Asynchronous User Interaction and Tool Integration in Isabelle/PIDE

Makarius Wenzel *

Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

Abstract. Historically, the LCF tradition of interactive theorem proving was tied to the read-eval-print loop, with sequential and synchronous evaluation of prover commands given on the command-line. This user-interface technology was adequate when R. Milner introduced his LCF proof assistant in the 1970-ies, but it severely limits the potential of current multicore hardware and advanced IDE front-ends.
Isabelle/PIDE breaks this loop and retrofits the read-eval-print phases into an asynchronous model of document-oriented proof processing. Instead of feeding a sequence of individual commands into the prover process, the primary interface works via edits over a family of document versions. Execution is implicit and managed by the prover on its own account in a timeless and stateless manner. Various aspects of interactive proof checking are scheduled according to requirements determined by the front-end perspective on the proof document, while making adequate use of the CPU resources on multicore hardware on the back-end.
Recent refinements of Isabelle/PIDE provide an explicit concept of asynchronous print functions over existing proof states. This allows to integrate long-running or potentially non-terminating tools into the document-model. Applications range from traditional proof state output (which may consume substantial time in interactive development) to automated provers and dis-provers that report on existing proof document content (e.g. Sledgehammer, Nitpick, Quickcheck in Isabelle/HOL). Moreover, it is possible to integrate query operations via additional GUI panels with separate input and output (e.g. for Sledgehammer or find-theorems). Thus the Prover IDE provides continuous proof processing, augmented by add-on tools that help the user to continue writing proofs.

1 Introduction

Already 10 years ago, multicore hardware has invaded the consumer market, and imposed an ever increasing burden on application developers to keep up with changed rules for *Moore's Law*: continued speedup is no longer for free, but has to be implemented in the application by explicit multi-processing. Isabelle has started to support parallel proof-processing in *batch-mode* already in 2006/2007, and is today routinely using multiple cores, with an absolute speedup factor of the order of 10 (on 16 cores). See also [17] for the situation of Isabelle2013.

* Research supported by Project Paral-ITP (ANR-11-INSE-001).

How does parallel processing affect user interaction? After the initial success of parallel batch-mode in Isabelle, it became clear in 2008 that substantial reforms are required in the interaction model, to loosen the brakes that are built into the traditional *read-eval-print* loop. The following aspects are characteristic to asynchronous interaction, in contrast to parallel batch processing:

- demand for real-time reactivity (at the order of 10–100 ms);
- instantaneous rendering of formal content in GUI components;
- continued edits of theory sources, while the prover is processing them;
- treatment of unfinished or failed proof attempts (error recovery);
- cancellation of earlier attempts that have become irrelevant (interrupts);
- orchestration of add-on proof tools that help in the editing process.

The present paper reports on results of more than 5 years towards asynchronous prover interaction, with recent improvements that integrate add-on proof tools via *asynchronous print functions*. All concepts are implemented in the current Isabelle/PIDE generation of Isabelle2013-2 (December 2013)[1]. The PIDE framework is implemented as a combination of Isabelle/ML and Isabelle/Scala, with Isabelle/jEdit the main application and default user-interface. The manual [14] provides further explanations and screenshots; the *Documentation* panel in Isabelle/jEdit includes some examples that help to get started.

The front-end technology of Isabelle/jEdit imitates the classic IDE approach seen in Eclipse, NetBeans, IntelliJ IDEA, MS Visual Studio etc. Fresh users of Isabelle who are familiar with such mainstream IDEs usually manage to get acquainted quickly, without learning about Emacs and the TTY loop first. In contrast, seasoned users of ITP systems may have to spend some efforts to *unlearn* TTY mode and manual scheduling of proof commands.

Subsequently, we assume some basic acquaintance with the look-and-feel of Isabelle/jEdit, but explanations of PIDE concepts are meant to extrapolate beyond this particular combination of prover back-end and editor front-end. Since Isabelle had similar starting conditions as other proof assistants several years ago, like Coq [18, §4], the HOL family [18, §1], PVS [18, §3], or ACL2 [18, §8], there are no fundamental reasons why such seemingly drastic steps from the TTY loop to proper IDE interaction cannot be repeated elsewhere. These explanations of PIDE concepts are meant to help other systems to catch up, although the level of sophistication in Isabelle/PIDE today poses some challenges.

2 PIDE Architecture

PIDE stands for "Prover IDE": it is the common label for efforts towards advanced user-interaction in Isabelle since 2009. The main application of the PIDE framework today is Isabelle/jEdit [13, 14], but there are already some alternative front-ends: Isabelle/Eclipse by A. Velykis, and Clide by C. Lüth and M. Ring [7, 8]. An experiment to connect Coq as alternative back-end is reported in [15].

[1] http://isabelle.in.tum.de/website-Isabelle2013-2

The general aims of PIDE are to renovate and reform interactive theorem proving for new generations of users, and to catch up with technological shifts (multicore hardware). The PIDE approach is *document-oriented*: all operations by the user, the editor, the prover, and add-on tools are centered around theory sources that are augmented by formal markup produced by proof processing. Document markup for old-school proof assistants is further explained in [11].

