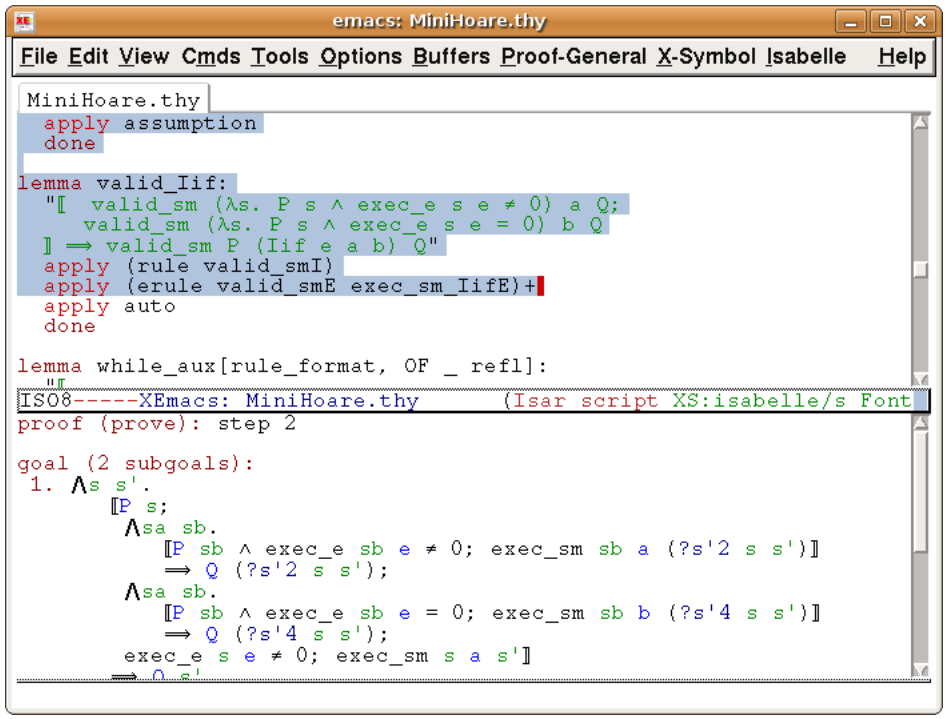
Managing Proof Documents for Asynchronous Processing

Holger Gast
Wilhelm-Schickard-Institut für Informatik
University of Tübingen

22.8.2008
UITP '08, Montreal

Traditional User Interfaces



- Idea: Script buffer [3, 1]
- UI keeps proof script for batch replay
- Linear processing, commands become “locked”
- Focus on mechanics of proving – user-friendly?

Document-Centered View

- Metaphor "proof document"
  - User edits human-readble a proof document
  - Prover checks the consistency
- Assisted authoring [2]
  - Isar as human-readable proof language [8]
  - Backflow: Assistance by prover for editing
  - Processing linear
- PlatΩ approach [6]
  - Near-natural, text-book style input language
  - Front-end parses structure & computes structural diff
  - Triggers necessary (re-)checking by Ωmega prover

Asynchronous Proof Processing [7]

- Observation
  - 95% of processing time spent in proofs
  - . . . but later commands do not depend on proofs at all
- Asynchronous proof processing (APP)
  - Process definitions, theorems, etc. immediately
  - Check syntax & explicit references immediately
  - Execute proofs when processor would be idle

Applications

- Parallel execution
- Traverse intermediate theory quickly between . . .
  - Definition & theorem
  - Theorem & reference
  - Tactic definition & application
- Strategy: Continuous proof processing
  - Isabelle processes the newest version of each command
  - Success/failure is presented as annotations

