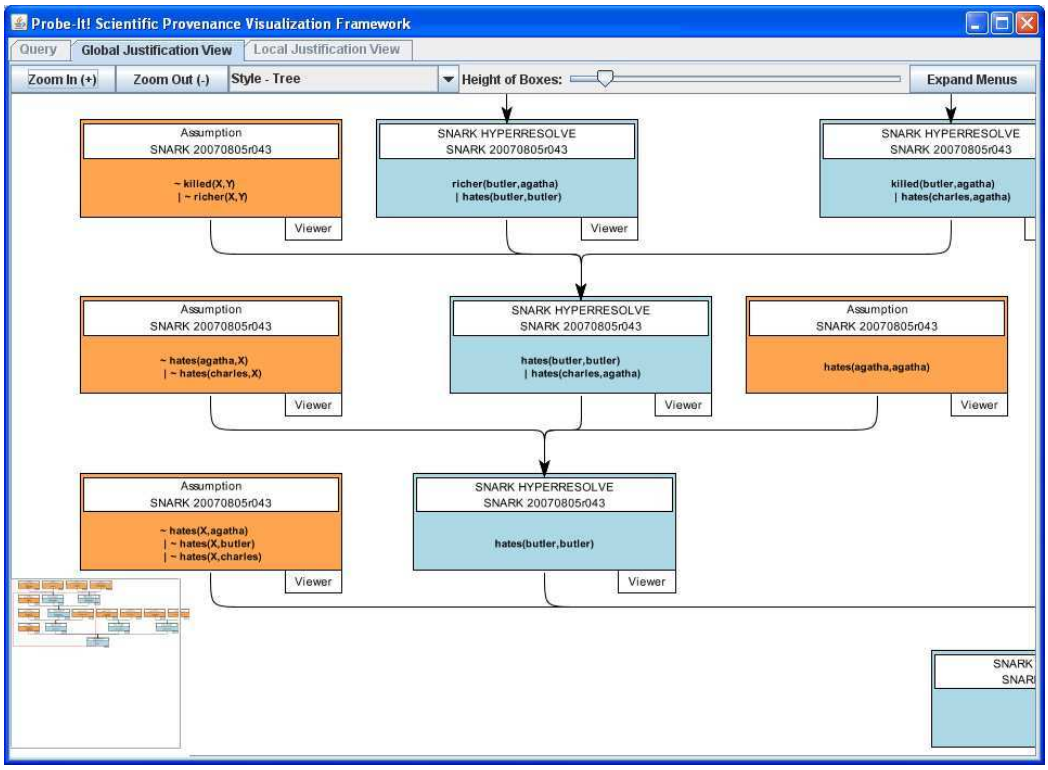


Fig. 3. ProbeIt! Global View

of a single proof step. This description is divided into four sections, as we can see in Figure 4: *conclusion*, *how*, *why*, and *to answer*. The conclusion section shows the main result of the selected inference step along with meta-information about the result. The how section identifies the antecedents as the inference rule applied to theses antecedents to infer the conclusion of the inference step. The why section shows the final conclusion of the entire proof and intermediate goals. The why section also identifies the following conclusion inferred from the conclusion of the current step of the proof. Last, the to answer section shows the question that the theorem prover is answering.

One very important aspect of the local view is that it provides information about sources and some usage information e.g., access time, during the execution of an application or workflow. Every node in the justification DAG has an associated provenance description. This information, usually textual, is accessible by selecting any of the aforementioned nodes. For example, upon selecting the "SNARK 20070805r043" hyper-link in the local view in Figure 4, meta-information about the inference engine, such as the responsible organization, is displayed in another panel. Similarly, users can access information transformation nodes, and view information about used algorithms. It is important to note that the requested meta-data is exactly the same information already presented in Figure 2. This exemplifies a case where user interface software can be reuse by multiple tools on the same way that the tools reuse meta-data to encode portable proofs.