The Connectivity Problem. Proof assistants are typically implemented in functional programming languages (like LISP, SML, OCaml, Haskell) that are not immediately connected to the outer world. If built-in interface technology exists, it is typically limited in scope and functionality: e.g. LablGtk for OCaml uses old GTK 2.x instead of GTK 3.x, and GTK is at home only on Linux.

Even if we could assume the ideal multi-platform GUI framework within our prover programming environment, what we really need is a viable text editor or IDE to work with. The Java platform is able to deliver that, e.g. with text editors like jEdit, full IDEs like Eclipse, NetBeans, IntelliJ IDEA, and web frameworks like Play (for remote applications). This observation has lead to the following *bilingual approach* of PIDE with Scala and ML (figure 1).

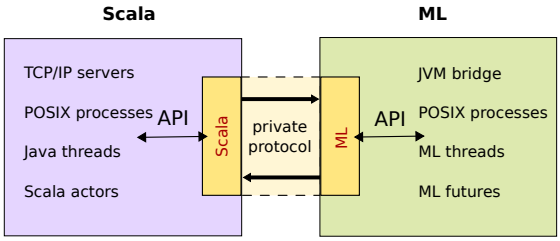

Fig. 1. The bilingual approach of PIDE: Scala and ML connected via private protocol.

Here the existing ML prover platform is taken for granted, but its scope extended into the JVM world with the help of Scala [9]. The manner and style of strongly-typed higher-order functional programming in ML is continued with Scala. Both sides happen to provide some tools and libraries for parallel programming with threads, processes, external communication, which serve as starting point for further PIDE functionality. A *private protocol* connects the two worlds: it consists of two independent streams of protocol operations that are a-priori unsynchronized. The conceptual *document-model* that is implemented on both sides is accessible by some *public APIs*, both in Scala and ML.

It is an important PIDE principle to cut software components at these APIs, and not the process boundaries with the protocol. API functions in ML or Scala are statically typed and more abstract than the communication messages of the implementation. APIs are more stable under continuous evolution than a public protocol. The combined Scala and ML sources of Isabelle/PIDE are maintained side-by-side within the same code repository: e.g. `src/Pure/General/pretty.scala` and `src/Pure/General/pretty.ML` for classic pretty-printing in

the style of D.C. Oppen (with support for document markup and font-metrics). Tools using the PIDE infrastructure may reside in ML (e.g. proof tools that output document markup), or in Scala (e.g. rendering for particular document content), or combine both worlds.

PIDE Protocol Layers. Conceptually, the two processes are connected by two independent streams of *protocol functions*. These streams are essentially symmetric, but input from the editor to the prover is called *protocol command*, and output from the prover to the editor is called *protocol message*. Syntactically, a protocol function consists of a name and argument list (arbitrary strings). Semantically, the stream of protocol functions is applied consecutively to a private *protocol state* on each side; there are extensible tables in Isabelle/Scala and Isabelle/ML to define the meaning for protocol functions.

The arguments of protocol functions usually consist of algebraic datatypes (tuples and recursive variants). This well-known ML concept is represented in Scala by case classes [9, §7.2]. The PIDE implementation starts out with raw byte streams between the processes, then uses YXML transfer syntax for untyped XML trees [11, §2.3], and finally adds structured XML/ML data representation via some combinator library. Further details are explained in [15], including a full implementation on a few pages of OCaml; the Standard ML version is part of Isabelle/PIDE. This elementary PIDE protocol stack is easily ported to other functional languages to connect different back-ends, but actual document-oriented interaction requires further reforms of the prover.

Approximative Rendering of Document-snapshots. Assume for the moment that the prover already supports document edits, and knows how to process partial theory content, while producing feedback of formal checking via messages (plain text output or markup over the original sources). How does the editor render that continuous flow of information in its single physical instance of the GUI, without getting blocked by the prover?

The classic approach of Proof General [3] makes a tight loop around each prover command, and synchronizes a full protocol round-trip for each transaction. This often leads to situations where the editor is non-reactive, not to speak of the "locked region" of processed text where the user is not allowed to edit.

PIDE avoids blocking by a notion of *document snapshot* and convergence of content, instead of synchronization. See also figure 2.

The editor and the prover are independent processes that exchange information monotonically: each side uses its present knowledge to proceed, and propagates results to its counterpart. The front-end ultimately needs to render editor buffers (painting text with colors, squiggly underlines etc.) by interpreting the source text with its accumulated markup. The flow of information is as follows:

1. editor knows text T, markup M, and edits ΔT (produced by user)
2. apply edits: $T' = T + \Delta T$ (*immediately* in the editor)
3. formal processing of T': ΔM after time Δt (*eventually* in the prover)
4. immediate approximation: $\tilde{M} = \mathit{revert}\ \Delta T; \mathit{retrieve}\ M; \mathit{convert}\ \Delta T$
5. eventual convergence after time Δt: $M' = M + \Delta M$

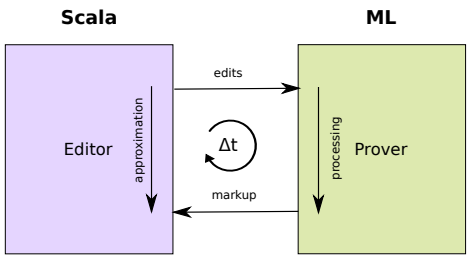

Fig. 2. Approximation and convergence of markup produced by proof processing.

