

Towards a Modular Extensible Isabelle Interface

Holger Gast

Wilhelm-Schickard-Institut für Informatik
University of Tübingen
gast@informatik.uni-tuebingen.de

Abstract. We present the architecture of an Isabelle user interface based on design principles and mechanisms from the Netbeans platform. The architecture addresses the challenges of maintainability and extensibility through a modular structure and the novel concept of features as units of prover functionality.

1 Introduction

Maintaining a user interface for a theorem prover can be a challenge. Since both the prover and the GUI technology are subject to frequent extensions and changes, the interface software must be adapted and extended in parallel. This problem is especially pressing as the maintenance of the interface takes resources off the central concern of developing the prover itself. Different strategies to solve this challenge have been proposed.

The first strategy is to distribute with each prover version a corresponding version of a specialized interface (e.g. [18, 14, 1]). In this solution, the burden of maintenance lies entirely with the prover developers, but the close coupling between the prover and interface also allows direct and pragmatic solutions. For instance, access to the term structure and the proof state can be implemented through shared data. In a variant, the prover interface is embedded into a programmable text editor since the high-level access to proof scripts offered there is more stable than the details of the underlying GUI libraries (e.g. [17, 2]).

A second strategy is to develop a generic interface, to specify a public protocol, and to let the prover and interface communicate only through this protocol [4, 3]. Ideally, this solution separates changes in the prover and in the interface from each other entirely. In practice, meeting the requirements of the protocol for a given prover can be surprisingly hard and may even require changes to the software structure of the prover [20]. Furthermore, designing a single protocol that anticipates the demands and capabilities of several provers is a major challenge, given that the provers change over time. However, the burden of maintenance is distributed between the developers of the user interface and the prover.

Finally, it is possible to implement advanced user interfaces with virtually no support from the prover [12]. By employing standard patterns in user interface design, the software can be structured in such a way that changes necessitated by developments in the prover can be accommodated by local changes to the interface software. The solution has, however, the drawback that the interface

software must duplicate some functionality already present in the prover. In particular, the demands on parsing proof scripts are substantial.

The conclusion to be drawn from this overview is that neither a strict separation nor a tight integration of prover and interface are ideal solutions to the maintenance problem. While the user interface should be mostly independent of the prover, it should access already implemented functionality through well-defined interfaces. When the prover is modified, most of the interface implementation should remain unchanged.

A second concern in designing the prover interface is extensibility. The user experience can often be enhanced substantially by application-specific modes of interaction (e.g. [16]). In such settings, the interface must be extended with specialized views that access functionality added to the prover in loaded theories.

This paper approaches the maintenance and extensibility problems by applying techniques used in the Netbeans platform [8]. As a platform for application programming, Netbeans provides patterns and mechanisms for assembling different modules into a consistent application. Using these mechanisms, we propose a modular architecture for an Isabelle interface. Specifically, we address the problem of translating new prover functionality into increased interface functionality: the overall functionality is split into fine-grained, self-contained *features*. Each feature comprises a prover-side implementation, an interface-side proxy [11], and one or more interface-side views that allow the user to access the functionality. The purpose of this split is to confine changes in the prover to specific features, thus reducing the part of the interface code that must be modified when the prover changes. Furthermore, new features can be installed into the prover at run-time to enable application-specific interaction modes. The concepts described in this paper have been validated through a prototype implementation.

Organization Section 2 summarizes the Netbeans mechanisms used. Section 3 describes the overall architecture. Section 4 gives applications that further motivate the decisions taken in the architecture. Section 5 concludes.

Acknowledgments I would like to thank Makarius Wenzel and Burkhart Wolff for discussions on the design goals of user interfaces for theorem provers, on the challenges encountered in the day-to-day maintenance of the Isabelle interface, and in programming interface extensions.

2 Netbeans Mechanisms for Modularity

The Netbeans platform [8], which underlies the Netbeans IDE, is a framework that supports the development of arbitrary applications. Its modular structure, platform independence, and straightforward usage make it particularly attractive for building a prover interface. The modularity itself rests on a few generic mechanisms, to be summarized in this section, which can be leveraged to the design problems identified in the introduction. Furthermore, the mechanisms described in this section are available as standalone libraries and can be used independently of the Netbeans platform itself.