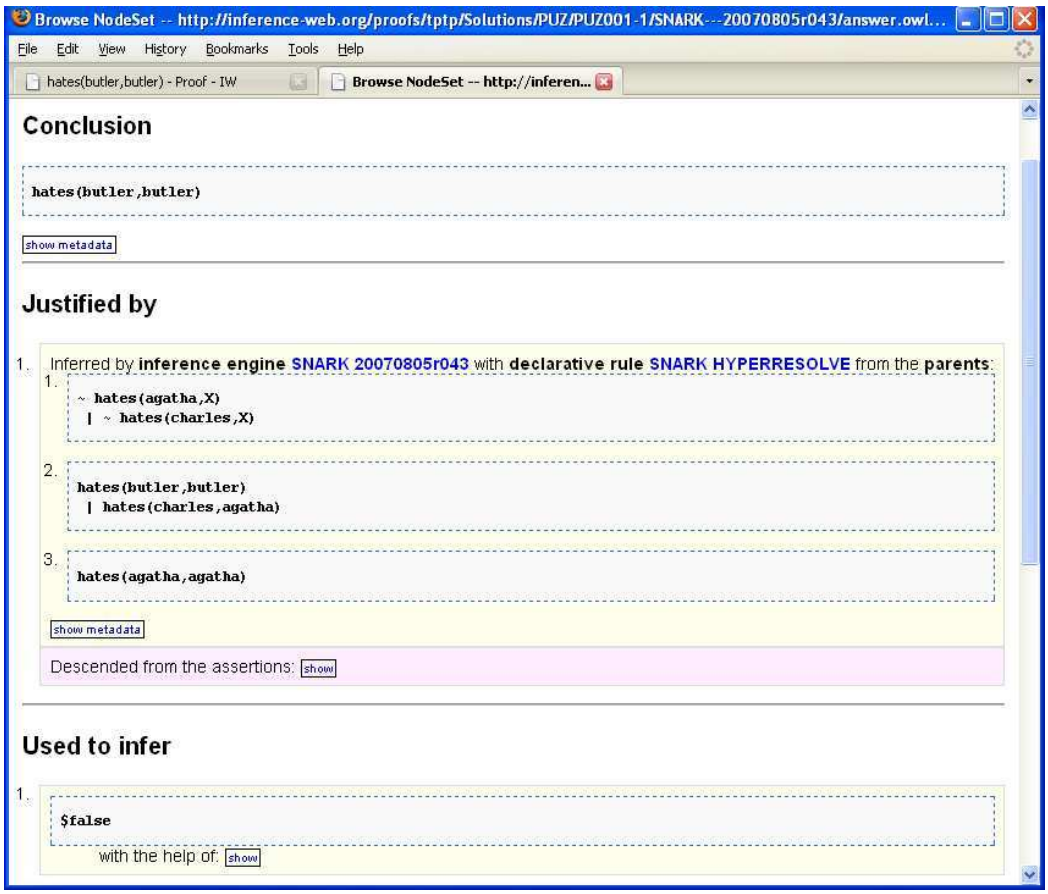


Fig. 4. ProbeIt! Local View

7 Implementation and Deployment

The core functionality provided by Probe-It! can be divided into three main subsystems: the PML API, the DIVA framework, and the visualization framework, which parse PML documents, provide a graphical framework from which execution traces can be rendered, and render the node set conclusions respectively. Both the PML API and the Diva framework are implemented in Java, while some viewers contained in the visualization framework require native libraries. XMDV, for example, is supported by OpenGL and both the 2D plot viewer and grid image viewer are based on native Generic Mapping Tools (GMT) scripts. Both the OpenGL and GMT libraries are implemented as Window's dynamic link libraries (DLLs). Although equivalent libraries for Linux and Macintosh exist, in the interest of time, only a Windows version was considered. The challenge of configuring Probe-It! to be compatible across all platforms will always exist because many of the popular viewers are pre-compiled commercial applications, that cannot be modified; instead, the current practice is to *wrap* these applications inside a Probe-It! by calling them from within Java.

