

# Engineering the Prover Interface

Holger Gast

*Wilhelm-Schickard Institut für Informatik  
Universität Tübingen  
gast@informatik.uni-tuebingen.de*

---

## Abstract

Practical prover interfaces are sizeable pieces of software, whose construction and maintenance requires an extensive amount of effort and resources. This paper addresses the engineering aspects of such developments. Using non-functional properties as quality attributes for software, we discuss which properties are particularly relevant to prover interfaces and demonstrate, by the example of the I<sup>3</sup>P interface for Isabelle, how judicious architectural and design decisions lead to an interface software possessing these properties. By a comparison with other proposed interfaces, we argue that our considerations can be applied beyond the example project.

*Keywords:* prover interfaces, software engineering, software quality

---

## 1 Introduction

The discussion of prover interfaces in the past has focused primarily on the functionality available to the user. A convenient management of proof scripts [11,8,1,3] is the current basis for using interactive provers like Isabelle and Coq. In this setting, the prover may support the user in writing the proof scripts [5], or check proof documents that are close to mathematical texts [16]. Proof-by-pointing [10,9,31], graphical proving metaphors [27], or prover-specific interaction models [23] promise improved user experience. Cooperative proof development is enabled by web-interfaces for provers [26]. Integrated verification environments offer prover interfaces for a special application [24,13,15].

Despite presenting medium to large software systems, the cited studies surprisingly have neglected the engineering considerations necessary to build these systems. Instead, they focus on the broad software structure: brokers are used to decouple interface components from prover components [4,32]. The construction of web-interfaces focuses on technical aspects of client/server

*This paper is electronically published in  
Electronic Notes in Theoretical Computer Science  
URL: [www.elsevier.nl/locate/entcs](http://www.elsevier.nl/locate/entcs)*

communication [26]. The proof-by-pointing interfaces exhibit the interaction between prover and interface, but not the implementation of the UI [10,27]. The generic interface [24] describes a concrete Java interface for prover plugins, but elides the main effort, the construction of the Eclipse plugin.

Although one can argue that this high-level mode of presentation is necessary for conciseness, it still leaves open a central question in the study of user interfaces for theorem provers: given a desirable functionality, how can one craft a software system that will deliver this functionality reliably and with as little programming and maintenance effort as possible? One indication that this question merits, indeed, more consideration, is that most of the cited systems exist only as prototypes, or have not been maintained for a long time.

This paper addresses the identified question by applying standard approaches for the construction of user interfaces to the particular domain of theorem provers. We base the discussion of engineering considerations on desirable non-functional properties of a prover interface software [6] (Section 2). The concrete example of I<sup>3</sup>P, the Interactive Interface for the Isabelle Prover ([21]; Section 3), is then used to demonstrate the realization of the engineering goals (Section 4). Throughout, we emphasize the relevance of the general principles for theorem prover interfaces. The discussion of related work (Section 5) then applies our reasoning to existing systems.

## 2 Desirable Non-functional Properties

There are always two possible views on a software system: its users are mainly interested in the available functionality, and the support they get in proceeding with their work; its developers, maintainers, financers, and project managers necessarily place strong requirements on the quality of the software, which can be captured by using *non-functional properties* [6] as quality attributes. This section discusses those properties most pertinent to the prover interfaces and thus sets the goals for the engineering considerations presented subsequently.

**Changeability** Software systems need to change over time, whenever their environment or user demands change. Whether changes are accommodated gracefully or disrupt the entire software structure by requiring many small changes (“shotgun surgery” [17]) is therefore a crucial quality criterion. In the case of prover interfaces, this necessity is particularly pronounced: since provers are research tools, they tend to adopt the latest developments very rapidly, often sacrificing backward compatibility by necessity. Consequently, when the prover changes, the interface is likely to change as well.

**Maintainability** A major portion of the total effort (and cost) of software development is spent on the maintenance of released systems. Although maintainability is related to changeability, its emphasis is on keeping the

existing system stable, rather than changing it to meet particular demands. For prover interfaces, maintainability is important because any resources spent on the interface are lost for the development of the prover itself.

**Traceability** Traceability expresses that the functional requirements of the software can be traced, through the architecture and design, to the concrete implementation. It is therefore a prerequisite to maintainability and changeability, because it clarifies which parts need to be modified to achieve a particular effect. Indeed, experience with the Proof General/Emacs code base, which suffers from the properties of Elisp, suggests that this is one of the biggest obstacles to maintaining a running system [33].