2.1 Modules

A Netbeans module is a standard JAR archive that will be loaded into the platform at runtime. Its manifest may contain additional meta-information such as the version and dependencies on other modules [8, Ch. 3]. Dynamic loading and unloading are supported as well [8, §3.3]. An update mechanism [8, Ch. 22] supports the distribution of new versions of modules to end-users. The reliance on JVM mechanisms, as well as a mature IDE support, ensure that splitting an application into different modules does not lead to much development overhead, compared to a monolithic implementation.

2.2 Lookups

Frameworks in general provide generic mechanisms that can be adapted to specific application contexts. This goal requires that the generic mechanisms can access application-specific functionality. In the Netbeans platform, objects expose their specific capabilities through *lookups* [8, §5.2]. A lookup is a collection of objects that can be queried using a required type. The interface `Lookup` provides a method that returns the first object in the collection implementing the interface given by the `clazz` argument:

```
T lookup(Class<T> clazz)
```

Consider, for instance, a menu item “Save” that operates on the current selection. Netbeans represents the selection by a set of generic *nodes* [8, Ch. 9]. Since not all selected elements can be saved, the “Save” entry must access that capability, which is represented by the `SaveCookie`, through the node’s lookup. If `s` is a selected element, it is sufficient to write:

```
SaveCookie sc = s.getLookup().lookup(SaveCookie.class);
```

If `sc` is not `null`, then `s` can be saved using `sc.save()`. Menu- and toolbar entries can also use the current selection’s capabilities to determine whether they should be disabled [8, §5.5].

2.3 The System Filesystem

Netbeans provides an abstraction over the OS filesystem [8, Ch. 6, 10]: files are assigned MIME types, and they can be accessed using readers registered for the MIME types. Writing applications that manipulate already defined types of files is thus greatly simplified. Besides this abstraction, the *system filesystem* also contains virtual paths contributed by the modules loaded into the platform. If a module’s JAR manifest contains a reference to a `layer.xml` file, that file’s content is visible in the system filesystem, and other modules can access the contribution. The system filesystem is used as the central extensibility mechanism in Netbeans.

For instance, an *action* is a UI representation of functionality. Actions usually appear in menus and toolbars, and the user can customize a Netbeans applications by assigning them to specific places. A module that wishes to contribute

a new action simply places its implementation class into a sub-folder of folder `Actions` in the system filesystem. The extension `.instance` here indicates that the file represents an instance of a Java class [8, §6.6]. Accessing the file yields that instance.

```
<filesystem>
  <folder name="Actions">
    <folder name="Isabelle">
      <file name="org-isabelle-theoryactions-GotoPointAction.instance"/>
    </folder>
  </folder>
</filesystem>
```

A frequent access pattern is to extract from a folder all instances that implement a given interface. The convenience class `Lookups` provides a method that makes all instances in a given folder available in a lookup (Section 2.2):

```
Lookup forPath(String path)
```

They can then be retrieved using the following method in `Lookup`:

```
Collection<? extends T> lookupAll(Class<T> clazz)
```

In summary, the Netbeans platform makes it simple to write extensible applications: an extension point defines the required capabilities of extensions as an interface type; it then specifies a folder in the system filesystem where extensions should be deposited; finally, it accesses the extensions using `Lookups.forPath()`. The approach is particularly lightweight in that the main part of the work is carried out at the programming language level — it is not necessary to understand a separate extension mechanism [10].

3 Architecture

This section describes the architecture addressing the problems stated in the introduction. Section 3.1 gives a general overview. Section 3.2 treats prover-specific capabilities in more detail. Section 3.3 shows an example feature loaded into the prover at startup. Section 3.4 points out aspects concerning the maintainability.

3.1 Overview

The system is divided into three layers (Figure 1): the lowest layer contains the prover running in a separate process. The infrastructure layer contains the functionality, including the management of proof documents (cf. the model-view-controller pattern [9]). A thin presentation layer lets the user interact with proof documents and displays the current state of the proof processing. The presentation layer does not contain functionality itself, but delegates all requests to the infrastructure. For instance, the theory outline component merely implements an adapter [11] that translates the existing proof document structure into a tree data structure suitable for the Netbeans UI components.