Although Probe-It! contains a small set of pre-configured viewers, it is anticipated that Probe-It! will become more of a framework, from which scientists can subscribe existing viewers, thus difficulties with adapting Probe-It! to run on any

OS greatly restrict the portability of Probe-It!; we are in the process of implementing Probe-It! as a Web application.

8 Summary

Inference Web provides an explanation infrastructure for many types of distributed question answering systems, including theorem provers. It uses a proof interlingua called the Proof Markup Language as an explanation interchange language. It provides a collection of applications to handle proofs distributed on the web. Some of these applications are interactive tools that enable users to better visualize and thus understand portable proofs. In this paper, we provided a theorem prover style use case chosen from the TPTP library. We showed how the IWSearch tool may be used to find proofs with particular properties and provided an example from TPTP. We also described a use of ProbeIt for browsing portable proofs. ProbeIt allows theorem prover developers and users to visually inspect the structure of proofs (with the help of the global view) and the details of each node of a proofs (with the help of the local view).

The Inference Web infrastructure and framework is not restricted to a fixed number of tools to support a given functionality. For example, in terms of interactive tools in support of portable proof browsing, the Inference Web provides the following tools in addition to ProbeIt, as discussed in [8]: the original IWBrowser for browsing PML proofs and proof fragments with the help of standard HTML browsing capabilities and the NodeSet browser that has been integrated into ProbeIt in replacement to its original local view.

Acknowledgments

This work was supported in part by NSF grant HRD-0734825 and by DHS grant 2008-ST-062-000007.

References

[1] Del Rio, N. and P. Pinheiro da Silva, *Probe-it! visualization support for provenance*, in: *Proceedings of the Second International Symposium on Visual Computing (ISVC 2)* (2007), pp. 732–741.

[2] Ding, L., T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi and J. Sachs, *Swoogle: A search and metadata engine for the semantic web*, in: *Proceedings of the 13th CIKM*, 2004.

[3] Kohlhase, M., "An Open Markup Format for Mathematical Documents (Version 1.2)," Number 4180 in Lecture Notes in Artificial Intelligence, Springer Verlag, 2006.

[4] McGuinness, D., L. Ding, P. Pinheiro da Silva and C. Chang, *PML2: A Modular Explanation Interlingua*, in: *Proceedings of the AAAI 2007 Workshop on Explanation-aware Computing*, Vancouver, British Columbia, Canada, 2007.
URL http://www.ksl.stanford.edu/KSL_Abstracts/KSL-07-07.html

[5] McGuinness, D. L. and P. Pinheiro da Silva, *Explaining Answers from the Semantic Web*, Journal of Web Semantics **1** (2004), pp. 397–413.

[6] Murdock, J. W., D. L. McGuinness, P. Pinheiro da Silva, C. Welty and D. Ferrucci, *Explaining Conclusions from Diverse Knowledge Sources*, in: *Proceedings of the 5th International Semantic Web Conference (ISWC2006)* (2006), pp. 861–872.

[7] Pinheiro da Silva, P., D. L. McGuinness and R. Fikes, *A Proof Markup Language for Semantic Web Services*, Information Systems **31** (2006), pp. 381–395.

[8] Pinheiro da Silva, P., G. Sutcliffe, C. Chang, L. Ding, N. Del Rio and D. McGuinness, *Presenting TSTP Proofs with Inference Web Tools*, in: R. Schmidt, B. Konev and S. Schulz, editors, *Proceedings of the Workshop on Practical Aspects of Automated Reasoning, 4th International Joint Conference on Automated Reasoning*, Sydney, Australia, 2008, p. Accepted.

[9] Reynolds, J., *Unpublished seminar notes* (1966), stanford University, Stanford, CA.

[10] Stickel, M., *SNARK - SRI's New Automated Reasoning Kit*, http://www.ai.sri.com/ stickel/snark.html.