This means the editor is streaming edits as document updates towards the prover, which processes them eventually to give feedback via semantic markup. Without waiting for the prover, the document snapshot of the editor uses the edit-distance over the text to stretch or shrink an old version of markup into the space of the new text: *revert* transforms text positions to move before edits, and *convert* to move after edits. The PIDE Scala API allows to make a document snapshot at any time: it remains an immutable value, while other document processing continues in parallel. Thus GUI painting works undisturbed.

A document snapshot is *outdated* if the edit-distance is non-empty or there is a pending "command-exec assignment" by the prover (see also §3). Text shown in an outdated situation is painted in Isabelle/jEdit with grey background. The user typically sees that for brief instances of time, while edits are passed through the PIDE protocol phases. Longer periods of "editor grey-out" (without blocking) may happen in practice, when the prover is unreactive due to heavy load of ML threads or during garbage collection of the ML run-time system.

Decoupling the editor and prover in asynchronous PIDE document operations provides sufficient freedom to schedule heavy-duty proof checking tasks. The prover is enabled to orchestrate parallel proof processing [17] and additional diagnostic tools (see also §5). The concrete implementation requires a fair amount of performance tuning and adjustment of real-time parameters and delays, to make the user-experience smooth on a given range of hardware: in Isabelle2013-2 this is done for high-end laptops or work-stations with 2–8 cores. Continuous proof processing becomes a highly interactive computer game and thus introduces genuinely new challenges to ITP. Even the graphics performance of the underlying OS platform becomes a relevant factor, since many GUI details need to be updated frequently as the editor or prover changes its state.

3 Document Content

The subsequent description of document content refers to data structures managed by a PIDE-compliant prover like Isabelle. This defines declarative outlines and administrative information for eventual processing: part of that is reported

to the front-end as "command-exec assignment". Further details of actual execution management are the sole responsibility of the prover (see §4).

3.1 Prover Command Transactions

The theory and proof language of Isabelle and other LCF-style systems consists of a sequence of *commands*. This accidental structure can be explained historically and is *not* challenged here. Existing implementations assume that format, and PIDE aims to minimize the requirement to rework old tools.

A theory consists of some text that is partitioned into a sequence of *command spans* as in Proof General [3]. The Isar proof language [18, §6] demonstrates that linearity is no loss of generality: block structure may be represented by a depth-first traversal of the intended tree, using an explicit stack within the proof state. Note that the superficial linearity of proof documents is in contrast to Mizar articles [18, §2], and most regular programming languages, but PIDE is focused on LCF-style proof assistants.

The internal structure of command transactions with distinctive phases of *read*, *eval*, *print* is discussed further in [16]. For PIDE proof documents, these phases are elaborated and specifically managed by the system. At first approximation, a command transaction is a partial function tr from some toplevel state st_0 to st_1, with sequential composition of its phases as shown in figure 3.

$$st_0 \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} st_1 \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} st_2 \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} st_3 \cdots$$

Fig. 3. Sequential scheduling of commands: *read*, *eval*, *print* loop.

Looking more closely, the separate phases may be characterized by their relation to the toplevel state that is manipulated here:

$tr\ st_0 =$
  let $eval = read\ ()$ **in** — *read* does not require st_0
  let $st_1 = eval\ st_0$ **in** — main transition $st_0 \longrightarrow st_1$
  let $() = print\ st_1$ **in** st_1 — *print* does not change st_1

For PIDE, the actual work done in *read*, *eval*, *print* does not matter, e.g. commands may put extra syntactic analysis or diagnostic output into *eval*. The key requirement is that all operations are *purely functional* wrt. the toplevel state seen as *immutable value*, optionally with *observable output* via managed message channels (not physical `stdout`). These assumptions are violated by traditional LCF-style provers, including classic Isabelle in the 1990s, so this is an important starting point for reforms of other proof assistants. Command transactions need to be clearly isolated, and operate efficiently in a timeless and stateless manner.

A simple document-model would merely maintain a partially evaluated sequence of command transactions, and interleave its continued editing and execution. This could even work within a sequential prover process, with asynchronous

signals for new input, but without multi-threading. On the other hand, explicit threads can simplify the implementation and provide additional potential for performance. In fact, the parallel aspect of proof processing [17] turns out relatively simple, compared to the extra entropy and hazards of user-interaction.

3.2 Document Nodes

Proof documents have additional structure that helps to organize continuous processing efficiently, to provide quick feedback to the user during editing.