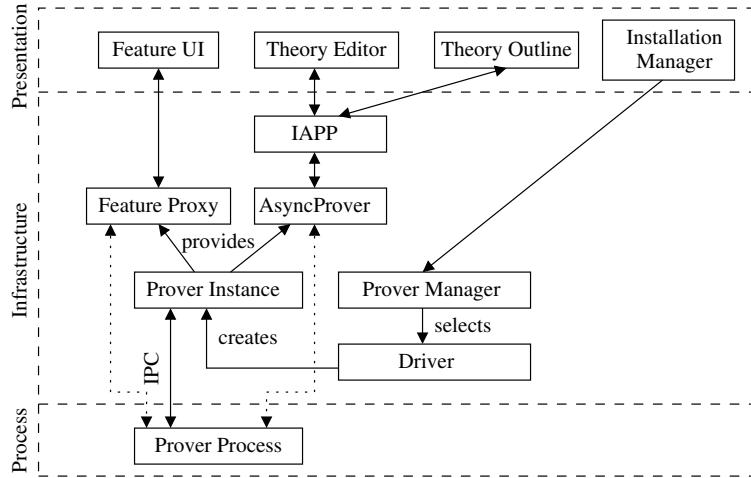


Fig. 1. Architecture Overview

Since the prover runs in a separate process, it must be accessed via inter-process communication (IPC). Following Wenzel [19], the actual communication protocol is hidden. The *prover instance* object encapsulates the life-time of a single prover process and the communication. The prover instance will usually depend on the specific prover version used.

However, the prover instance is not a monolithic object defining the interface to the prover once and for all. Instead, it exposes capabilities of the supported prover version as *features* using the lookup mechanism (Section 2.2). One feature is the support required by the IAPP (infrastructure for asynchronous proof processing) [13]. However, the set of features is not fixed (Section 3.2). The features are represented as proxies [11] that access functionality inside the prover in the background. Again, the prover communication is encapsulated.

The *prover manager* keeps track of provers installed on the system. It uses a *prover driver* to create a suitable prover instance for an installed prover. Prover drivers are implemented as separate modules and registered with the prover manager via the system filesystem (Section 2.3). This setup allows the driver for a new prover version to be distributed through the Netbeans update mechanism [8, Ch. 22]. It is expected that prover drivers can determine whether they apply to a given prover installation, for instance by checking the version number printed by the prover upon startup. The platform can thus accommodate any number of drivers simultaneously.

3.2 Features

Features represent the capabilities of specific prover versions. Figure 2 highlights the role of a feature as a remote proxy [11]: a UI component, or a component in the infrastructure, accesses the capability by a method call. In the background,

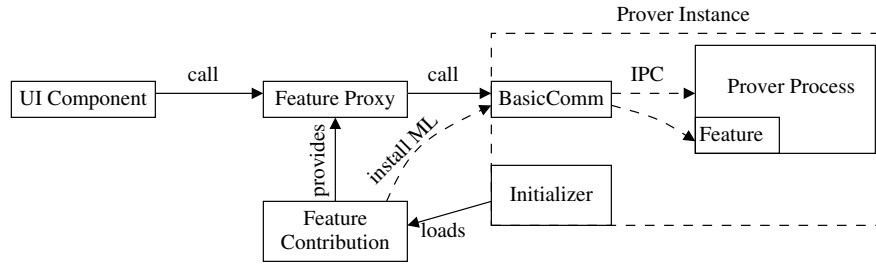


Fig. 2. Feature Proxies

the feature proxy communicates with the prover process to compute the result. Two extensions to the basic idea are worth mentioning. First, the feature proxy itself usually does not use low-level IPC. Instead, it uses a basic communication facility shared by all features in a given prover instance. This facility is, again, published as a feature (see Section 3.3).

Second, not all features will be available in the basic prover version: some may be application-specific, others may be contributed by developers of the user interface. At startup time, the prover instance therefore scans a particular directory in the system filesystem (Section 2.3) for feature implementations, identified by the interface `FeatureImpl`. Each feature implementation is given the chance to install new functionality into the running prover by sending ML code. The result of the method `installFeature()` is a new feature proxy, which is added to the prover instance’s lookup. The code for loading feature extensions is, indeed, very short:

```

Lookup l = Lookups.forPath("Provers/Isabelle2008/features");
for (FeatureImpl fi: l.lookupAll(FeatureImpl.class)) {
    features.add(fi.installFeature(this));
}

```

3.3 Implementing a Contributed Feature

Contributing new functionality is straightforward. We consider as an example a feature that extracts information from the current proof state. It is prototypical of applications that wish to render goals in an application-specific fashion. A module `ExtStateQuery` (Ext for *extension*) contains the implementation.