[11] Sutcliffe, G. and C. Suttner, *The TPTP Problem Library: CNF Release v1.2.1.*, Journal of Automated Reasoning **21** (1998), pp. 177–203.

Talk Abstract:

Aspects of ACL2 User Interaction

Matt Kaufmann[1,2,3]

Dept. of Computer Sciences
University of Texas
Austin, Texas 78712 USA

Users of the ACL2 theorem prover typically interact with the system in many ways beyond submitting definitions and proving theorems. This talk will show some examples to provide a sense of the diversity of ACL2 user interaction. A few more in-depth examples may be found in the TPHOLS 2008 ACL2 tutorial.

Certainly we will consider the narrow sense of "user interface" as control of input and output. Users traditionally interact efficiently with ACL2 through Emacs, and we will demonstrate some Emacs customizations provided with the system. More recently, two Eclipse-based interfaces have been developed by other groups for teaching purposes: the ACL2 Sedan (ACL2s) and DrACuLa. Beyond the choice of editor (or terminal) is the issue of how to control ACL2 output, traditionally through generated English commentary. A *proof-tree* display illustrates proof structure and provides help for navigating that commentary. But a recent *gag-mode* enhancement is probably much more effective for debugging failed proof attempts.

Other useful output includes error and warning messages, which often point to user documentation. But ACL2 supports effective user interaction in ways beyond the above notions of input/output control.

- Proof commands include not only definitions and theorems, but also scoping mechanisms. Hints can affect the course of proof attempts, and (less often) are dynamically generated by user programs (roughly in analogy to tactics in LCF-style systems).
- Users can direct how proved theorems are to be stored as rules, by default as (conditional, congruence-based) rewrite rules. Syntactic control mechanisms can

[1] ACL2 is joint work with J Strother Moore, with early contributions from Bob Boyer

[2] This material is based upon work supported by DARPA and the National Science Foundation (NSF) under Grant No. CNS-0429591 and also NSF grant EIA-0303609.

[3] Email: kaufmann@cs.utexas.edu

affect the applicability of rules.

- The system state can be affected by setting various modes. Two examples include a *program mode*, which allows the user to write programs that need not terminate but can be used in macros, and a backchain limit for rewriting.
- Session management commands include undoing, redoing, and querying the state.
- Proof assistance is provided by two break/trace utilities for the rewriter; an interactive goal manager; and a report mechanism that can show "expensive" rules.
- The programming interface provides features to bridge the gap between the user and the computing engine, such as Lisp-independent trace and backtrace utilities and verifiable alternative function bodies. This interface also makes available powerful Lisp features including packages and macros. State and other single-threaded objects are supported efficiently with an applicative semantics. A *guard* mechanism provides a powerful, though less automatic, alternative to types that is separate from the logical content of definitions.
- Users can interactively extend the system's capabilities by providing *books* of logical definitions and theorems, as well as system utilities and even (through *trust tags*) system modifications.
- A *clause-processor* mechanism provides an interface through which the user connects ACL2 with another tool.

A separate question, not addressed in this talk, is how to communicate proof results effectively to non-users. Our focus is on interface support for effective usage.

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

A Brief Overview of the PVS User Interface

Sam Owre[1]

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

An overview of the PVS system is presented from a user interface perspective. We present the interfaces from the PVS Lisp core to Emacs, Tcl/Tk, the Prover, markup languages, and some of the various back-end and front-end systems that have been integrated with PVS.

1 Introduction

PVS is an open source verification system that has been in use since it was first released in 1993. The PVS interface historically was simply Emacs, with the Lisp image comprising most of PVS as a subprocess. This is still the standard way to use PVS, but over the years it has been substantially augmented with browsing tools, enhanced prover interfaces, ground evaluation, graphical displays, and LaTeX, HTML, and XML output. In addition, it has been used as both a back-end and front-end with many systems, and has a ground evaluator that even allows PVS to be used as a scripting language. Figure 1 shows the basic architecture of PVS from a user perspective. The rest of this paper is an overview of some of the aspects of the PVS system, focusing on the user interface.

