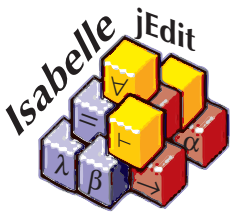

Isabelle/jEdit

Makarius Wenzel

5 December 2013

Abstract

Isabelle/jEdit is a fully-featured Prover IDE, based on Isabelle/Scala and the jEdit text editor. This document provides an overview of general principles and its main IDE functionality.

> *Isabelle's user interface is no advance over LCF's, which is widely condemned as "user-unfriendly": hard to use, bewildering to beginners. Hence the interest in proof editors, where a proof can be constructed and modified rule-by-rule using windows, mouse, and menus. But Edinburgh LCF was invented because real proofs require millions of inferences. Sophisticated tools — rules, tactics and tacticals, the language ML, the logics themselves — are hard to learn, yet they are essential. We may demand a mouse, but we need better education and training.*
>
> Lawrence C. Paulson, "Isabelle: The Next 700 Theorem Provers"

Acknowledgements

Research and implementation of concepts around PIDE and Isabelle/jEdit has started around 2008 and was kindly supported by:

- TU München http://www.in.tum.de
- BMBF http://www.bmbf.de
- Université Paris-Sud http://www.u-psud.fr
- Digiteo http://www.digiteo.fr
- ANR http://www.agence-nationale-recherche.fr

Contents

List of Figures

Chapter 1

Introduction

1.1 Concepts and terminology

Isabelle/jEdit is a Prover IDE that integrates *parallel proof checking* [5, 10] with *asynchronous user interaction* [6, 9], based on a document-oriented approach to *continuous proof processing* [7, 8]. Many concepts and system components are fit together in order to make this work. The main building blocks are as follows.

PIDE is a general framework for Prover IDEs based on Isabelle/Scala. It is built around a concept of parallel and asynchronous document processing, which is supported natively by the parallel proof engine that is implemented in Isabelle/ML. The prover discontinues the traditional TTY-based command loop, and supports direct editing of formal source text with rich formal markup for GUI rendering.

Isabelle/ML is the implementation and extension language of Isabelle, see also [3]. It is integrated into the logical context of Isabelle/Isar and allows to manipulate logical entities directly. Arbitrary add-on tools may be implemented for object-logics such as Isabelle/HOL.

Isabelle/Scala is the system programming language of Isabelle. It extends the pure logical environment of Isabelle/ML towards the "real world" of graphical user interfaces, text editors, IDE frameworks, web services etc. Special infrastructure allows to transfer algebraic datatypes and formatted text easily between ML and Scala, using asynchronous protocol commands.

jEdit is a sophisticated text editor implemented in Java.[1] It is easily extensible by plugins written in languages that work on the JVM, e.g. Scala[2].

[1] http://www.jedit.org
[2] http://www.scala-lang.org/

Isabelle/jEdit is the main example application of the PIDE framework and the default user-interface for Isabelle. It targets both beginners and experts. Technically, Isabelle/jEdit combines a slightly modified version of the jEdit code base with a special plugin for Isabelle, integrated as standalone application for the main operating system platforms: Linux, Windows, Mac OS X.

The subtle differences of Isabelle/ML versus Standard ML, Isabelle/Scala versus Scala, Isabelle/jEdit versus jEdit need to be taken into account when discussing any of these PIDE building blocks in public forums, mailing lists, or even scientific publications.