**Testability** Agile software processes [7,28] have propagated the insight that programmers will only be confident about making necessary changes if they can be sure not to break existing functionality. This can be ensured by automatic tests which are run frequently during development. However, software must be written to be testable: it must be possible to access single components in isolation to identify the source of failures easily. Since rapid changes in the prover induce rapid changes in its interface, testability is a crucial necessity. Furthermore, a prover user will not tolerate introduced failures that occur only when a particular function is requested — they are not interested in the interface, but only in the prover itself.

**Re-usability** The development of a prover interface requires substantial efforts. It is therefore desirable that it can be re-used in different contexts, for instance in integrated verification environments [24,13,15].

**Extensibility** A software is extensible if it provides well-defined mechanisms for integrating new functionality without modifying the existing code base. The user interface for a prover, in particular, must be extensible to reflect newly developed prover features to achieve a maximum benefit for the user.

It is important to note that these properties are not goals in themselves, which are prescribed by some abstract software design regime. The real goals have been given by identifying their relevance for prover interfaces. The non-functional properties serve as stepping stones in achieving these goals, since the software engineering literature provides techniques to achieve the properties, while not giving any recommendations for prover interfaces. The transfer of the general principles to the specific application is a main point of this paper.

We also note that the property of usability is omitted from the list on purpose, since it is complementary to those included: it concerns the outer appearance of the software towards the user, whereas the above properties refer to the inner software structure and quality. While usability is an important aspect of any interface software, and has correspondingly been treated widely in the literature on user interfaces for provers, this paper focuses on the internal quality attributes.

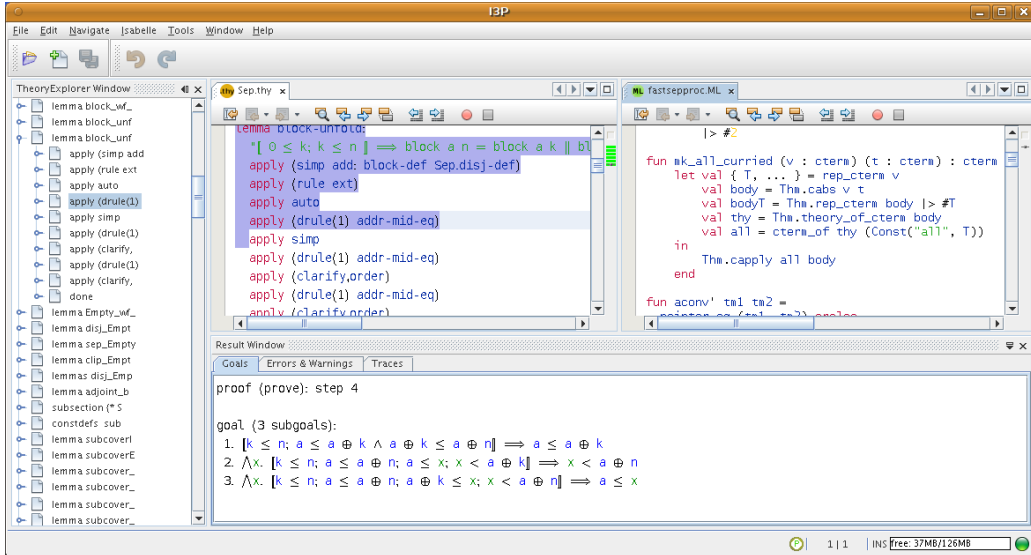


Fig. 1. Screenshot of Running System

### 3 System Description

This section gives an overview of the functionality of I<sup>3</sup>P from the user’s perspective. I<sup>3</sup>P offers the main components known from the current standard interface, Proof General/Emacs (see Figure 1): editors for theories and ML files, a viewer for processing results, and a tree-structured overview over the currently selected theory. We will now briefly summarize their functionality.

The *theory editor* component lets the user edit the text of proof documents. It offers an input facility for mathematical symbols that emulates the Emacs XSymbols mode. Files are saved as usual with encoded XSymbol names. Automatic indentation follows the nesting level of proofs. The theory editor provides syntax highlighting and also handles dynamic definitions of new Isar keywords. The syntax highlighting and automatic indentation also recognize embedded L<sup>A</sup>T<sub>E</sub>X and ML sources. Processed commands are locked and highlighted as expected, and files loaded by the prover are locked as well.

The *theory explorer* component arranges the commands in the current theory according to the nested Isar proof structure. It offers a context menu for direct manipulation [30], e.g. execution or undoing commands, or jumping to a command in an editor.

The *result viewer* lets the user examine the prover messages stored in the state of single commands (Section 4.5). The display is split into normal output, errors and warnings, and trace messages. Since messages for previously processed commands are kept, the user can go back in the proof script.