2 Emacs

The basic User Interface for PVS is Emacs (or XEmacs), an extensible and very flexible editor. The PVS Lisp image runs as a subprocess of Emacs, with PVS Emacs commands translated to forms for the underlying Lisp. For example, a proof is started by placing the cursor on the lemma to be proved, and issuing the `M-x prove` command. This sends the current line and theory information to Lisp, which then locates the internal (typechecked) form of the lemma and starts the proof.

[1] This work was partially supported by NSF CCR-ITR-0325808 and CNS-0823086.

©2008 Published by Elsevier Science B. V.

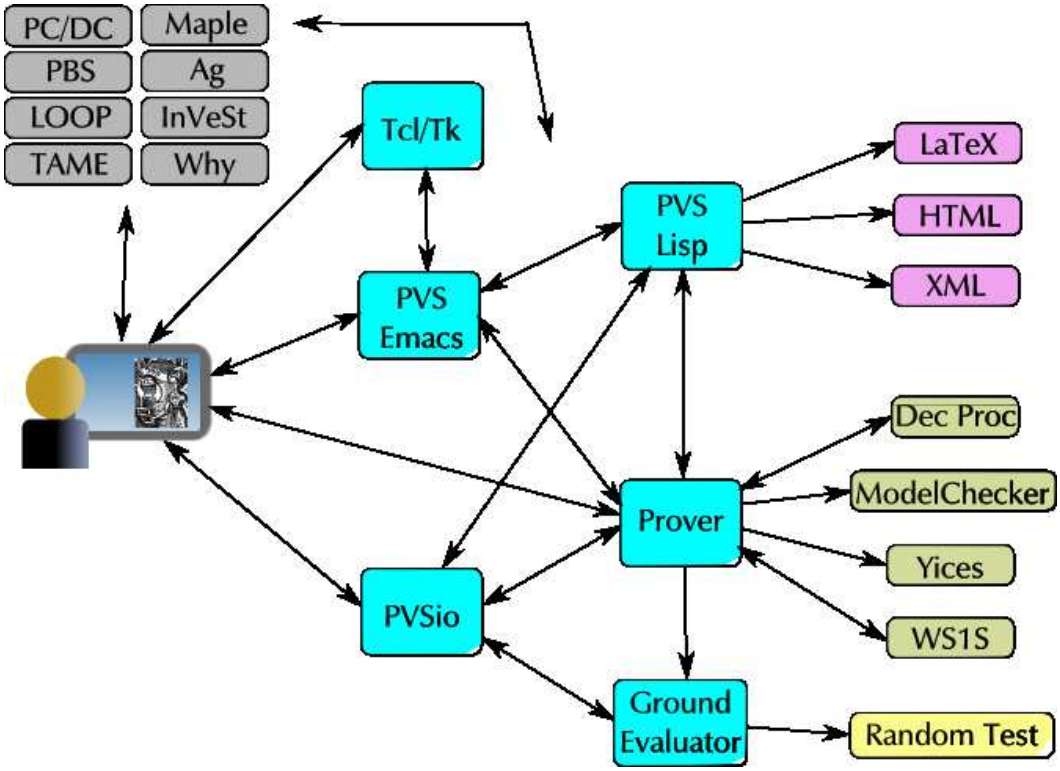


Fig. 1. PVS User Interface Overview

Specifications are edited in a special `pvs-mode` in Emacs, which in addition to the usual keyword highlighting, provides numerous functions, all of which are available from the PVS menu.

The interface is built on a modified version of ILISP [12], allowing the same interface to be used both for developing the PVS system and for creating PVS specifications. In fact, as it is just an extended version of Emacs, PVS may be used to undertake any task normally done using Emacs. PVS Lisp makes requests of Emacs by means of specially formatted strings, that are recognized by the output filter associated with the PVS Lisp subprocess. For example, by this means PVS Lisp can create a buffer and have it displayed in Emacs.

