Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 226 (2009) 49–66
www.elsevier.com/locate/entcs

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

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.

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

doi:10.1016/j.entcs.2008.12.097

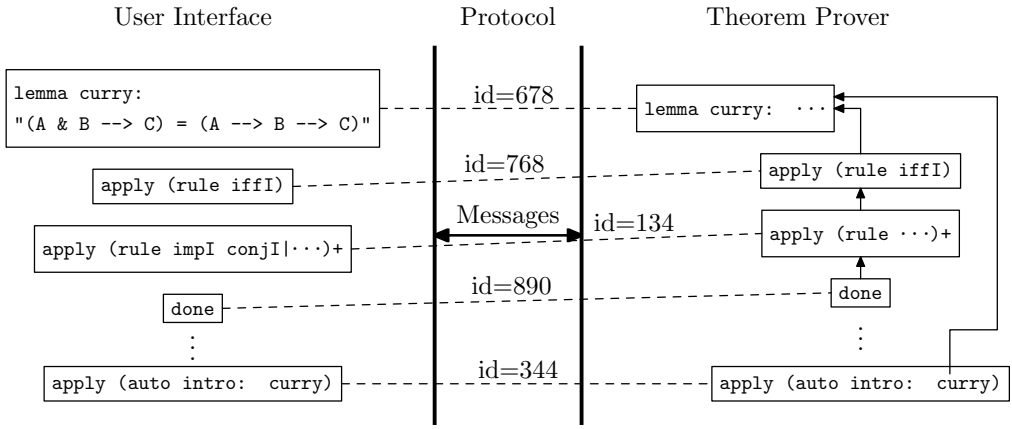

Fig. 1. Commands in the Interface and Prover

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

Wenzel [18] has recently pointed out that the linear processing model is far from optimal. The first possible improvement is the use of multi-core processors for parallel processing of independent proof commands. In the Isar [20] 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. The goal is to provide a behaviour that is similar to that of the Mizar system [12]. There, the prover runs in batch mode and annotates the input proof document with error messages where processing fails. However, it continues processing at the first command that does not depend on the erroneous command. In this way, 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 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 proof document 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 [19]. 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

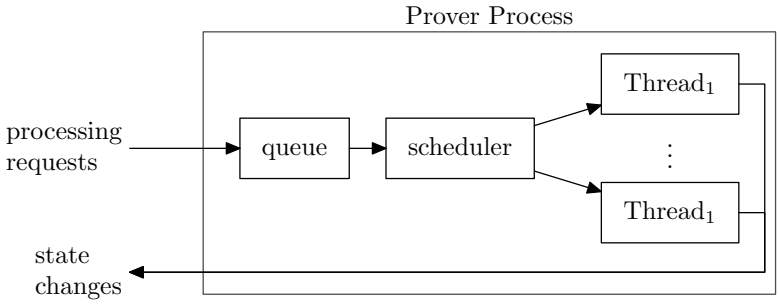

Fig. 2. Model of the Prover based on the Active Object Pattern

has proven successful in systems that perform 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 not have a 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 serial-

izing accesses to shared resources which occurs in any form of asynchronous or concurrent processing. In the present 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* reflect that neither sending nor revoking a command are synchronous operations, because due to dependencies among commands, both may take a noticeable amount of 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.

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 proof document processing and delineate the approach from sequential, history-based models.

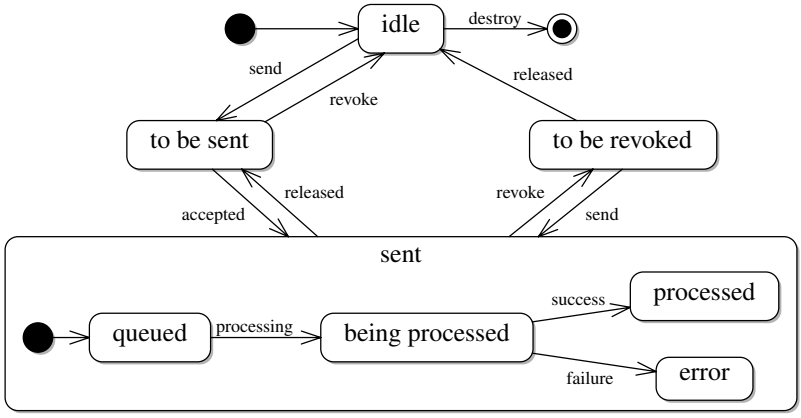

Fig. 3. A new state model

- 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 de-

scribed 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

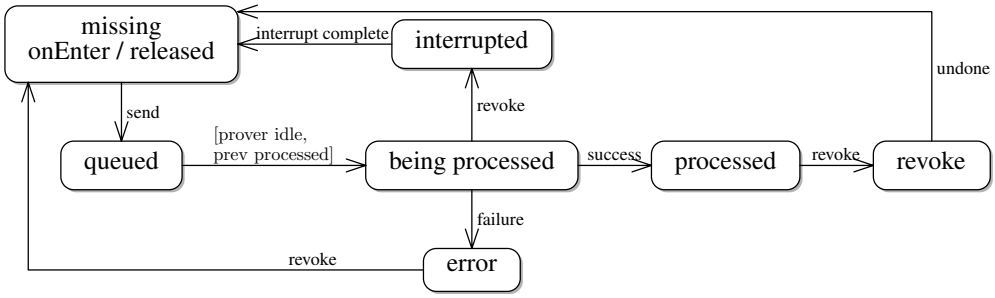

Fig. 4. State Model in the Emulator

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.