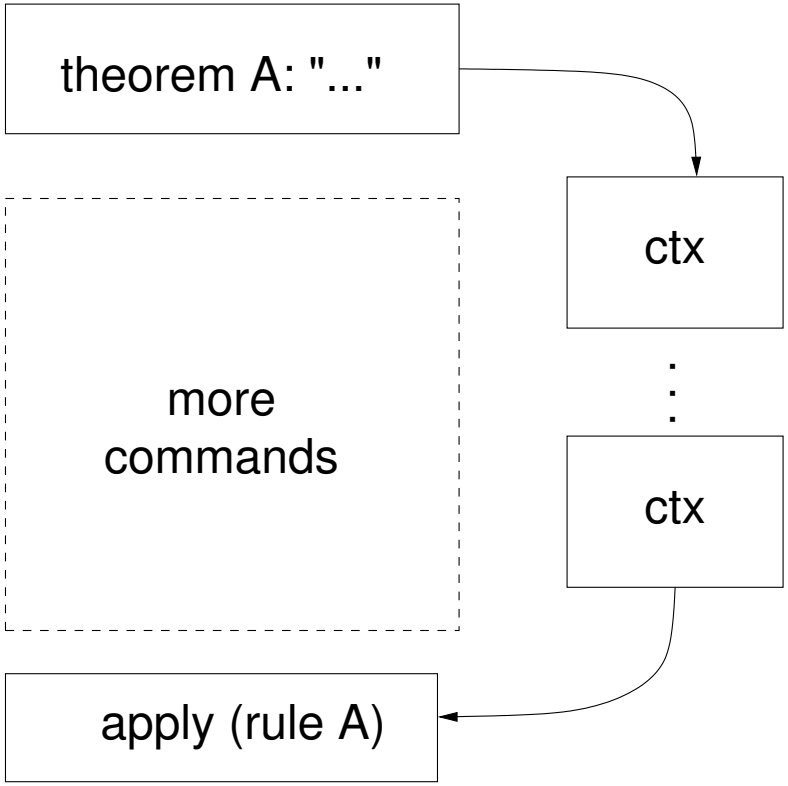
  $\Rightarrow$ IDE-like theory development

Outline

- Design considerations for asynchronous proof checking
- Protocol for UI/prover communication
- Infrastructure for asynchronous proof checking
- Design principle: minimal assumptions = maximal re-use
- Applied to Isabelle/Isar – ongoing work with Makarius Wenzel