The **global structure** is that of a *theory graph*, which happens to be acyclic due to the foundational order of theory content in LCF-style provers. The node *dependencies* are given as a list of imports, cf. the syntax of Isabelle theory headers "**theory** A **imports** $B_1 \ldots B_n$". Parallel traversal of DAGs is a starting point to gain performance and scalability for big theory libraries.

The **local structure** of each document node consists of command *entries*, *perspective* and *overlays*, which are described below.

Node entries are given as a linear sequence of commands (§3.1), but each command span is *interned* and represented by a unique *command-id*. There is a global mapping from *command-id* to the corresponding command transaction, which is updated before applying document edits. This indirection avoids redundant invocation of *read* in incremental processing of evolving document versions, since command positions change more often than the content of command spans.

A *command-id* essentially refers to some function tr on the toplevel state. In different document versions it may be applied in different situations. A particular command application $tr\ st_0$ is called *exec* and identified via some *exec-id*, which serves as a physical transaction identifier of the running command. The *exec-id* identifies both the command execution and its result state $st_1 = tr\ st_0$, including observable output (prover messages are always decorated by the *exec-id*).

For a given document version, the *command-exec assignment* relates each *command-id* to a list of *exec-ids*. An empty list means the command is *unassigned* and the prover will not attempt to execute it. A non-empty list refers to the main *eval* as head, and additional *prints* as tail. *Coincidence* of execs means that in the overall document history, a *command-id* has the same *exec-id* in multiple versions. This re-use of old execution fragments in new versions typically happens, when a shared prefix of commands is unaffected by edits applied elsewhere (see also figure 4). The prover is free to execute commands from different document versions, independently of the one displayed by the editor.

The command-exec assignment is vital for the editor to determine which exec results belong to which command in a particular document version, in order to display the content to the user. Whenever this information is updated on the prover side, the editor needs to be informed about it. Edits that are not yet acknowledged by the corresponding assignment lead to an outdated document snapshot (§2). This intermediate situation is now more often visible in Isabelle/PIDE, because execution is strictly monotonic: while the document is updated the prover continues running undisturbed, so the PIDE protocol thread

needs to compete with ML worker threads. In the past, execution was canceled and restarted, but this is in conflict with long-running *eval* and *prints* (cf. §5).

Node perspective specifies *visible* and *required* commands syntactically within the document. The set of visible commands is typically determined by open text windows of the editor. Required commands may be ticked separately by some GUI panel (Isabelle/jEdit does that only for document nodes, meaning the last command entry of a theory.) Visible commands are particularly interesting for the user and need full execution of *read-eval-print*. Required commands are only needed to get there: *read-eval* is sufficient to produce the subsequent toplevel state. The set of required commands is implicitly completed wrt. the transitive closure of node imports and the precedence relation of command entries.

Commands that are neither visible nor required are left unassigned, and thus remain unevaluated. There is usually a long tail-end of the overall document that is presently unassigned. Likewise, there is a long import chain, where the previous assignment is not changed, because edits are typically local to the visible part. This differentiation of document content by means of the perspective is important for scalability, in order to support continuous processing of hundreds of theory nodes, each with thousands of command entries.

Node overlays assign *print functions* (with arguments) to existing command entries within the document. The idea is to analyze the toplevel state at the point after *eval* via additional *prints*. Document overlays may be added or deleted, without changing the underlying sequence of toplevel states.

The prover also maintains a global table of *implicit print functions* (with empty arguments), which are added automatically to any visible command in the current perspective. This may be understood as a mechanism for default overlays for all commands seen in the document.

Given $st_1 = \mathit{eval}\ st_0$, each print function application $\mathit{print}\ st_1$ is identified by a separate *exec-id*. The observable result of an assigned command is the union of results from the *exec-ids* of its *eval* and all its *prints*. This union is formed by the editor whenever it retrieves information from a document snapshot (§2). It may combine *eval* and *prints* stemming from different document versions due to exec coincidence within the ML process.

3.3 Document Edits

Edits emerge in the editor by inserting or removing intervals of plain text, but these are preprocessed to operate on command entries with corresponding *command-ids* (§3.2). Changes of document node dependencies, perspective, and overlays are represented as edits, too. The PIDE document-model provides one key operation *Document.update* to turn a given document version into a new one, where the edits are syntactically represented as algebraic datatype:

datatype *edit* = *Dependencies* | *Entries* | *Perspective* | *Overlays*
val *Document.update*: *version-id* → *version-id* →
(*node* × *edit*) *list* → *state* → (*command-id* × *exec-id list*) *list* × *state*

Type *edit* is given in stylized form above: its constructors take arguments, e.g. *Entries* the commands that are inserted or removed. *Document.update* operates on a "big" document *state*, which maintains all accessible versions. This must not be confused with a "small" toplevel state *st* for single commands.

Edits are relative to given document nodes, and can happen simultaneously when the user opens several theory files or the system completes imports transitively. While the editor usually shows a few text windows only, the document-model always works on the whole theory library behind it.