1.2 The Isabelle/jEdit Prover IDE

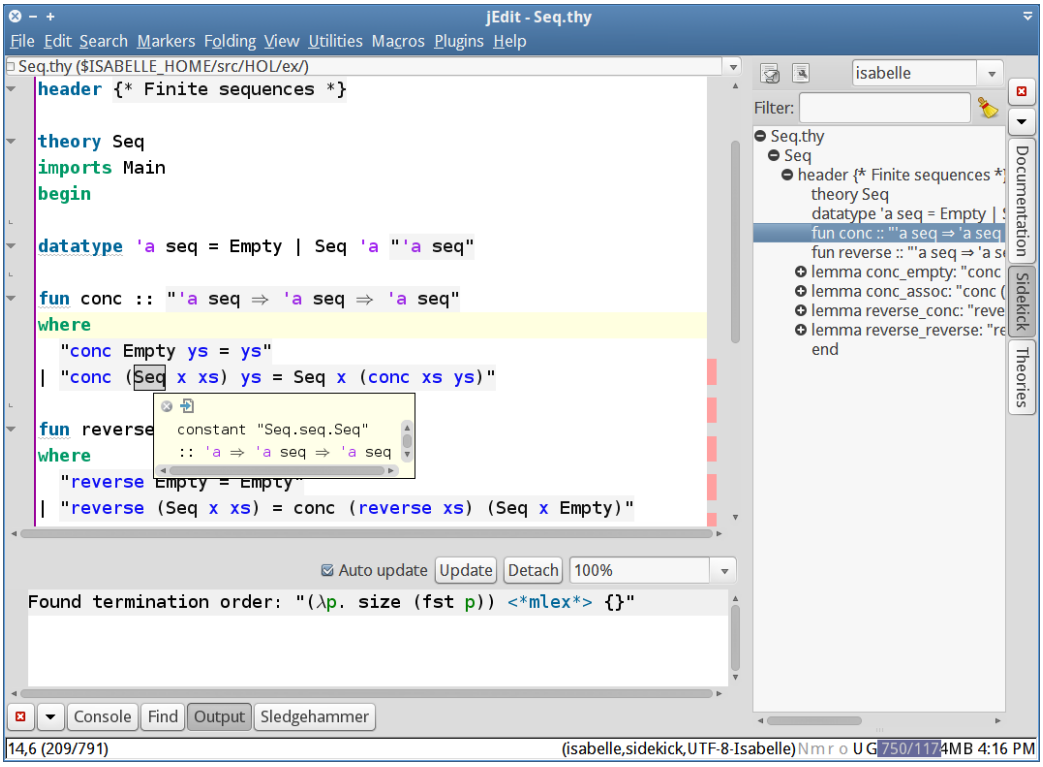

Figure 1.1: The Isabelle/jEdit Prover IDE

Isabelle/jEdit (figure 1.1) consists of some plugins for the well-known jEdit text editor http://www.jedit.org, according to the following principles.

- The original jEdit look-and-feel is generally preserved, although some default properties are changed to accommodate Isabelle (e.g. the text area font).
- Formal Isabelle/Isar text is checked asynchronously while editing. The user is in full command of the editor, and the prover refrains from locking portions of the buffer.
- Prover feedback works via colors, boxes, squiggly underline, hyperlinks, popup windows, icons, clickable output — all based on semantic markup produced by Isabelle in the background.
- Using the mouse together with the modifier key `CONTROL` (Linux, Windows) or `COMMAND` (Mac OS X) exposes additional formal content via tooltips and/or hyperlinks.
- Formal output (in popups etc.) may be explored recursively, using the same techniques as in the editor source buffer.
- Additional panels (e.g. *Output*, *Symbols*) are organized by the Dockable Window Manager of jEdit, which also allows multiple floating instances of each window class.
- The prover process and source files are managed on the editor side. The prover operates on timeless and stateless document content as provided via Isabelle/Scala.
- Plugin options of jEdit (for the *Isabelle* plugin) give access to a selection of Isabelle/Scala options and its persistent preferences, usually with immediate effect on the prover back-end or editor front-end.
- The logic image of the prover session may be specified within Isabelle/jEdit. The new image is provided automatically by the Isabelle build tool after restart of the application.

1.2.1 Documentation

Regular jEdit documentation is accessible via its `Help` menu or `F1` keyboard shortcut. This includes a full *User's Guide* and *Frequently Asked Questions* for this sophisticated text editor. The same can be browsed without the technical restrictions of the built-in Java HTML viewer here: http://www.jedit.org/index.php?page=docs (potentially for a different version of jEdit).