First, the functionality is implemented at the ML level by defining a new (improper) Isar command “`ext_state_query what`”, whose parameter indicates which information is to be retrieved. Using Isabelle’s `Term` module, the constants in the current proof state are determined easily and written to the standard output. The ML implementation resides in `ExtStateQuery` as a plain text file.

Next, the ML implementation is installed to the prover during startup. Towards that end, `ExtStateQuery` registers the class `StateQueryImpl` with the prover driver using the `layer.xml` file (Sections 2.3, 3.2):

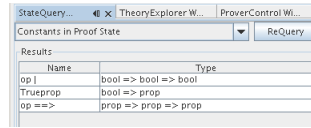


Fig. 3. View Component for State Query

```

<folder name="Provers">
  <folder name="Isabelle2008">
    <folder name="features">
      <file name="StateQuery.instance">
        <attr name="instanceClass"
          stringvalue="org.isabelle.ext.statequery.prover.StateQueryImpl"/>
      </file>
    </folder>
  </folder>
</folder>
...

```

Class `StateQueryImpl` has a method `Object installFeature(ProverInstance pi)`. It reads the ML file content into a variable and sends it to the prover using a basic communication facility:

```

InjectedCommands inj = proverInst.getFeature(InjectedCommands.class);
inj.ML(mlFileContent)

```

The method `getFeature()` is a convenience for `getLookup().lookup()`. The `InjectedCommands` feature executes commands when the prover becomes idle. The commands are thus injected into the normal processing of proof documents.¹ After installing the ML implementation in this way, the method returns the feature proxy of type of `StateQueryFeature`.

The class `StateQueryFeature` provides a method that carries out the query in the background. Its signature is `Constant[] constantsInProofState()`, i.e. it presents the result as proper objects. The method's implementation uses, again, `InjectedCommands` to communicate with the prover:

```

InjectionResult res = inj.command("ext_state_query constants");

```

The `InjectionResult` is a future, a placeholder for a result computed asynchronously. Using `res.await()`, the method blocks until the prover has executed the command. The future then contains the standard output of `ext_state_query`, which is analyzed and packed into `Constant` objects (with name and type).

Finally, a view using the feature is implemented directly using Netbeans' Matisse GUI builder (Figure 3). Accessing the constants in the current goals in its implementation is straightforward, using the proxy introduced above:

```

StateQueryFeature q = pi.getFeature(StateQueryFeature.class);
Constant res[] = q.constantsInProofState();

```

¹ For asynchronous proof processing[13], `command()` takes an additional parameter, the ID of the command after which the injected command should be executed.

3.4 Addressing Maintainability

The architecture supports maintainability by modularity: features represent fine-grained units of functionality, made available to the user interface through standard method calls. The remainder of the infrastructure and presentation layers is insulated from changes to the feature implementation as long as the method-call interface remains the same. Furthermore, it can be expected that changes to one feature do not affect other features. The result is that changes to the prover have local effects on a well-defined parts of the interface implementation.

Furthermore, the strict division of presentation and infrastructure makes the infrastructure testable by standard approaches such as JUnit. Testability, on the other hand, is a prerequisite for changeability and maintainability [6]: developers can be more confident about necessary changes if they can determine quickly and automatically whether existing functionality has been broken. Note that the situation with a user interface differs from the case for the prover itself: a change has not broken a prover’s functionality if all libraries continue to be accepted. Most code in a user interface, on the other hand, is executed only very infrequently, when the user happens to request a particular action.

4 Further Applications

The proposed architecture supports several usages beyond a standard prover interface. This section sketches these applications briefly.

Isabelle as a Back-end Prover Isabelle is not only used for interactive proof development, but also for background proofs in higher-order logic (e.g. [21]). In such applications, the infrastructure layer can serve as a high-level interface to access Isabelle’s functionality. In particular, the `InjectedCommands` feature (Section 3.3) enables execution of single commands.

Background Queries The example in Section 3.3 shows how the term structure in goal states can be queried through features. We expect this possibility to be very useful for application-specific theorem proving where theories would be accompanied by specialized viewers for the formalized objects. Proof-by-pointing applications [16, 15] could be realized by suitable markups in terms. More extensive computations, for instance the generation of fragments of proof documents [5], could be realized on the prover side by contributed features as well.