The update assignment (*command-id* × *exec-id list*) *list* is conservative: the list mentions only those *command-ids* that change. The result is reported to the editor to acknowledge the *Document.update*: until that protocol message arrives, the editor re-uses the assignment of the old version, and marks any document snapshots derived from it as outdated (§2).

The prover maintains a command exec assignment for each document version, depending on the visible and required commands of the perspective, and given node overlays: *exec-ids* for *eval* and *prints* are assigned as required. Old assignments are preserved on a common prefix that is not affected by the edits, as illustrated in figure 4. Here st_2 is the last common exec of the old versus new version; subsequent execs are removed and new ones assigned.

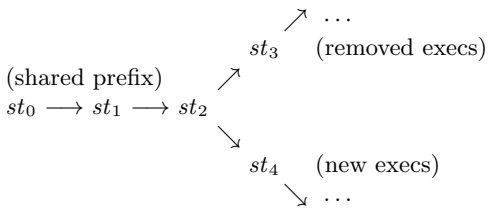

Fig. 4. Update of command-exec assignment, with shared prefix between versions.

The precise manner of exec assignment is up to the prover: it can use further information about old versions and more structure of the command language. Earlier observations about the inherent structure of proof documents for parallelization [17, §2.2] apply here as well, but the additional aspect of incremental editing introduces extra complexity. Current Isabelle/PIDE is still based on the simple linear model explained above, with some refinements on how the *read-eval-print* phases of each command transaction is scheduled. This allows internal *forks* of *eval* and independent *prints* as illustrated in figure 5.

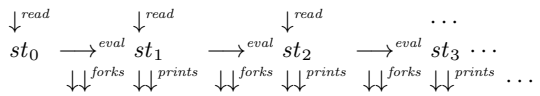

Fig. 5. Parallel scheduling of commands: *read*, *eval*, with multiple *forks* and *prints*.

4 Execution Management

There is open-ended potential for sophistication of execution management, to improve parallel performance and reactivity. The subsequent explanations give some ideas about current Isabelle/PIDE, with its recently introduced ML module `Execution`. Its managed *Execution.fork* now supersedes the earlier approach of goal forks [17, §3.3].

Prerequisite: Future Values in Isabelle/ML. The underlying abstraction for parallel ML programming [17, §3.1] is the polymorphic type α *future* with operations *fork*: $(unit \to \alpha) \to \alpha$ *future* and *join*: α *future* $\to \alpha$ to manage evaluation of functional expressions, with optional *cancel*: α *future* $\to$ *unit*. Moreover, *promise*: *unit* $\to \alpha$ *future* and *fulfill*: α *future* $\to \alpha \to$ *unit* allow to create an open slot for some future result that is closed by external means.

Futures are common folklore in functional programming, but Isabelle/ML implements particular policies that have emerged over several years in pursuit of parallel theorem proving: **strict evaluation** (spontaneous execution via thread-pool), **synchronous exceptions** (propagation within nested task groups), **asynchronous interrupts** (cancellation and signaling of tasks), **nested task groups** (block structure of parallel program), and explicit **dependencies**.

Hypothetical Execution. Each document version is associated with an implicit execution process. After document update, the old execution needs to be turned into a new one, without disturbing active tasks. To this end, Isabelle/PIDE maintains a lazy *execution outline*: chains of commands are composed with their *eval* and *prints* as one big expression, which mathematically determines all prover results beforehand (with corresponding *exec-ids*).

The scheduling diagram of figure 5 illustrates the local structure of this expression: each arrow corresponds to some function application. The global structure has two further dimensions: the DAG of theory nodes and the version history, so many such filaments of *read-eval-print* exist simultaneously.

Since *Document.update* (§3.3) merely performs hypothetical execution, by manipulating a symbolic expression that consists of lazy memo cells, it is able to produce the new assignment quickly and report it back to the editor.

Execution Frontiers. Actual execution is an ongoing process of parallel tasks that force their way through the lazy execution outline. After each document update, the latest document version is associated with a fresh execution, but that needs to coexist with older executions with remaining active tasks.

To prevent conflicting attempts to force these lazy values, the PIDE ML module `Execution` ensures that at most one execution is formally *running*, in the sense defined below. The module manages a separate notion of *execution-id*, with the following operations:

Execution.start: *unit* $\to$ *execution-id*
Execution.discontinue: *unit* $\to$ *unit*
Execution.running: *execution-id* $\to$ *exec-id* $\to$ *bool*

Execution.start () creates a fresh *execution-id* and makes it the currently running one. *Execution.discontinue* () resets that state: a previously running *execution-id* cannot become running again. *Execution.running execution-id exec-id* requests the exclusive right to explore the given *exec-id*, which is only granted if the *execution-id* is currently running. Moreover, the *exec-id* is registered for management of derived execution forks (see below).

Given a document version, the *execution frontier* is the set of tasks that may explore its execution outline, guarded by invocations of *Execution.running* as shown above. Each PIDE update cycle first invokes *Execution.discontinue*, then updates the document content with its execution outline, and then uses *Execution.start* to obtain a new running *execution-id*. Finally, the exploration tasks are forked as ML futures, with the old execution frontier as tasks dependencies.