Design Considerations

Dependencies in Isabelle

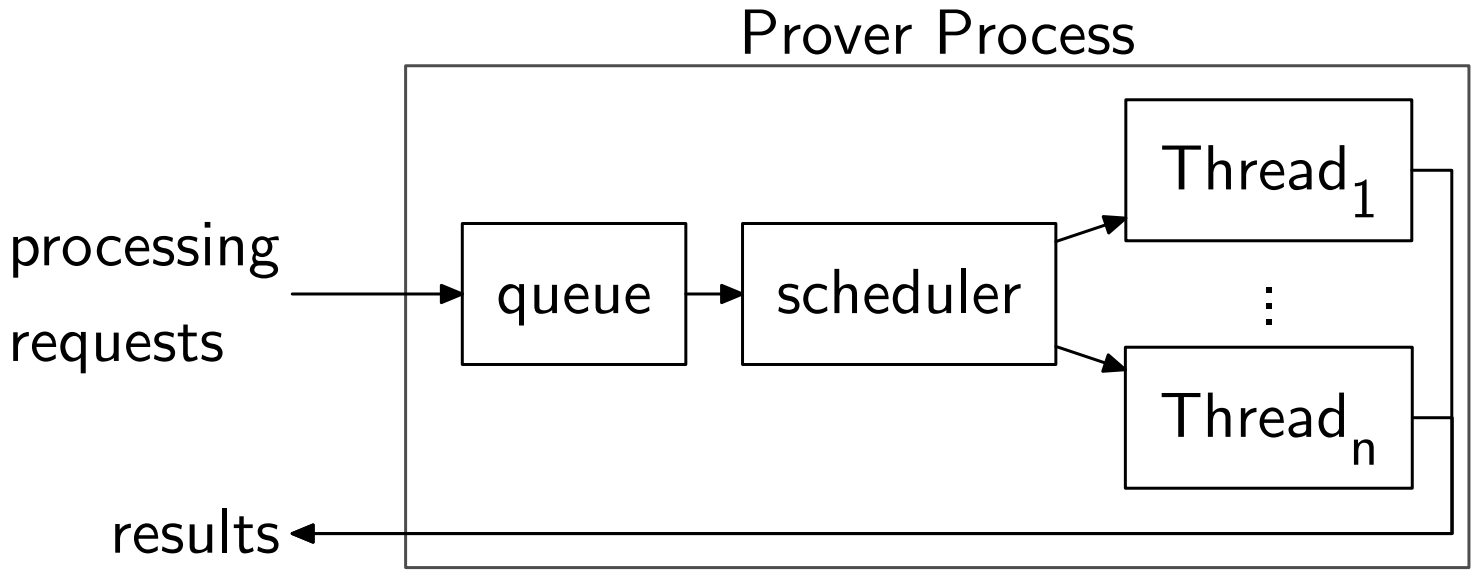


Implicit Dependencies

- Implicit rule sets: `intro`, `simp`, `cong`, . . .
- Used by various methods: `blast`, `simp`, `auto`, . . .
- Proofs may break because new rules are added
- Extensible: Isar framework is not aware of dependencies
- Conclusions
  - Dependencies are hard to track
  - Design APP to be independent of possible solution

  $\Rightarrow$ Don't prescribe particular dependency model

Order of Processing



- Active Object [5] / Lightweight Executable Framework [4]
- Asynchronous / parallel / concurrent / distributed processing
  - Results computed at the convenience of the service provider
  - Clients perform other computations until result arrives

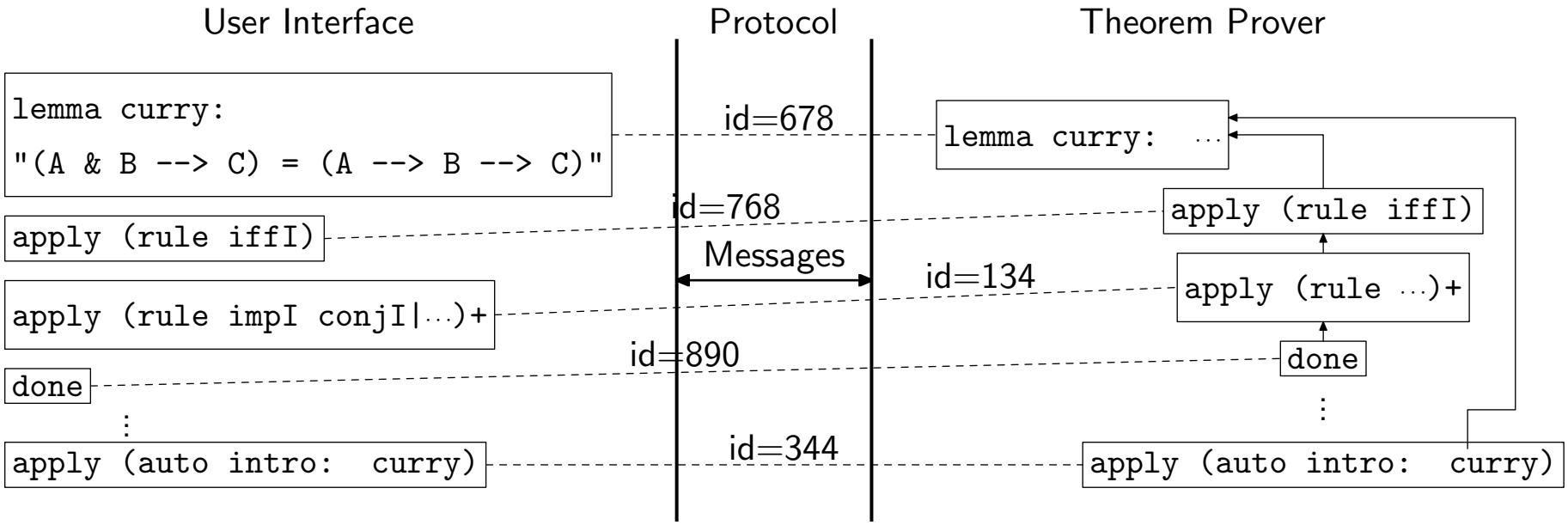
Minimal Assumptions — The Prover

- Principle: The UI makes no assumptions about
  - The processing order of individual commands
  - The dependencies between / independence of commands
- Benefit: Incremental implementation
  1. Start with linear processing
  2. Develop dependency tracking models
  3. Multi-threading & asynchronous processing
- Benefit: Software structure
  - Adapt implementation to existing mechanisms
  - Exploit internal knowledge in implementation
  - Changes possible without changing the UI