The *SML editor* offers syntax highlighting and automatic indentation. Again, files loaded by the prover are highlighted as “locked” and the user cannot edit them.

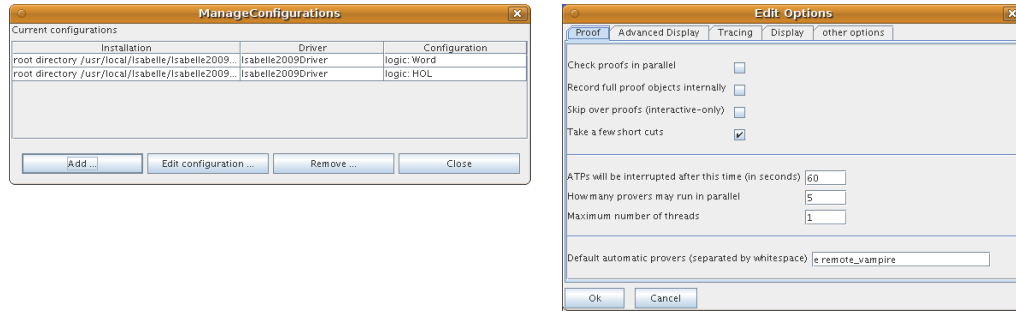


Fig. 2. Editing Dialogs for Configurations and Options in I<sup>3</sup>P

I<sup>3</sup>P enables the user to manage different installed provers and startup options (Figure 2). Each configuration includes an installation, the chosen prover driver (Section 4.6) and the startup options, which is the logic image in the case of Isabelle. A wizard allows the user to define new configurations by selecting a prover installation, the driver to be applied, and the startup options to be given. The wizard is generic such that the individual prover driver can determine which startup options are available.

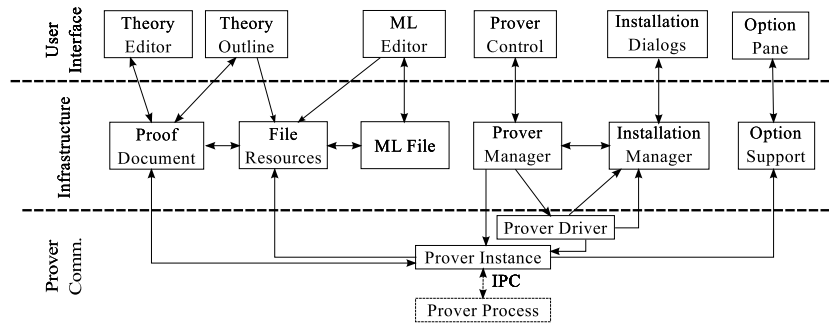
I<sup>3</sup>P provides a dialog for editing Isabelle’s run-time options (Figure 2), and stores the settings between sessions. Furthermore, option settings can be attached to single commands as *command-local options* and will then be enabled only during the execution of these particular commands. Such a feature is desirable, in particular, for tracing the behaviour of tools such as Isabelle’s simplifier in situations where they do not perform as expected.

## 4 Architecture

This section presents the architecture of I<sup>3</sup>P, which has been developed to achieve the desirable non-functional properties identified in Section 2. In this way, it explains how engineering considerations have led to the overall system presented in Section 3. Each of the subsections describes a particular design decision or component, and then proceeds to discuss the resulting non-functional properties. For the property of traceability, we remark in advance that the components, type-set in italics, correspond to concrete classes of the implementation. Further structure is omitted for brevity here, but is provided in the JavaDoc comments of the available source code.

### 4.1 Overview

Figure 3 gives an overview over the main components of I<sup>3</sup>P. The system is divided into three layers. At the top, the user interface layer contains the actual widgets from Section 3 that the user interacts with.

Fig. 3. Overview of the I<sup>3</sup>P Architecture

The infrastructure layer contains the entire functionality of I<sup>3</sup>P: it stores *proof documents* and ML files, as well as the different installations and runtime options. The *prover manager* controls the life-cycle of the prover, i.e. it starts and stops the prover as requested. The *file resources* track the files loaded by the prover to enable the UI layer to prevent modifications if necessary.

The prover communication layer encapsulates the access to the prover process. A *prover driver* enables the prover manager to start a process for a specific installation and set of options. The driver also lets the installation manager examine an installation, for instance to list the available logics. When a prover is started, the *prover instance* hides the inter-process communication, in particular the physical protocol in use.