Thus the new execution frontier is semantically appended to the old one: the old frontier cannot explore new transactions and finishes eventually (or diverges), afterwards the new execution continues without conflict. This approach enables strictly monotonic execution management: running tasks within the document execution are never canceled; only those tasks are terminated that become inaccessible in the new version (removed commands etc.).

Execution forks. The running futures of the execution frontier work on command transactions that are presently accessible, guarded by *Execution.running*. This provides a central checkpoint to control access to individual execs within the given execution outline. After having passed *Execution.running* successfully, further future tasks may be managed as follows:

Execution.fork: *exec-id* $\rightarrow$ ($\alpha \rightarrow$ *unit*) $\rightarrow$ α *future*
Execution.cancel: *exec-id* $\rightarrow$ *unit*

Here the *exec-id* serves as a general handle to arbitrary future forks within that execution context: it is associated with some future task group for cumulative cancellation. Execution management ensures *strict* results: forks need to be joined eventually, and ML exceptions raised in that attempt are accounted to the transaction context. Thus a command transaction may "fail late" due to pending execution forks, even though its *eval* phase has finished superficially, and subsequent commands are already proceeding from its toplevel state.

Execution.fork provides the main programming interface to *forks* of figure 5. The primary application are goal forks in the sense of [17, §3.3], which has been retrofitted into the new execution concept. Note that the Isabelle/PIDE document model still lacks the structural proof forking of batch mode: interactive goal forks are limited to terminal **by** steps (where Isar proofs spend most of the time) or derived definitions with internal proofs like **datatype**, **inductive**, **fun**.

Moreover, *Execution.fork* is now used implicitly for diagnostic commands, which are marked syntactically to be state preserving, and can thus be forked immediately in the main evaluation sequence. Such commands are identity functions on the toplevel state, with observable output, and the potential to fail later. Note that **sledgehammer** is such a diagnostic command as well, and several copies put into a theory already causes parallel execution.

5 Asynchronous Print Functions

Diagnostic command output may happen in the main *eval* phase, but this has the disadvantage that linear editing (§3.3) reassigns intermediate execs and thus disrupts the evaluation sequence. PIDE document updates could be made smarter, but it turns out that separate management of *print* phases over existing commands is simpler and more flexible. Further observations indicate that print functions deserve special attention:

- Cumulative *print* operations consume more space and time than *eval*: proof state output is often large and its printing slower than average proof steps.
- Printing depends on document perspective: text that becomes visible requires additional output, but it can be disposed after becoming invisible.
- Printing may fail or diverge, but it needs to be interruptible to enable the system stopping it.
- Different ways of printing may run in parallel, with specific priorities.

These are notable refinements of the former approach [16, §2.3], which was restricted to one *print* as lazy value that was forked eventually; its execution had to terminate relatively quickly, and the result was always stored persistently.

The current notion of asynchronous print functions allows better management of plain proof state output, and more advanced tools to participate in the continuous document processing. The PIDE ML programming interface accepts various declarative parameters to provide hints for execution management:

Startup delay: extra time to wait, after the print becomes active. This latency reduces waste of CPU cycles when the user continues editing and changes already assigned commands again before printing starts.

Time limit: maximum time spent for a potentially diverging print operation.

Task priority: scheduling parameter for the underlying ML future (for task queue management). Note that this is not a thread priority: an already running task of low priority is unaffected by later forks of high priority.

Persistence: keep results produced by *print* (including observable output), or delete them when visibility gets lost.

Application (1): Proof State Output. Printing proof states efficiently is less trivial than it seems. Command-line users do not mind to wait fractions of a second to see the result after each command, but continuous document processing in PIDE means that maybe 10–100 commands become visible when opening or scrolling text windows. If printing requires 10–100 ms for each command, it already causes significant slowdown.

Proof states are now printed asynchronously, with the following scheduling parameters: no startup delay, no time limit, high task priority, no persistence.

The absence of delay and the priority means that the *print* phase runs eagerly whenever possible, after its corresponding *eval* has finished. On multiple cores, the ongoing *eval* sequence proceeds concurrently with corresponding

prints, resulting in fairly good performance. On a single core, the system adapts its task scheduling to do the interleaving of *eval* versus high priority *prints* sequentially: this is important for the user to proceed, but results in considerable slowdown. The difference can be seen e.g. in the long unstructured proof scripts of `$ISABELLE_HOME/src/HOL/Hoare_Parallel/OG_Hoare.thy`, setting in jEdit *Plugin Options / Isabelle / General / Threads* to 2 versus 1, restarting proof processing via *File / Reload*, scrolling around etc.

Non-persistence is based on the observation that each individual proof state output is reasonably fast, but its result can be big and needs to be stored in the document model (in Scala). For commands that lose visible perspective, the corresponding *print* is unassigned and the document content eventually disposed by garbage collection. Thus we conserve Scala/JVM space, by investing extra ML time to print again later.