Division of Reponsibilities

- Isabelle
  - Processing order
  - Dependency tracking
- User Interface
  - Identification commands in the proof document
  - Synchronization of access by user / Isabelle

Identifying Commands



- Splitting done by user interface
  - Isabelle works at command level ⇒ Facilitate re-work
  - Immediate reaction; auxiliary information (outline, . . .)
  - Service available to other provers

Demands on Identifying Commands

- Reliability
  - Entire protocol depends on identity of commands

  $\Rightarrow$ Must not fail in any case
- Stability
  - State information attached to commands
  - Creation/deletion may cause expensive (re-)proving

  $\Rightarrow$ Avoid spurious changes to document structure
- Interactivity
  - User keeps typing
  - Prover keeps proving

  $\Rightarrow$ Recognize & propagate changes immediately
- Synchronization: Negotiate creation/deletion with prover

Solution: Separating Keywords

- Conclusion: build small, specialized incremental parser
- Isar design: keywords separate commands
- Implementation
  - Maintain partitioning of proof document into commands
  - Split elements on creation of keywords
  - Join elements on deletion of keywords
  - Keep track of quotes (comments, inner syntax, etc.)
  - Delay split/join during negotiation with prover
- Generalization: Splitter
  - Informed about textual changes
  - Split/join components in arbitrary (Java) code

Protocol Part 1: Document Structure

- UI sends messages
  - `create`$(id, prevId)$
  - `destroy`(id)
- Isabelle
  - Maintains doubly-linked list of commands
  - And hash map $id \mapsto command\ object$

Synchronization

- Two complementary models of the proof document
  - UI: Text, keywords, markups, outline, . . .
  - Isabelle: Processing state, dependencies, . . .
- Need synchronization for consistency
- Mutexes, semaphores, etc. need shared memory
- Approach "ownership"
  - Only one process owns a shared object at a given time
  - And only the owner may access the object

Ownership of Commands

- UI is Owner
  - User may edit the command text
  - Split/join with commands are possible
  - UI may send messages `create` / `destroy`
  - Isabelle maintains no references (dependencies etc.)
  - But: Command is still present in the document structure
- Isabelle is Owner
  - Isabelle may choose to execute the command
  - Isabelle may register dependencies on command
  - UI receives processing information from Isabelle
  - User may not edit the command

Protocol Part 2: Negotiating Ownership

- Transfer UI $\longrightarrow$ Isabelle
  - Send command text to Isabelle
  - Prevent user from editing
  - Quick — except for long sequences
- Transfer Isabelle $\longrightarrow$ UI
  - Undo any dependent commands & erase references
  - Kill thread currently processing command
  - May take time
- Protocol
  - UI $\Rightarrow$ Prover
    - `send`$(id, cmdStr)$
    - `revoke`(id)
  - Prover $\Rightarrow$ UI
    - `released`(id)
    - `request`(id)

That's it — basically

Protocol Part 3: Information Messages

- Isabelle decides on processing order
- But reports progress, success/failure
- Informational messages
  - `queued`(id)
  - `startProcessing`(id)
  - `success`(id)
  - `error`(id)
  - `result`(id,$result$)
- Note: no reaction by interface required