#### 4.2 Document View Separation

The overview in Section 4.1 highlights a central feature of the I<sup>3</sup>P architecture: the strict division between the infrastructure layer and the user interface layer. We will discuss this separation and its consequences for the resulting non-functional properties before proceeding further, because it influences many later design decisions.

The decision is based on the DOCUMENT-VIEW variant of the MODEL-VIEW-CONTROLLER pattern [14]. The “document” in this pattern contains the data structures, algorithms, and business logic of the application. The “view” is a thin UI component that merely enables the user to access this functionality. In the other direction, the “document” does not contain any code that relates to the “view”.

The resulting division between infrastructure and UI layer is known to yield several of the desirable non-functional properties [14]. First, testability is much enhanced, because the core logic of the application is contained in ordinary, non-UI classes which can be accessed by automated unit tests. Furthermore, test-driven development (TDD; [7]) is enabled because the infrastructure’s functionality can be specified as a standalone product. As others before, we have found TDD to increase productivity very much.

Testability has enabled us to write around 350 unit tests for different parts of I<sup>3</sup>P. With a large test coverage, the implication of testability is changeability (Section 2): we can be confident about making changes because any introduced failures will be spotted early, the next time that the automated tests are run. In particular, re-factoring the software [17] tends to clarify the overall structure and enhances, in turn, traceability and maintainability.

The document-view separation yields yet another benefit for changeability: since the main functionality is independent of the UI, it becomes simple to adapt the UI to the user’s expectations. Note that this possibility crucially relies on the fact that tests target the infrastructure layer, not the UI layer.

Finally, the separation enables re-usability and portability, because the main part of the implementation remains independent of the concrete user interface. For instance, the I<sup>3</sup>P infrastructure has been successfully re-used to obtain an Eclipse-based Isabelle interface [22], which is desirable from the perspective of integrated verification environments [24,13,15].

This list of non-functional properties shows clearly that a strict document-view separation is crucial, since it already achieves many of the goals set in Section 2. Furthermore, we have found that the overhead in development time introduced by the separation is outweighed by far by the entailed benefits.

### 4.3 *Event-based Design*

A second principle adhered to throughout the design of I<sup>3</sup>P is the reliance on events. All components in Figure 3 offer generic notification mechanisms that signal changes in their state (see OBSERVER [18]): the file resources fire events for newly opened, loaded, and closed files, the prover manager offers notifications about the startup and shutdown of the prover, the proof documents have a detailed model of user- and prover access (Section 4.5), and so on. The central guideline in this context is to have components define their supported events in terms of their own specification, without considering possible recipients of the notification.

Event-based design yields several benefits: the software becomes testable, because the components defining events can be used independently of the context receiving events in the application. In the other direction, the recipients of the events can be tested by generating events directly in unit tests. Testability then enables changeability and maintainability, as before.

A further contribution of event-based design is its role in enabling extensibility, as discussed below.

### 4.4 *Enabling Extensibility*

I<sup>3</sup>P employs the concepts developed for object-oriented frameworks [25] and the INTERCEPTOR pattern [29], which can be seen as a concise summary of

the main points of frameworks, to enable extensibility. In these approaches, a framework is a platform that provides mechanisms that are common for a family of software products. The specific applications are created as extensions to this platform. The interface between the framework and its extensions is defined in an event-based way: the framework specifies some abstract state, such as the file resources or proof documents in Section 4.1. It also defines events as changes in that state, and notifies interested extensions about them.

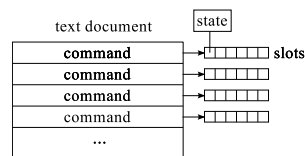
A prerequisite for extensibility is that the programming platform supports some form of run-time-loadable modules which can interact in well-defined ways. The Netbeans platform [12], on which I<sup>3</sup>P builds, offers a particularly lightweight module system. Modules can place ordinary Java objects into a central *system file system*. Other modules can then retrieve these objects from the system file system with a few lines of code and interact with them directly. Providing extension points is therefore as easy as specifying some path in the file system where extension objects must be deposited, and defining an interface expected of these objects. Whenever extension points are mentioned subsequently, this mechanism is employed.

Extensibility itself is thus implemented in a traceable manner: all components in Figure 3 define event models, which enables component-wise extensibility. Beyond that, a *central event dispatcher* retrieves interested event receivers from sub-folders of **Events** in the system file system. It thus reflects the concept of a framework defining a global event model to be used by application-specific extensions.

Finally, we emphasize that also the core functionality of I<sup>3</sup>P, as presented in Section 3, employs these extension mechanisms for the implementation. The mechanisms have therefore been validated in practical scenarios, they are not merely design ideas to be evaluated in later stages of the development.