3 Tcl/Tk Interface

The Tcl/Tk interface provides some graphical interface, in particular, it allows proof trees and theory hierarchies to be viewed and manipulated. This is especially useful for large proofs or specifications. The displays are mouse-sensitive; clicking on a theory name in the theory hierarchy will display the corresponding theory specification in an Emacs buffer, and clicking on a sequent symbol in the proof tree window pops up a Tck/Tk window showing the full sequent at that point in the proof tree.

Tcl/Tk is invoked as a subprocess of PVS, and strings are passed from the PVS Lisp process to Emacs, which passes them on to Tcl/Tk. This works in both directions. Unfortunately, the interface is slow, inflexible, and buggy. In particular, there is a bug that seems to be due to a difficult to track race condition that happens when rerunning a large proof as the Tcl/Tk window tries to keep up. We plan on moving to Gtk in the future, which

can be invoked directly from PVS Lisp as foreign functions. In addition to fixing the race condition, this should make it easier to create more graphical interfaces to PVS, as process of going from Lisp to Emacs to Tcl/Tk and back again is unwieldy.

4 Prover interaction

The PVS prover is interactive; starting form a goal sequent, the user constructs a proof tree using available prover commands. The prover provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. These proof commands can be combined to form proof strategies. To make proofs easier to debug, the PVS proof checker permits proof steps to be undone, and checkpointed, and allows the specification to be modified during the course of a proof. After modification, the prover offers to rerun the proof to see that it is still valid. It marks all formulas whose proofs depend on the modified declaration as unchecked. To support proof maintenance, PVS allows proofs (and partial proofs) to be edited and rerun, and allows for multiple proofs to be associated with a formula. Currently, the proofs generated by PVS can be made presentable but they still fall short of being humanly readable.

New strategies and rules may be defined as described in [18], using the `defstep` and `addrule` functions, which may be added to an automatically loaded PVS strategies file. Typically only `defstep` is used to define new user strategies in terms of existing ones. In this way, strategies are built up from primitive rules, and only they need to be trusted. However, some extensions require the addition of new rules, which must be done carefully as soundness may be compromised.

Note that although the strategy language allows arbitrary calls to Lisp, the proofs may be rerun in a mode in which all strategies have been expanded to their primitive rules, in which the Lisp calls are no longer made. In this way the soundness of PVS relies only on the primitive rules and the core execution engine.

5 Generating Latex, HTML, and XML

PVS specifications are in ASCII, which is fine for developing specifications and proofs, but it is often desirable to present them differently. Toward this end, PVS includes facilities for generating LaTeX, HTML, and XML output. The LaTeX output can be generated for specifications or proofs, and the user has control over the mapping from PVS identifiers and operators to LaTeX. The HTML and XML are similar, but only available for specifications. The HTML interface does provide links that lead from a symbol to its corresponding declaration.

The XML output provides much more than the LaTeX and HTML output, as it is a complete representation of the internal typechecked form of PVS entities. This makes it easy to map from PVS to other systems, which is very difficult to do directly from PVS specifications and proofs. Not only is the PVS grammar difficult to parse, but the overloading and automatic conversions allowed by PVS makes it impossible to know how to interpret a concrete expression without typechecking it. The XML form solves this, as it directly represents the parse tree, and provides full resolutions for each identifier. The XML representation includes enough information that the original concrete syntax may be generated, and we have generated an XSLT script that does this.

6 PVS as a Back-end

It is often desirable to have the PVS typechecking and theorem proving available at the back-end of a system. This can easily be done by invoking PVS in *raw* mode, which runs it without the Emacs interface. In this mode it waits for Lisp input, and returns the results, exactly in the way it does with Emacs. There are several functions (e.g., `typecheck-formula-decl` and `prove-formula-decl`) that provide support for proving individual formulas, without generating a full theory. Several existing systems have used PVS as a back-end typechecker and/or theorem prover. Skakkebæk [21] made a deep embedding of the Duration Calculus in his PC/DC system.