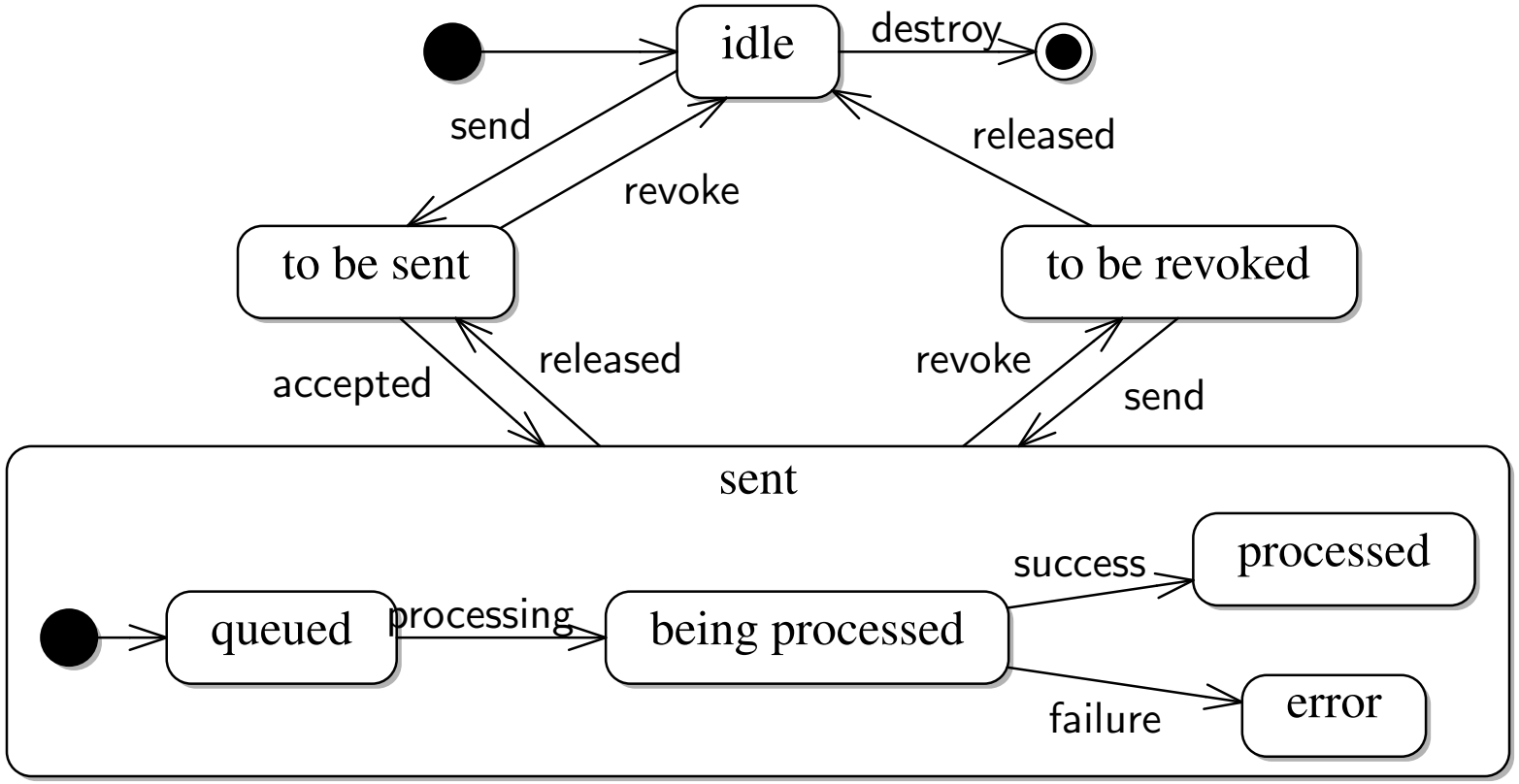
Sequences of Messages?

- Approach: Asynchronous processing
  - Prover & UI work independently
  - Events occur on commands (created, destroyed, . . .)
  - Messages are only notifications of events

$\Rightarrow$ No specific ordering of messages prescribed

- Instead: state model for commands
  - Events change the state of a command
  - The state determines which events may legally occur