#### 4.5 Proof Documents

Traceability requires that conceptual elements of the proposed solution map to elements in the design and implementation. A central instance of this principle is the treatment of *proof documents* in I<sup>3</sup>P. The figure below depicts the adopted model: a proof document is an ordinary text document which is partitioned into a sequence of *commands*, such that each command can be sent to the prover separately.

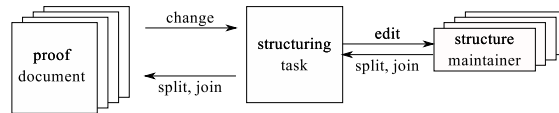


Commands have attached a set of *slots* which associate arbitrary data with commands efficiently. Slots address extensibility in that other modules can

register new types of slots and access them in a type-safe manner. For instance, prover drivers will commonly want to store private information with commands, and the theory explorer (Section 4.1) deposits the document’s tree structure in yet another slot. Slots support event-driven design, thus realizing its described benefits, by a generic *slot changed* event.

One of the slots is the *command state*, which models the processing of a command by the prover. As motivated in [20], the state distinguishes between *idle* commands, which can be edited by the user, and *sent* commands, which have been transmitted to the prover. The processing itself is further divided into sub-states *queued*, *being processed*, and *finished*, which is again split into *successful* and *erroneous*. The command state also contains the sequence of *messages* generated by the prover during processing (see Section 4.6). When the UI wants to execute or undo some command, it is sufficient to set its state to *send* or *revoke* [20], which simplifies the implementation and addition of UI-level functionality.

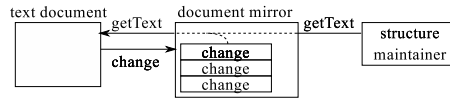
I<sup>3</sup>P defines a consistent, lightweight mechanism for maintaining the partitioning of the document into commands.



The proof documents in the system notify a *structuring task* about occurring textual changes, along with the command in which they occur. The structuring task runs in a separate thread and queues the reported commands. It delegates the processing to a *structure maintainer* associated with each proof document. The maintainer is responsible for updating the command partitioning. Since it can only split and join commands, the commands always form a partitioning. To decide about changes, the structure maintainer can access the entire proof document, for example to handle comments in a single run or to examine adjacent commands.

Executing the structure maintainer in a separate thread allows it to spend any amount of processing time without stalling the user interface. This decision reduces the complexity of the maintainer, compared to the earlier approach [20] of immediate reactions. However, the user and the structure maintainer now access the document concurrently, which requires proper synchronization. I<sup>3</sup>P offers a generic solution through a *document mirror*. The mirror collects the textual edits in the document and translates any positions accessed by the structuring task accordingly. If a clash with an edit is detected,

the maintainer is re-started.<sup>1</sup>



Proof documents are a main source of events in I<sup>3</sup>P: structural modifications and state changes are reported to the registered *proof document listeners*; *structure listeners* and *state listeners* receive the respective subsets.

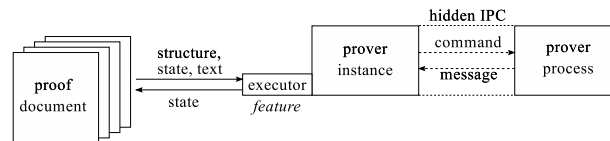
This particular instance of event-based design yields, of course, the general benefits discussed in Section 4.3. Furthermore, the notifications sent by proof documents have proven sufficient to implement the connections to all other components, including the entire UI layer and the prover (Section 4.6). Further, having these events available enables extensibility by UI components that display the state of commands.

#### 4.6 Prover Interaction

A central design consideration is the expected interaction between the prover and the interface, and the relative distribution of functionality between them. One possible view is that the prover can basically execute and undo commands [2,24], and one can parameterize the interface accordingly. Going one step further, one can explore how much useful functionality can be achieved by the interface without support from the prover [19]. These approaches, by intention, do not target further capabilities of the prover.

I<sup>3</sup>P, on the other hand, starts from the premise that the main role of the interface is to make the functionality present in the prover available to the user. I<sup>3</sup>P's mechanisms are therefore very general and make minimal assumptions about the prover, such that the prover driver can employ these mechanisms according to the prover's specific needs.

The central linking point is the prover driver, which creates a prover instance when the prover is started (Section 4.1). The prover instance encapsulates any communication, and in particular the protocol used. This allows the driver to also access special and new functionality in prover-specific ways.