César Muñoz implemented a shallow embedding of the B-method [1] into a front-end for PVS called PBS [14]. The B-method is a state-oriented formal method for software development that provides a uniform language, the abstract machine notation, to specify, design, and implement systems. The method is founded on set theory with a first-order predicate calculus, which is embedded into the higher-order logic of PVS.

The LOOP project [22, 6] has developed a tool for specifying and verifying properties of Java programs, using PVS as a back-end. It represents Java objects as coalgebras, and has been used to prove properties of some Java libraries, as well as proving properties of smartcards, as part of the Verificard project.

TAME (Timed Automata Modeling Environment) [4, 3] is a system for specifying several classes of automata, providing templates, a set of auxiliary theories, and specialized prover strategies for specifying and proving properties of automata models.

An interface between the Maple computer algebra system and the PVS theorem prover was implemented [2]. The interface allows Maple users access to the robust and strongly typechecked proof environment of PVS. The environment was extended by a library of proof strategies for use in real analysis. This provides both strong typechecking and theorem proving capabilities to Maple users.

Carlos Pómbo [19], used PVS to provide the semantics of $\mathbf{A_g}$ specifications, defining the semantics of First Order Dynamic Logic and Fork Algebras, along with rules and strategies that allow a user to reason in $\mathbf{A_g}$. Here conversions were defined, such as a meaning function, and arguments such as the current world of the Kripke structure, that by default are included in the prover interaction, but add clutter to the proof. In this case the function for pretty-printing applications was modified in order to suppress the meaning function and the world argument.

There are many other systems that use PVS as a back-end, including Pamela [7], InVeSt [5], the Java Interactive Verification Environment (JIVE) [13], TRIO [10], SOCOS [11], and Why [9]. This is just a partial list.

7 PVS as a Front-end

PVS has also been used as a front-end to several systems. Generally this involves creating a proof rule that interacts with the specified system. This interaction can be through a shell, or directly via foreign function calls. The usual method is to define supporting theories in PVS, define a translation from these theories to the target system, and to define a rule that performs the translation and invokes the system. If the system is intended to return more than simply true or false, a translator must also be provided to convert the results into a

valid PVS sequent. Note that, in general, the soundness of the resulting proof depends on the soundness of the underlying system.

For decision procedures, a special interface was created making it easy to implement new decision procedures. This was used to integrate ICS in earlier versions of PVS.

The built-in PVS model checker [20] is an example of this, in which the model checker only returns true, finishing the proof of this sequent, or unknown with an explanation, leaving the sequent untouched. The model checker relies on the mu-calculus, and theories to support this were provided in the PVS prelude.

The Mona WS1S system was integrated into PVS [17] in the same way. Yices was recently integrated as well, as an end-game prover. This greatly speeds up many kinds of proofs.

PVS may also be used for programming, by using the ground evaluator to translate specifications to Lisp or the Clean functional programming language (see the description at http://clean.cs.ru.nl/). This opens up many possibilities. Using semantic attachments, one can evaluate, test, and animate specifications [8]. The PVS random tester [16] builds on the ground evaluator, and allows specifications to be randomly tested, which is often useful for detecting bugs in specifications before attempting difficult proofs.

César Muñoz developed PVSio [15] an extension of the ground evaluator that makes it simple to define new attachments, use the ground evaluator during proof, and even create PVS scripts that may be used from the command line as with any other scripting language.

8 Proof discovery and maintenance

PVS has limited capabilities for browsing formulas and proofs, and copying proofs from one formula to another. Proof trees may be displayed, and proofs may be single-stepped and check-pointed. Declarations may be modified and added during a proof.

Proof discovery and maintenance is a wide open area of research. Much more is needed, for example, it should be possible to match the current sequent to formulas in the prelude or existing libraries and list the ones likely to be useful. This is quite difficult for several reasons: the libraries might not be referenced, and may only be available remotely, the matching formulas may be useless because some precondition is false, or because an inequality is in the wrong direction. Formulas might not be considered because theories were developed with different names, though they are actually relating to the same entities - for example groups could be defined in one theory using $\star$ and using $+$ in another, thus rendering syntactic matches useless.