Most of this information about jEdit is relevant for Isabelle/jEdit as well, but one needs to keep in mind that defaults sometimes differ, and the official jEdit documentation does not know about the Isabelle plugin with its special support for theory editing.

1.2.2 Plugins

The *Plugin Manager* of jEdit allows to augment editor functionality by JVM modules (jars) that are provided by the central plugin repository, which is accessible via various mirror sites.

Connecting to the plugin server infrastructure of the jEdit project allows to update bundled plugins or to add further functionality. This needs to be done with the usual care for such an open bazaar of contributions. Arbitrary combinations of add-on features are apt to cause problems. It is advisable to start with the default configuration of Isabelle/jEdit and develop some understanding how it is supposed to work, before loading additional plugins at a grand scale.

The main *Isabelle* plugin is an integral part of Isabelle/jEdit and needs to remain active at all times! A few additional plugins are bundled with Isabelle/jEdit for convenience or out of necessity, notably *Console* with its Isabelle/Scala sub-plugin and *SideKick* with some Isabelle-specific parsers for document tree structure. The *ErrorList* plugin is required by *SideKick*, but not used specifically in Isabelle/jEdit.

1.2.3 Options

Both jEdit and Isabelle have distinctive management of persistent options.

Regular jEdit options are accessible via the dialog for *Plugins / Plugin Options*, which is also accessible via *Utilities / Global Options*. Changed properties are stored in `$ISABELLE_HOME_USER/jedit/properties`. In contrast, Isabelle system options are managed by Isabelle/Scala and changed values stored in `$ISABELLE_HOME_USER/etc/preferences`, independently of the jEdit properties. See also [11], especially the coverage of sessions and command-line tools like `isabelle build` or `isabelle options`.

Those Isabelle options that are declared as **public** are configurable in jEdit via *Plugin Options / Isabelle / General*. Moreover, there are various options for rendering of document content, which are configurable via *Plugin Options / Isabelle / Rendering*. Thus *Plugin Options / Isabelle* in jEdit provides a view on a subset of Isabelle system options. Note that some of these options

affect general parameters that are relevant outside Isabelle/jEdit as well, e.g. `threads` or `parallel_proofs` for the Isabelle build tool [11].

All options are loaded on startup and saved on shutdown of Isabelle/jEdit. Editing the machine-generated `$ISABELLE_HOME_USER/jedit/properties` or `$ISABELLE_HOME_USER/etc/preferences` manually while the application is running is likely to cause surprise due to lost update!

1.2.4 Keymaps

Keyboard shortcuts used to be managed as jEdit properties in the past, but recent versions (2013) have a separate concept of *keymap* that is configurable via *Global Options / Shortcuts*. The `imported` keymap is derived from the initial environment of properties that is available at the first start of the editor; afterwards the keymap file takes precedence.

This is relevant for Isabelle/jEdit due to various fine-tuning of default properties, and additional keyboard shortcuts for Isabelle specific functionality. Users may change their keymap later, but need to copy Isabelle-specific key bindings manually (see also `$ISABELLE_HOME_USER/jedit/keymaps`).

1.2.5 Look-and-feel

jEdit is a Java/AWT/Swing application with some ambition to support "native" look-and-feel on all platforms, within the limits of what Oracle as Java provider and major operating system distributors allow (see also §4).

Isabelle/jEdit enables platform-specific look-and-feel by default as follows:

Linux The platform-independent *Nimbus* is used by default.

GTK+ works under the side-condition that the overall GTK theme is selected in a Swing-friendly way.[3]

Windows Regular *Windows* is used by default, but *Windows Classic* also works.

Mac OS X Regular *Mac OS X* is used by default.

Moreover the bundled *MacOSX* plugin provides various functions that are expected from applications on that particular platform: quit from

[3]GTK support in Java/Swing was once marketed aggressively by Sun, but never quite finished, and is today (2013) lagging a bit behind further development of Swing and GTK. The graphics rendering performance can be worse than for other Swing look-and-feels.

menu or dock, preferences menu, drag-and-drop of text files on the application, full-screen mode for main editor windows etc.