Application (2): Automatically Tried Tools. As explained in the manual [14, §2.7], Isabelle/HOL provides a collection of tools that can prove or disprove goals without user intervention: automated methods (*auto*, *simp*, *blast* etc.), **nitpick**, **quickcheck**, **sledgehammer**, **solve-direct**.

In Isabelle Proof General, such tools run synchronously within regular proof state output, and a tight timeout of 0.5s to guarantee reactivity of the command loop. This limits the possibilities of spontaneous feedback by the prover to relatively light-weight tools like **quickcheck** and **solve-direct**, and even that may cause cumbersome delays in sequential command processing.

In Isabelle/PIDE automatically tried tools are asynchronous print functions, with default parameters like this: startup delay = 1s, time limit = 2s, low task priority, persistence.

Thus tools usually run only after some time of inactivity, and do not compete directly with the main *eval* and high-priority *prints*. Persistence is enabled, since tools usually take a long time to produce small output: nothing on failure (or timeout) or a short message on success. In particular, the often unsuccessful applications are retained and not tried again.

Tool output is marked-up as *information message*, which is rendered in Isabelle/jEdit with a blue information icon and blue squiggles for the corresponding goal command. This is non-intrusive information produced in the background, while the user was pondering the text. Cumulatively, automatically tried tools can consume significant CPU resources, though. For high-end work-stations connected to grid power that is rarely a problem, but small mobile devices on batteries should disable extraneous instrumentation.

Application (3): Query Operations. The idea is to support frequently used and potentially long-running diagnostic commands via explicit GUI components in the editor, for example *Sledgehammer* and *Find theorems* as explained in [14, §2.8,§2.9] (with screenshots and minimal examples).

In such situations, Proof General [3] provides a separate command-line to issue state-preserving commands synchronously: the user first needs to move the prover focus to some point in the text and then wait while the query is running.

In Isabelle/PIDE this is now done via asynchronous print functions with explicit document overlays (§3.2). Arguments are provided by some GUI dialog box: input causes a document update that changes the corresponding overlay; the command position is determined from the current focus in the text.

The asynchronous approach allows the user to input the query and start the operation at any time, while the system schedules the print process to run spontaneously after the command that defines its context is evaluated; afterwards it presents query results as they arrive incrementally. There is also a button to cancel the process (notably for *Sledgehammer*). The transitional states of a pending query are visualized by some "spinning disk" icon (with tooltip).

Isabelle/PIDE provides a hybrid module `Query_Operation` in ML and Scala. The ML side accepts a function that takes a toplevel state with arguments and produces output on some private channel; this interface resembles traditional command-line tools. Likewise, the Scala side works with conventional event-based GUI components, without direct exposure to the timeless and stateless PIDE document model. The implementation of the hybrid `Query_Operation` module takes care of the management of different instances for each GUI view, and keeps the connection to running command execs (for cancellation etc.).

This completes the full round-trip of PIDE concepts: from the sequential and synchronous *read-eval-print* loop that connects the user directly to a single command execution, over an intermediate document model that is detached from particular time and space, leading to simple PIDE APIs that recover the appearance of working directly with some command execution that is connected to physical GUI elements. The benefit of this detour is that the system infrastructure is enabled to manage the details of execution efficiently, for many tools on many CPU cores, instead of asking the user to do this sequentially by hand.

6 Conclusion and Related Work

The Isabelle/PIDE approach combines user interaction and tool integration into a uniform document-model. This enables advanced front-end technology in the style of classic IDEs for mainstream programming languages. It also allows us to integrate interactive or automatic theorem proving tools to help the user composing proof documents. The present paper continues earlier explanations of PIDE concepts [12, 11, 13, 15, 16]. The following improvements are newly introduced in the current generation of Isabelle/PIDE (December 2013):

- strictly monotonic document update: avoid cancellation and restart of running command transactions;
- explicit document execution management;
- support for asynchronous print functions, with various execution policies;
- support for document overlays and query operations, with separate GUI components for input and output.

Related work. Explicit parallelism has been imposed on application developers before, when classic CISC machines became stagnant in the 1990s, and workstation clusters were considered a potential solution. A notable experiment from that time is the *Distributed Larch Prover* [5]: it delegates proof problems to CPU nodes, with a central managing process and some Emacs front-end to organize pending proofs. The report on that early project clearly identities the need to rethink prover front-ends, when the back-end becomes parallel.

Concerning prover front-ends, the main landmark to improve upon raw TTY interaction of proof assistants is *Proof General* by D. Aspinall [3]. It only requires a classic *read-eval-print* loop with annotated prompt and *undo* operation, and thus implements "proof scripting" within the editor. The user can navigate forwards and backwards to move the boundary between the *locked region* of the text that is already checked and the remaining part that is presently edited.