The main, but very weak, assumption that I<sup>3</sup>P makes is that the communication is message-based, i.e. the prover instance sends commands and receives

<sup>1</sup> While the change notifications for the structuring task are sent asynchronously and are queued, those for the document mirror must be sent synchronously, while the editor holds the document lock. Otherwise, the structuring task might obtain the lock and access text at the wrong positions before the mirror has received the update.

*messages* in return. Different types of messages are supported, among them normal output (e.g. for goals), error and warning messages, and debug and tracing messages. These are modelled after Isabelle’s defined output channels. A message consists of the message text and a set of *markups*, which might indicate different types of identifiers. The driver is, of course, free to use internal messages for special functionality without relaying them to I<sup>3</sup>P.

Towards supporting a wide range of functionality, I<sup>3</sup>P defines the concept of prover *features*. A feature is offered as a representative of a specific capability of the prover (see PROXY [18]), which communicates with the prover in the background to access the functionality. Other components in the interface may then rely on different prover capabilities by simply requesting the corresponding feature. We will now use the features offered by the current Isabelle-2009-1 driver for demonstration purposes.

The **Executor** feature represents the execution of commands in proof documents. Besides life-cycle methods to connect and disconnect executor and framework (see COMPONENT CONFIGURATOR [29]), it allows the prover to receive all events related to proof documents (Section 4.5) and files opened in the editor (Section 4.1). The prover instance is free to react in any appropriate way, by changing the commands’ states to indicate processing. Due to the event-based design, the UI layer will react to these state changes by highlighting and locking of commands in the theory editors. The Isabelle-2009-1 driver emulates the traditional linear processing model with multi-theory support.

Complementary to command execution, the **FileReporting** feature offers notifications whenever the prover loads or releases files. This feature is requested by the file resources component (Section 4.1) to reflect the current state on the interface side.

The **StructureMaintainer** feature is a **FACTORY** [18] for the structure maintainers attached to proof documents. As described in Section 4.5, these maintainers receive events about textual changes and re-examine commands to split and join them. The Isabelle-2009-1 driver supports dynamic keyword recognition, thus avoiding the need to maintain external configuration files.

The **ProofHierarchyClassification** feature allows the interface to classify commands into different categories (start of proof, end of proof, top-level, etc.). This is used, for instance, to reconstruct the tree structure for the theory explorer and to indent commands in the theory editor (Section 4.1). The **DynamicKeywords** feature offers access to all currently defined keywords and their classification in isolation. The syntax highlighting for theories (Section 3) uses this feature to adapt the presentation accordingly.

The **RuntimeOptions** feature defines a hierarchical grouping of runtime options and offers proxy objects to set the option values conveniently. It is accessed both by the local options and the options dialog (Section 3).

These features demonstrate how specific prover capabilities can be rep-

resented in the interface without fixing a protocol. It is a cornerstone to extensibility: when Isabelle acquires new functionality that should be accessible to the user, the prover defines and provides a new feature which is then requested by a suitable UI layer extension. At the same time, testability is enhanced, since unit tests can be written for each feature in isolation.

I<sup>3</sup>P is also extensible by new drivers, since the prover manager retrieves installed drivers from the `Drivers` directory of the system file system (Section 4.4). Similarly, the installation manager (Section 3) looks for installation types in the `Installations` directory. Both drivers and installation types can therefore be supplied by modules developed independently of the I<sup>3</sup>P core.

## 5 Related Work

The engineering considerations discussed in this paper — apart from re-usability — have not been addressed explicitly in the literature. Therefore, this section analyzes existing user interfaces from our perspective. Interestingly, we find that many architectural decisions in the related studies could in principle be used to achieve desirable non-functional properties.

The Proof General project [1] aims at re-usability of the prover interface. With support for Isabelle, Coq, LEGO, PhoX and experimental drivers for several others, this goal has been achieved very successfully. The method of adapting the interface is the customization of around 150 variables [2] which characterize the communication with the prover. In comparison, I<sup>3</sup>P does not make assumptions about the precise nature of the prover protocol but allows a prover driver component to encapsulate the communication. Combined with the concept of features (Section 4.6), this approach is more flexible since different ways of prover interaction can be implemented and new functionality besides undo/redo of theory commands can be made available. Furthermore, I<sup>3</sup>P’s prover drivers are testable as standalone components, such that deviations between prover versions can be detected early. It is interesting to note that the effort of writing an I<sup>3</sup>P driver and customizing Proof General are roughly the same ( $\approx 3000$  lines of code) in the case of Isabelle.