Users may experiment with different look-and-feels, but need to keep in mind that this extra variance of GUI functionality is unlikely to work in arbitrary combinations. The platform-independent *Nimbus* and *Metal* should always work. The historic *CDE/Motif* is better avoided.

After changing the look-and-feel in *Global Options / Appearance*, it is advisable to restart Isabelle/jEdit in order to take full effect.

Chapter 2

Prover IDE functionality

2.1 File-system access

File specifications in jEdit follow various formats and conventions according to *Virtual File Systems*, which may be also provided by additional plugins. This allows to access remote files via the `http:` protocol prefix, for example. Isabelle/jEdit attempts to work with the file-system access model of jEdit as far as possible. In particular, theory sources are passed directly from the editor to the prover, without indirection via files.

Despite the flexibility of URLs in jEdit, local files are particularly important and are accessible without protocol prefix. Here the path notation is that of the Java Virtual Machine on the underlying platform. On Windows the preferred form uses backslashes, but happens to accept Unix/POSIX forward slashes, too. Further differences arise due to drive letters and network shares.

The Java notation for files needs to be distinguished from the one of Isabelle, which uses POSIX notation with forward slashes on *all* platforms.[1] Moreover, environment variables from the Isabelle process may be used freely, e.g. `$ISABELLE_HOME/etc/symbols` or `$POLYML_HOME/README`. There are special shortcuts: `~` for `$USER_HOME` and `~~` for `$ISABELLE_HOME`.

Since jEdit happens to support environment variables within file specifications as well, it is natural to use similar notation within the editor, e.g. in the file-browser. This does not work in full generality, though, due to the bias of jEdit towards platform-specific notation and of Isabelle towards POSIX. Moreover, the Isabelle settings environment is not yet active when starting Isabelle/jEdit via its standard application wrapper (in contrast to `isabelle jedit` run from the command line).

For convenience, Isabelle/jEdit imitates at least `$ISABELLE_HOME` and `$ISABELLE_HOME_USER` within the Java process environment, in order to allow easy access to these important places from the editor.

Moreover note that path specifications in prover input or output usually

[1] Isabelle on Windows uses Cygwin file-system access.

include formal markup that turns it into a hyperlink (see also §2.4). This allows to open the corresponding file in the text editor, independently of the path notation.

2.2 Text buffers and theories

As regular text editor, jEdit maintains a collection of open *text buffers* to store source files; each buffer may be associated with any number of visible *text areas*. Buffers are subject to an *edit mode* that is determined from the file type. Files with extension `.thy` are assigned to the mode *isabelle* and treated specifically.

Isabelle theory files are automatically added to the formal document model of Isabelle/Scala, which maintains a family of versions of all sources for the prover. The *Theories* panel provides an overview of the status of continuous checking of theory sources. Unlike batch sessions [11], theory nodes are identified by full path names; this allows to work with multiple (disjoint) Isabelle sessions simultaneously within the same editor session.

Certain events to open or update buffers with theory files cause Isabelle/jEdit to resolve dependencies of *theory imports*. The system requests to load additional files into editor buffers, in order to be included in the theory document model for further checking. It is also possible to resolve dependencies automatically, depending on *Plugin Options / Isabelle / General / Auto load*.

The open text area views on theory buffers define the visible *perspective* of Isabelle/jEdit. This is taken as a hint for document processing: the prover ensures that those parts of a theory where the user is looking are checked, while other parts that are presently not required are ignored. The perspective is changed by opening or closing text area windows, or scrolling within an editor window.

The *Theories* panel provides some further options to influence the process of continuous checking: it may be switched off globally to restrict the prover to superficial processing of command syntax. It is also possible to indicate theory nodes as *required* for continuous checking: this means such nodes and all their imports are always processed independently of the visibility status (if continuous checking is enabled). Big theory libraries that are marked as required can have significant impact on performance, though.