9 Conclusion

PVS has a rich user interface, which we have outlined here. It continues to grow, and new paradigms are being explored. In our view, PVS may be treated as a tool bus, allowing exploration of interfaces between often disparate tools. The system is open source, and we encourage any and all additions to the system. More information, and instructions for obtaining and installing PVS are available at http://pvs.csl.sri.com.

References

[1] J.-R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, Volume 552 of Springer-Verlag *Lecture Notes in Computer Science*, pages 398–405, Noordwijkerhout, The Netherlands, October 1991. Volume 2: Tutorials.

[2] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, Volume 2152 of Springer-Verlag *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001.

[3] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *Proceedings of FMSP '00: The Third Workshop on Formal Methods in Software Practice*, pages 25–36, Association for Computing Machinery, Portland, OR, August 2000.

[4] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, July 1998. Informal proceedings available at http://www.win.tue.nl/cs/ipa/uitp/proceedings.html.

[5] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, pages 505–510, Vancouver, Canada, June 1998.

[6] C.-B. Breunesse, N. Cataño, M. Huisman, and B. P. F. Jacobs. Formal methods for smart cards: An experience report. *Science of Computer Programming*, 55(1–3):53–80, March 2005.

[7] Bettina Buth. PAMELA + PVS. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, Volume 1349 of Springer-Verlag *Lecture Notes in Computer Science*, pages 560–562, Sydney, Australia, December 1997.

[8] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available from http://www.csl.sri.com/users/rushby/abstracts/attachments.

[9] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification (tool paper). In *Computer Aided Verification*. pages 173–177, 2007.

[10] Carlo A. Furia, Matteo Rossi, Dino M, and Angelo Morzenti. Automated compositional proofs for real-time systems. *Theoretical Computer Science*, pages 326–340, 2006.

[11] Ralph johan Back, Johannes Eriksson, and Magnus Myreen. Verifying invariant based programs in the SOCOS environment. In *In Teaching Formal Methods: Practice and Experience (BCS Electronic Workshops in Computing). BCS-FACS*, 2006.

[12] Todd Kaufmann, Chris McConnell, Ivan Vazquez, Marco Antoniotti, Rick Campbell, and Paolo Amoroso. *ILISP User Manual*, 2002. Available with at http://sourceforge.net/projects/ilisp/.

[13] J. Meyer, P. Müller, and A. Poetzsch-Heffter. *The JIVE System Implementation Description*, 2000. Available at http://softech.informatik.uni-kl.de/old/en/publications/jive.html.

[14] César Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-1, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1999.

[15] César Muñoz. *Rapid Prototyping in PVS*. National Institute of Aerospace, Hampton, VA, 2003. Available from http://research.nianet.org/~munoz/PVSio/.

[16] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods (AFM)*, Seattle, WA, August 2006. Available at http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf.

[17] Sam Owre and Harald Rueß. Integrating WS1S with PVS. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, Volume 1855 of Springer-Verlag *Lecture Notes in Computer Science*, pages 548–551, Chicago, IL, July 2000.

[18] Sam Owre and N. Shankar. Writing PVS proof strategies. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, pages 1–15, Hampton, VA, September 2003. Available at http://research.nianet.org/fm-at-nia/STRATA2003/.

[19] Carlos López Pombo, Sam Owre, and Natarajan Shankar. A semantic embedding of the Ag dynamic logic in PVS. Technical Report SRI-CSL-02-04, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2004. Available at http://pvs.csl.sri.com/papers/AgExample/.

[20] N. Shankar. PVS: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, Volume 1166 of Springer-Verlag *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996.

[21] Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[22] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, Volume 2031 of Springer-Verlag *Lecture Notes in Computer Science*, pages 299–312, Genova, Italy, April 2001.