The PGKit framework [4] proposes an XML-based communication protocol for provers and display components. A broker component mediates the communication between provers and interfaces, manages the open proof documents, and synchronizes the user edits and the prover execution. This architecture aims at re-usability, as Proof General does, but also addresses extensibility by new display components. However, the protocol does not contain mechanisms for accessing prover-specific functionality. Furthermore, display components in PGKit are heavy-weight in that they need to implement the PGPIP protocol, while I<sup>3</sup>P UI extensions can directly access the objects in the infrastructure layer. In principle, the defined protocol should enable testabil-

ity of single components, and the properties and benefits entailed by it, but this is not discussed in the published work.

Charles and Kiniry [24] address re-usability by providing a minimal interface which can be adapted to different provers by prover-plugins. Their interface/prover communication is limited to traditional undo/redo of commands and in broad terms follows the adaptation model of Proof General/Emacs [2]. There are no mechanisms to expose prover-specific functionality in the interface. The architecture of the Eclipse-based interface itself is not discussed, thus neglecting traceability; if a document-view separation is followed in the implementation, the corresponding benefits could be claimed for the software. While the definition of clear interfaces for prover plugins enables their independent testability, testing is not discussed.

Lüth and Wolff [27] consider re-usability of a truly graphical interface for Isabelle. Their generic implementation [27, §4.2] is given as an SML functor, integrated into the Isabelle heap image, which can be instantiated for different logics and applications. The functor implements a document-view separation (Section 4.2), but the potential benefits of this decision are not discussed.

Kaliszyk [26] proposes to access provers through web-services. While the technical foundations of the employed AJAX and DOM APIs are discussed in detail, it remains unclear whether the concrete protocol is private or fixed and public. In the latter case, testability of the server components would be enabled. Furthermore, the benefits of the document-view separation, which is implicit in employing web-services, could also be realized in this system.

Bertot and Théry [11] describe a generic, thus re-usable approach to building sophisticated graphical user interfaces, including proof-by-pointing mechanisms. Their presentation very clearly exhibits the document-view separation (in particular [11, §2.1, §3.1, §6.4]), and extensibility is identified as a goal, albeit in the restricted form of configuration of generic mechanisms by data [11, §3.2, §4.3].

In summary, the goals for prover interfaces identified in Section 2 have been discussed, though mostly implicitly, in various previous studies. However, the ways to achieve them have always been derived from the specific application, rather than from desirable quality attributes and general software engineering principles, and the potential further benefits of applying these principles have therefore been neglected. As a relative contribution, this paper thus shows how the general principles and strategies from software engineering can be applied to prover interfaces in order to realize these benefits.

## 6 Conclusion

We have analyzed the task of constructing a prover interface from an engineering perspective: starting from a set of goals that prover interfaces must

attain, we have identified relevant quality attributes, or non-functional properties, that the interface software should possess. Among them, maintainability and changeability are perhaps the most pressing concerns for practical developments, while re-usability and extensibility address the question of how the effort once invested can yield the maximum benefit.

We have then demonstrated that these properties can be achieved for a user interface software at the example of I<sup>3</sup>P, a newly developed interface for Isabelle. Based on the principles of document-view separation and event-based design, I<sup>3</sup>P's architecture has been presented to the extent of detail where the desirable non-functional properties become visible.

This paper thus addresses cross-cutting issues that are relevant for the development for any serious prover interface, but which have nevertheless been discussed in the literature only rarely, and mostly implicitly. Our overview of related work shows, however, that a consideration of the engineering questions could lead to an improvement of other interfaces as well.

## References

- [1] Aspinall, D., *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] Aspinall, D. and T. Kleymann, *Adapting Proof General*, <http://proofgeneral.inf.ed.ac.uk/releases/ProofGeneral/doc/PG-adapting.pdf> (2008).
- [3] Aspinall, D., C. Lüth and A. Fayyaz, *Proof General in Eclipse: System and architecture overview*, in: *Eclipse Technology Exchange Workshop at OOPSLA 2006*, 2006.
- [4] Aspinall, D., C. Lüth and D. Winterstein, *A framework for interactive proof*, in: *Calculemus '07 / MKM '07: Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants* (2007), pp. 161–175.
- [5] Aspinall, D., C. Lüth and B. Wolff, *Assisted proof document authoring*, in: *Mathematical Knowledge Management 2005 (MKM '05)*, number 3863 in Springer LNAI, 2005, pp. 65–80.
- [6] Bass, Clements and Kazman, “Software Architecture in Practice,” Addison-Wesley, 2003, 2nd edition.
- [7] Beck, K., “Extreme Programming Explained: Embrace Change,” Addison-Wesley Longman, Amsterdam, 1999.
- [8] Bertot, Y., *The CtCoq system: Design and architecture*, Formal Aspects of Computing **11** (1999), pp. 225–243.
- [9] Bertot, Y., G. Kahn and L. Théry, *Proof by pointing*, in: M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, number 789 in LNCS (1994), pp. 141–160.
- [10] Bertot, Y., T. Kleymann-Schreiber and D. Sequeira, *Implementing proof by pointing without a structure editor*, Technical Report ECS-LFCS-97-368, Department of Computer Science, Edinburgh University (1997).
- [11] Bertot, Y. and L. Théry, *A generic approach to building user interfaces for theorem provers*, J. Symbolic Computation **25** (1998), pp. 161–194.
- [12] Boudreau, T., J. Tulach and G. Wielenga, “Rich Client Programming: Plugging into the Netbeans platform,” Prentice Hall, 2007.