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

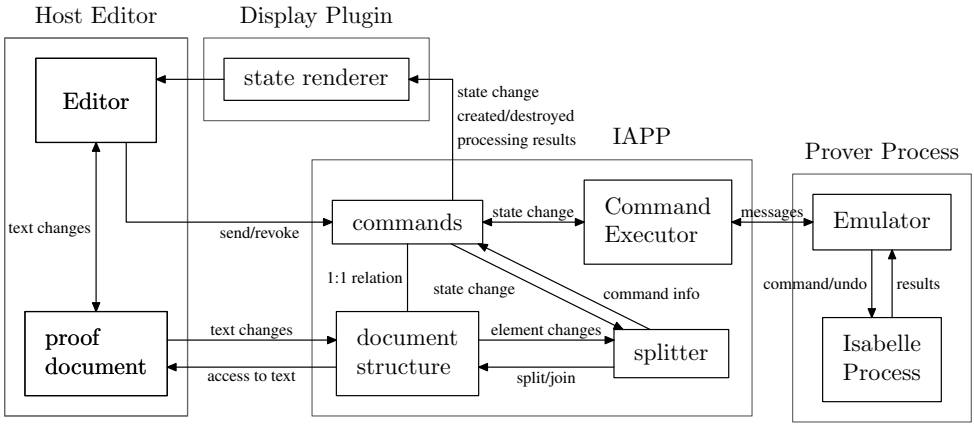

Fig. 5. Software Structure

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, version control, and syntax highlighting are available without cost, and the user may 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 Java interfaces that define the expected functionality (Figure 6). A `Document` has methods for accessing the text and for adding and removing observers of type `DocumentListener`. Following the SWT widget set [16], a single change to the document consists

```
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

of removing a number of characters and inserting a number of characters at a specific position. 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.

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 elements: when text is inserted or removed, the positions of subsequent elements are increased or decreased. The implementation uses a gap-store data structure to make the computation efficient. The document structure also identifies those elements that are affected by a change and reports them to the splitter. The splitter then decides whether the change leads to splitting or joining of 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 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`.

```
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

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 [19], we have chosen to use basic Swing widgets for the prototype. Figure 8 shows the result.

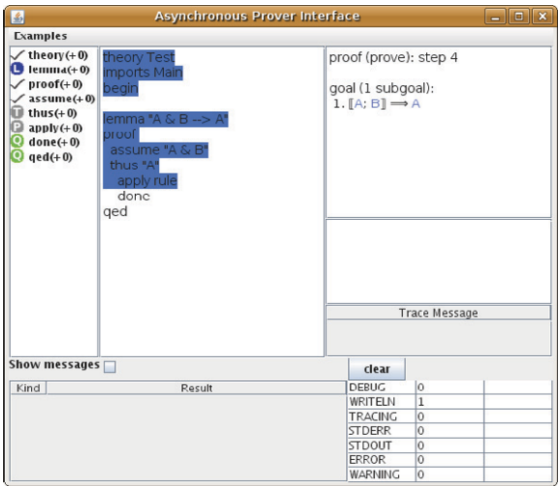

Fig. 8. Screenshot of Example Interface

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) 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 reflects edits by the user immediately: 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 [18,19] 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. Summaries 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 ef-

fectively 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 [18], for instance, the interface sends 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. If a command is found to contain an error, it is revoked and left idle until the user has edited it again. Although some commands are executed only speculatively, with multi-core processors the user does not notice an increased response 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 textual commands in the provers' native languages, the document structure is represented by XML markups.

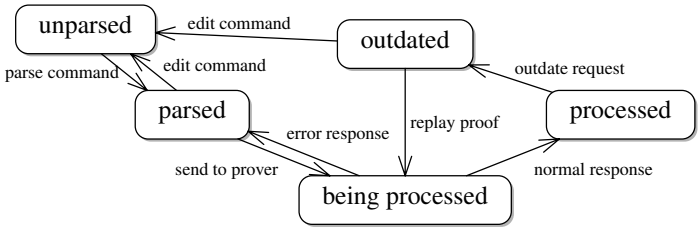

Fig. 9. PGIP Command States

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. [17,8] 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 [17] 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. Department of Computer Science, University of Kaiserslautern, 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] SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/.

[17] 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.

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

[19] Makarius Wenzel. personal communication, 2008.

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