Formal markup of checked theory content is turned into GUI rendering, based on a standard repertoire known from IDEs for programming languages: colors, icons, highlighting, squiggly underline, tooltips, hyperlinks etc. For outer

syntax of Isabelle/Isar there is some traditional syntax-highlighting via static keyword tables and tokenization within the editor. In contrast, the painting of inner syntax (term language etc.) uses semantic information that is reported dynamically from the logical context. Thus the prover can provide additional markup to help the user to understand the meaning of formal text, and to produce more text with some add-on tools (e.g. information messages by automated provers or disprovers running in the background).

2.3 Prover output

Prover output consists of *markup* and *messages*. Both are directly attached to the corresponding positions in the original source text, and visualized in the text area, e.g. as text colours for free and bound variables, or as squiggly underline for warnings, errors etc. (see also figure 2.1). In the latter case, the corresponding messages are shown by hovering with the mouse over the highlighted text — although in many situations the user should already get some clue by looking at the position of the text highlighting.

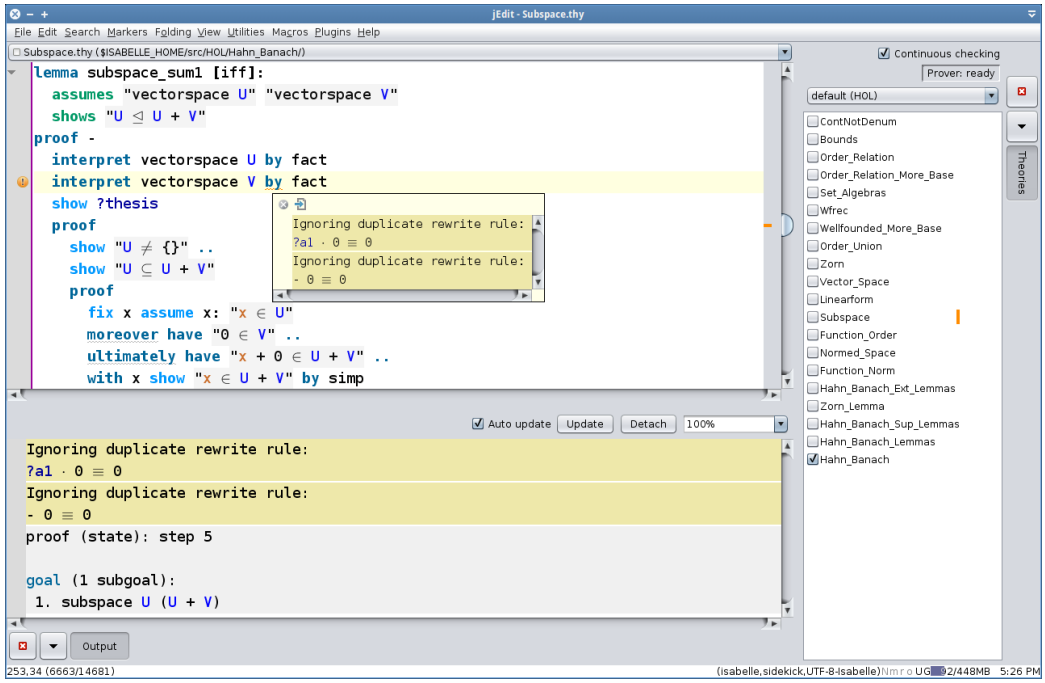

Figure 2.1: Multiple views on prover output: gutter area with icon, text area with popup, overview area, Theories panel, Output panel

The "gutter area" on the left-hand-side of the text area uses icons to provide a summary of the messages within the adjacent line of text. Message priorities are used to prefer errors over warnings, warnings over information messages etc. Plain output is ignored here.

The "overview area" on the right-hand-side of the text area uses similar information to paint small rectangles for the overall status of the whole text buffer. The graphics is scaled to fit the logical buffer length into the given window height. Mouse clicks on the overview area position the cursor approximately to the corresponding line of text in the buffer. Repainting the overview in real-time causes problems with big theories, so it is restricted to part of the text according to *Plugin Options / Isabelle / General / Text Overview Limit* (in characters).