State Model for Commands



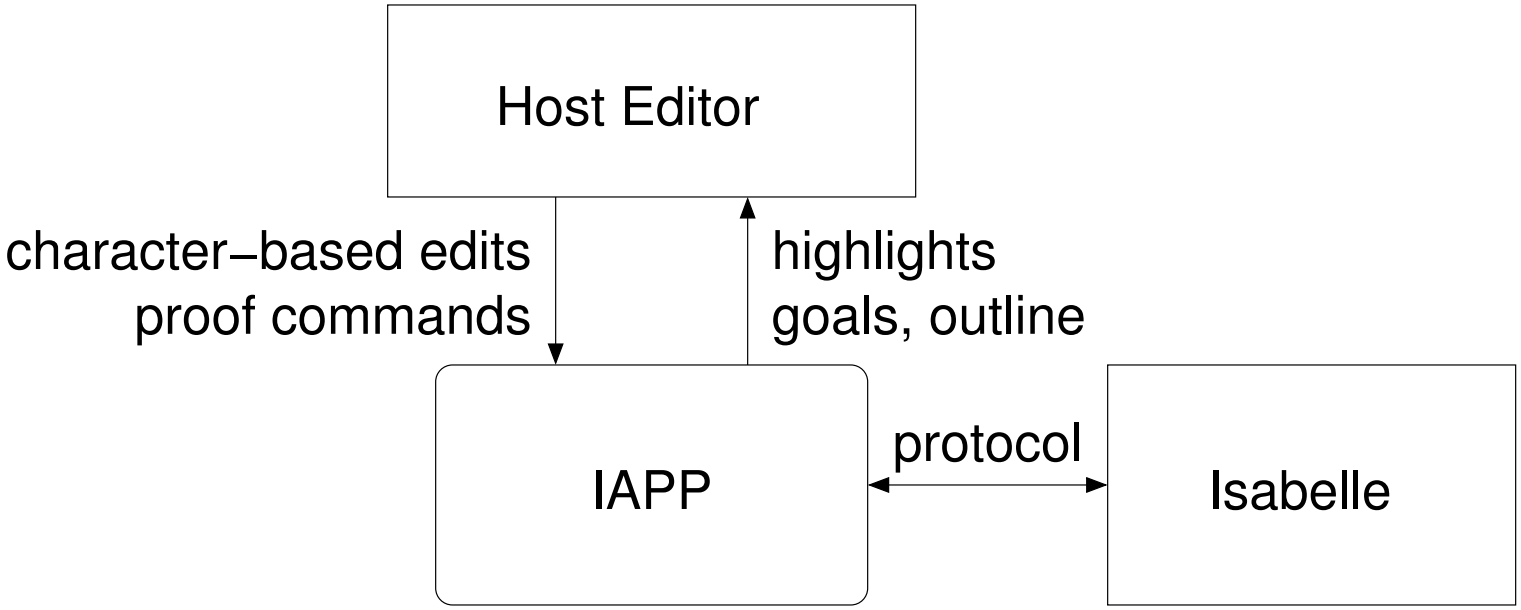
- Edit/execute/revoke cycle is present
- Commands not "locked" any longer than necessary
- Interrupt internal to Isabelle

Software Design

Points of Variation

- APP is current development
- Three points of variation
  - Editor: Emacs, Eclipse, Netbeans, JEdit, . . .
  - Interaction mode: Continous, explicit requests, . . .
  - Prover: Extension of Isabelle; perhaps other provers
- Solution: Infrastructure for Asynchronous Proof Processing
  - Abstracts over concrete editor used
  - Abstracts over the prover used

Minimal Assumptions — The Editor



- Infrastructure for Asynchronous Proof Processing (IAPP)
- Uses APP protocol for Isabelle communication
- Displays results in editor
- Receives user actions from editor

Controlling Command Execution

- The state of a command determines its execution
  - `to be send` $\Rightarrow$ will be executed at some point
  - `to be revoked` $\Rightarrow$ will be undone at some point
- Nice: Program logic for sending/revoking is small

$\Rightarrow$ Facilitates experiments with interaction styles

```
CommandState state = elem.getData();
if (state.isIdle())
  state.send();
else
  state.revoke();
```