Driver-specific Capabilities Prover drivers so far are responsible only for the creation of prover instances for installed provers. However, by applying the lookup pattern to the drivers themselves, more detailed support can be made available. For instance, an `InstallationFeature` might offer the capability of downloading and installing the prover itself into a particular location and configuring and compiling it according to the system environment.²

² Thanks to Burkhart Wolff for this suggestion.

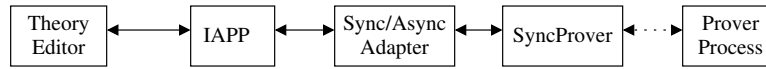


Fig. 4. Support for Synchronous Provers

Synchronous Provers The IAPP, as defined in [13], assumes the prover instance to support the protocol for asynchronous proof processing, even if the proving proceeds in linear order effectively. For backwards compatibility, it would be desirable if a prover could implement a simpler **SyncProver** interface, modelled e.g. after [7]. Using features, the desired adaptation is straightforward (Figure 4): when failing to retrieve the **AsyncProver** feature, the IAPP asks for the **SyncProver** feature and uses a simple adapter, the emulator component from [13, Figure 5], to obtain the desired functionality.

5 Conclusion

We have proposed a modular architecture for an Isabelle user interface. Its main design goal is to simplify the maintenance of the prover interface. Towards that end, we have introduced *features* as units of functionality that consist of a prover-side implementation and a interface-side proxy. The proxy object hides the background communication such that changes to the prover do not affect the interface. The architecture separates an infrastructure layer, which provides the functionality, from a presentation layer, which is responsible for the user interaction. The division increases testability as a necessary prerequisite to changeability, and makes Isabelle accessible as a background prover in non-interactive contexts. Finally, the architecture supports application-specific theorem proving by contributed features, which access functionality that is related to loaded theories or is installed at startup.

The design elements underlying the architecture are taken from the Netbeans platform: its *lookups* are used to publish the specific capabilities of objects and its *system filesystem* serves as an extension mechanism. Since these facilities are also available as standalone libraries, the architecture does not, in principle, depend on a Netbeans realization of the entire interface.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.
2. D. 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.
3. D. Aspinall, C. Lüth, and A. Fayyaz. Proof General in Eclipse: System and architecture overview. In *Eclipse Technology Exchange Workshop at OOPSLA 2006*, 2006.

4. D. Aspinall, C. Lüth, and D. Winterstein. Parsing, editing, proving: the PGIP display protocol. In *International Workshop on User Interfaces for Theorem Provers 2005 (UITP'05)*, 2005.
5. D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In *Mathematical Knowledge Management 2005 (MKM '05)*, number 3863 in Springer LNAI, pages 65–80, 2005.
6. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Amsterdam, 1999.
7. Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *J. Symbolic Computation*, 25:161–194, 1998.
8. T. Boudreau, J. Tulach, and G. Wielenga. *Rich Client Programming: Plugging into the Netbeans platform*. Prentice Hall, 2007.
9. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A System of Patterns*, volume 1. Wiley & Sons, 1996.
10. The Eclipse workbench. <http://www.eclipse.org>.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
12. H. Gast. An architecture for extensible Click'n Prove interfaces. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, number 364/07. Department of Computer Science, University of Kaiserslautern, Aug. 2007.
13. H. Gast. Managing proof documents for asynchronous processing. In *User Interfaces for Theorem Provers (UITPs 2008)*, volume 226 of *ENTCS*, pages 49–66. Elsevier Science Publishers B. V., 2009.
14. D. Haneberg, S. Bäumlner, 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.
15. C. Lüth, H. Tej, Kolyang, and B. Krieg-Brückner. TAS and IsaWin: Tools for transformational program development and theorem proving. In *Fundamental Approaches to Software Engineering: Second International Conference (FASE'99)*, volume 1577 of *LNCS*, 1999.
16. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 19(2):167–189, 1999.
17. S. Owre. A brief overview of the PVS user interface (invited tutorial). In *UITP*, 2008.
18. C. Team. The Coq proof assistant. <http://www.lix.polytechnique.fr/coq/>, 2009.
19. M. Wenzel. Interactive proof documents – theorem provers for user interfaces. <http://www4.in.tum.de/wenzelm/papers/edinburgh2008.pdf>, Nov. 2008.
20. M. Wenzel. personal communication, 2008.
21. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.