The approach of Proof General was so convincing that it has been duplicated many times, with slightly different technical side-conditions, e.g. in CoqIDE [18, §4] (OCaml/GTK), Matita [2] (OCaml/GTK), Matitaweb [1] (OCaml web server). The great success of Proof General 15 years ago made it difficult to go beyond it. Early attempts by D. Aspinall to formalize its protocol as PGIP and integrate it with Eclipse [4] have never reached a sufficient level of support by proof assistants to become relevant to users. Nonetheless, PGEclipse was an important initiative to point beyond classic TTY and Emacs, into a greater world of IDE frameworks.

Dafny [6] follows a different approach to connect automated theorem proving (Boogie and Z3) with Visual Studio as the IDE. Thus it introduces some genuine user-interaction into a world of automatic SMT solving, bypassing TTY mode. The resulting application resembles Isabelle/jEdit, while the particular proof tools and logical foundations of the proof environment are quite different.

Agora [10] is a recent web-centric approach to document-oriented proof authoring, for various existing back-ends like Coq [18, §4] and Mizar [18, §2]. The main premise of this work is to take the proof assistant *as-is* and to see how much added value can be achieved by wrapping web technology around it. C. Tankink also points beyond classic IDEs, which are in fact already 10–20 years old. More recent movements on IDE design for programming languages integrate old and new ideas of *direct manipulation* of static program text and dynamic execution side-by-side, and a non-linear document-model of source snippets. A notable project is http://www.chris-granger.com/lighttable, which is implemented in Clojure and works for Clojure, Javascript, and Python.

Incidently, interactive proof checking has been based on direct access to proof states from early on, and PIDE already provides substantial support to manage incremental execution and continuous checking of proof-documents. So further alignments with such newer IDE approaches would be a rather obvious continuation of what has been achieved so far, but the ITP community also requires time to get acquainted even with the classic IDE model seen in Isabelle/jEdit.

References

[1] Asperti, A., Ricciotti, W.: A web interface for Matita. In Jeuring, J., et al., eds.: Intelligent Computer Mathematics (CICM 2012). Volume 7362 of LNCS., Springer (2012)

[2] Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Journal of Automated Reasoning **39**(2) (2007)

[3] Aspinall, D.: Proof General: A generic tool for proof development. In Graf, S., Schwartzbach, M., eds.: European Joint Conferences on Theory and Practice of Software (ETAPS). Volume 1785 of LNCS., Springer (2000)

[4] Aspinall, D., Lüth, C., Winterstein, D.: A framework for interactive proof. In Kauers, M., Kerber, M., Miner, R., Windsteiger, W., eds.: Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007). Volume 4573 of LNAI., Springer (2007)

[5] Kapur, D., Vandevoorde, M.T.: DLP: A paradigm for parallel interactive theorem proving (1996)

[6] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). Volume 6355 of LNCS., Springer (2010)

[7] Lüth, C., Ring, M.: A web interface for Isabelle: The next generation. In Carette, J., et al., eds.: Intelligent Computer Mathematics (CICM 2013). Volume 7961 of LNCS., Springer (2013)

[8] Lüth, C., Ring, M.: Collaborative interactive theorem proving with Clide. In Klein, G., Gamboa, R., eds.: Interactive Theorem Proving (ITP 2014). LNCS, Springer (2014)

[9] Odersky, M., et al.: An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne (2004)

[10] Tankink, C.: Documentation and Formal Mathematics — Web Technology meets Theorem Proving. PhD thesis, Radboud University Nijmegen (2013)

[11] Wenzel, M.: Isabelle as document-oriented proof assistant. In Davenport, J.H., et al., eds.: Conference on Intelligent Computer Mathematics (CICM 2011). Volume 6824 of LNAI., Springer (2011)

[12] Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In Coen, C.S., Aspinall, D., eds.: User Interfaces for Theorem Provers (UITP 2010). ENTCS (July 2010)

[13] Wenzel, M.: Isabelle/jEdit — a Prover IDE within the PIDE framework. In Jeuring, J., et al., eds.: Conference on Intelligent Computer Mathematics (CICM 2012). Volume 7362 of LNAI., Springer (2012)

[14] Wenzel, M.: Isabelle/jEdit. Part of Isabelle distribution. (December 2013) http://isabelle.in.tum.de/website-Isabelle2013-2/dist/Isabelle2013-2/doc/jedit.pdf.

[15] Wenzel, M.: PIDE as front-end technology for Coq. http://arxiv.org/abs/1304.6626 (2013)

[16] Wenzel, M.: READ-EVAL-PRINT in parallel and asynchronous proof-checking. In Kaliszyk, C., Lüth, C., eds.: User Interfaces for Theorem Provers (UITP 2012). Volume 118 of EPTCS. (2013)

[17] Wenzel, M.: Shared-memory multiprocessing for interactive theorem proving. In Blazy, S., Paulin-Mohring, C., Pichardie, D., eds.: Interactive Theorem Proving (ITP 2013). Volume 7998 of Lecture Notes in Computer Science., Springer (2013)

[18] Wiedijk, F., ed.: The Seventeen Provers of the World. Volume 3600 of LNAI. Springer (2006)