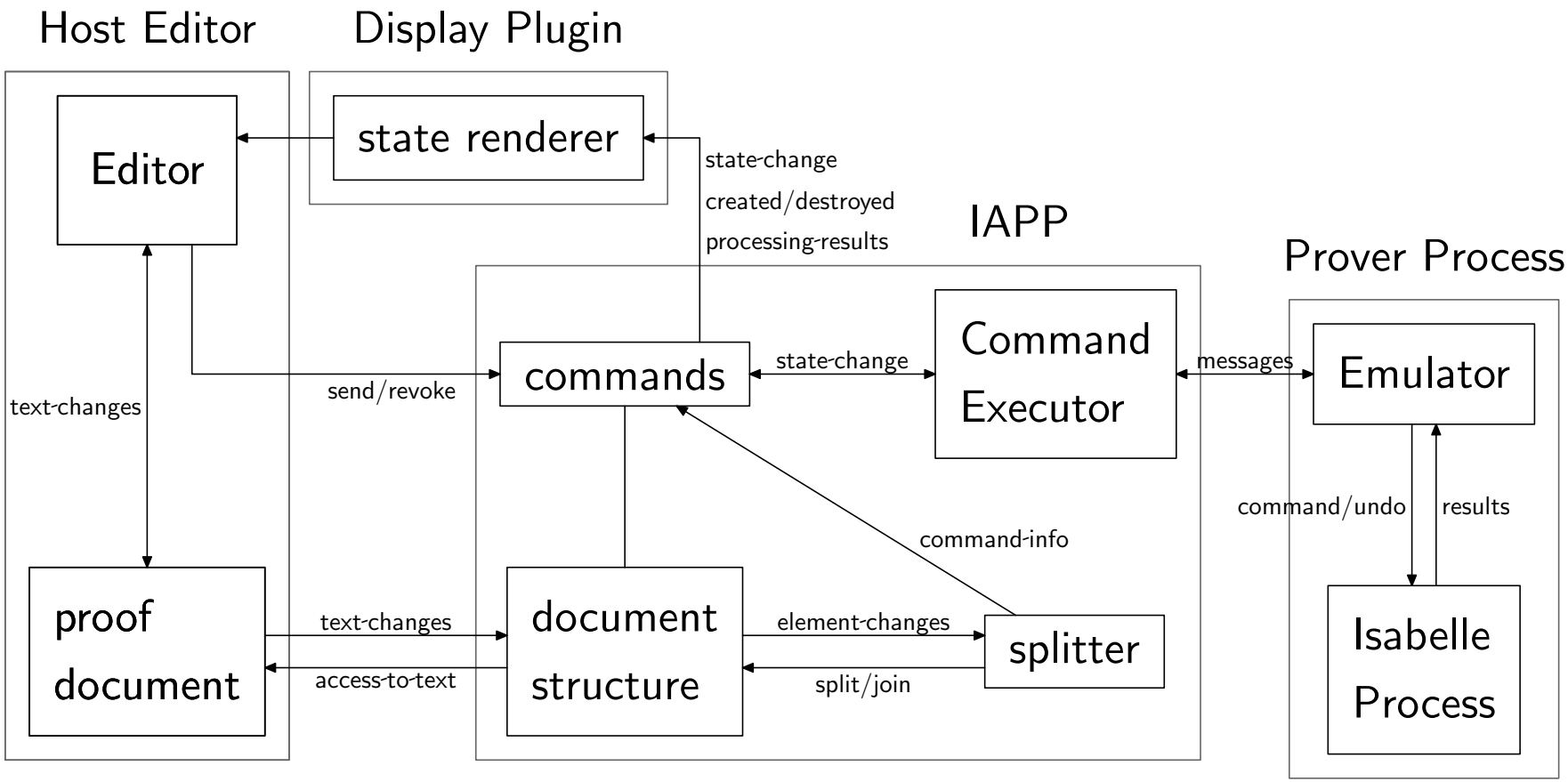
Synchronous Proving with the IAPP

- Provers may always choose to linear processing order
- Possible: Emulator that translates IAPP $\leftrightarrow$ TTY

$\Rightarrow$ Talk to existing provers in the background

- Gains
  - Experiment with styles of user interactions
  - Elements of APP desirable for traditional interface
    - Identification of commands for outline etc.
    - Event-based $\Rightarrow$ non-stalling user interface
    - Deals gracefully with long traces

Overall Software Structure



Conclusion

- UI for asynchronous proof processing
- Design principle: minimal assumptions = maximal re-use
- Collaboration of UI + prover for APP
  - Maintenance of document structure
  - Protocol
  - State model for commands
- Characteristics
  - Design entirely event-based $\Rightarrow$ asynchronous processing
  - Also TTY-based provers receive benefits

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, 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.

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

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

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

[6] 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.

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

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