- [13] Burdy, L., A. Requet and J.-L. Lanet, *Java applet correctness: A developer-oriented approach*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *FME*, LNCS **2805** (2003), pp. 422–439.
- [14] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, “Pattern-oriented Software Architecture: A System of Patterns,” Wiley & Sons, 1996.
- [15] Chalin, P., P. R. James and G. Karabotsos, *JML4: Towards an industrial grade IVE for Java and next generation research platform for JML*, in: N. Shankar and J. Woodcock, editors, *VSTTE*, Lecture Notes in Computer Science **5295** (2008), pp. 70–83.
- [16] Dietrich, D., E. Schulz and M. Wagner, *Authoring verified documents by interactive proof construction and verification in text-editors*, in: *Intelligent Computer Mathematics*, LNAI **5144** (2008), pp. 398–414.
- [17] Fowler, M., “Refactoring: Improving the Design of Existing Code,” Addison-Wesley, 2000.
- [18] Gamma, E., R. Helm, R. Johnson and J. Vlissides, “Design Patterns – Elements of Reusable Object-Oriented Software,” Addison-Wesley, 1995.
- [19] Gast, H., *An architecture for extensible Click’n Prove interfaces*, in: K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, 364/07, Department of Computer Science, University of Kaiserslautern, 2007.
- [20] Gast, H., *Managing proof documents for asynchronous processing*, in: *User Interfaces for Theorem Provers (UITPs 2008)*, ENTCS **226** (2009), pp. 49–66.
- [21] Gast, H., *I<sup>3</sup>P web page*, [www-pu.informatik.uni-tuebingen.de/i3p](http://www-pu.informatik.uni-tuebingen.de/i3p) (2010).
- [22] Gneiting, A., “I<sup>3</sup>P/Eclipse,” Master’s thesis, Wilhelm-Schickard-Institut, University of Tübingen (2010).
- [23] Haneberg, D., S. Bäuml, M. Balsler, H. Grandy, F. Ortmeier, W. Reif, G. Schellhorn, J. Schmitt and K. Stenzel, *The user interface of the KIV verification system - a system description*, in: *Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005)*, 2005.
- [24] J. Charles, J. K., *A lightweight theorem prover interface for Eclipse*, in: *User Interfaces for Theorem Proving*, 2008.
- [25] Johnson, R. E., *Components, frameworks, patterns (extended abstract)*, in: *Proceedings of the 1997 symposium on Software reusability* (1997), pp. 10–17.
- [26] Kaliszzyk, C., *Web interfaces for proof assistants*, in: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, ENTCS **174** (2007), pp. 87–107.
- [27] Lüth, C. and B. Wolff, *Functional design and implementation of graphical user interfaces for theorem provers*, *Journal of Functional Programming* **19** (1999), pp. 167–189.
- [28] Martin, R. C., “Clean Code,” Prentice Hall, 2009.
- [29] Schmidt, D., M. Stal, H. Rohnert and F. Buschmann, “Pattern-oriented Software Architecture: Patterns for concurrent and networked objects,” Wiley & Sons, 2000.
- [30] Shneiderman, B., “Designing the User Interface: Strategies for Effective Human-Computer Interaction,” Addison-Wesley, 1998, 3rd edition.
- [31] Théry, L., Y. Bertot and G. Kahn, *Real theorem provers deserve real user-interfaces*, in: *Proceedings of the fifth ACM SIGSOFT symposium on software development environments*, 1992, pp. 120–129.
- [32] Wagner, M., S. Autexier and C. Benz Müller, *PlatΩ: A mediator between text-editors and proof assistance systems*, in: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, ENTCS **174(2)** (2007), pp. 87–107.
- [33] Wenzel, M., *personal communication*.