Another course-grained overview is provided by the *Theories* panel, but without direct correspondence to text positions. A double-click on one of the theory entries with their status overview opens the corresponding text buffer, without changing the cursor position.

In addition, the *Output* panel displays prover messages that correspond to a given command, within a separate window.

The cursor position in the presently active text area determines the prover commands whose cumulative message output is appended and shown in that window (in canonical order according to the processing of the command). There are also control elements to modify the update policy of the output wrt. continued editor movements. This is particularly useful with several independent instances of the *Output* panel, which the Dockable Window Manager of jEdit can handle conveniently.

Former users of the old TTY interaction model (e.g. Proof General) might find a separate window for prover messages familiar, but it is important to understand that the main Prover IDE feedback happens elsewhere. It is possible to do meaningful proof editing efficiently, using secondary output windows only rarely.

The main purpose of the output window is to "debug" unclear situations by inspecting internal state of the prover.[2] Consequently, some special messages for *tracing* or *proof state* only appear here, and are not attached to the original source.

In any case, prover messages also contain markup that may be explored recursively via tooltips or hyperlinks (see §2.4), or clicked directly to initiate

[2]In that sense, unstructured tactic scripts depend on continuous debugging with internal state inspection.

certain actions (see §2.7 and §2.8).

2.4 Tooltips and hyperlinks

Formally processed text (prover input or output) contains rich markup information that can be explored further by using the `CONTROL` modifier key on Linux and Windows, or `COMMAND` on Mac OS X. Hovering with the mouse while the modifier is pressed reveals a *tooltip* (grey box over the text with a yellow popup) and/or a *hyperlink* (black rectangle over the text); see also figure 2.2.

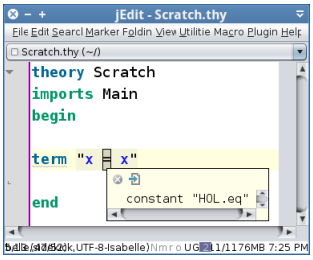

Figure 2.2: Tooltip and hyperlink for some formal entity

Tooltip popups use the same rendering principles as the main text area, and further tooltips and/or hyperlinks may be exposed recursively by the same mechanism; see figure 2.3.

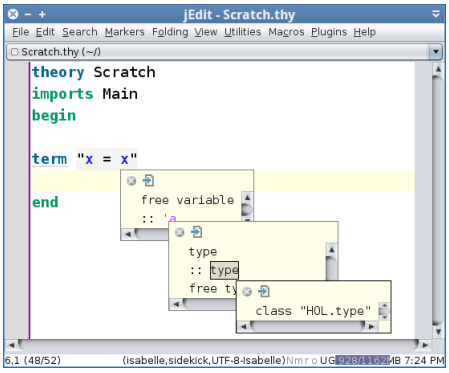

Figure 2.3: Nested tooltips over formal entities

The tooltip popup window provides some controls to *close* or *detach* the window, turning it into a separate *Info* window managed by jEdit. The

ESCAPE key closes *all* popups, which is particularly relevant when nested tooltips are stacking up.

A black rectangle in the text indicates a hyperlink that may be followed by a mouse click (while the CONTROL or COMMAND modifier key is still pressed). Presently (Isabelle2013-2) there is no systematic navigation within the editor to return to the original location.

Also note that the link target may be a file that is itself not subject to formal document processing of the editor session and thus prevents further exploration: the chain of hyperlinks may end in some source file of the underlying logic image, or within the Isabelle/ML bootstrap sources of Isabelle/Pure.

2.5 Text completion

Completion tables are determined statically from the "outer syntax" of the underlying edit mode (for theory files this is the syntax of Isar commands), and specifications of Isabelle symbols (see also §2.6).

Symbols are completed in backslashed forms, e.g. \forall or \<forall> that both produce the Isabelle symbol ∀ in its Unicode rendering.[3] Alternatively, symbol abbreviations may be used as specified in $ISABELLE_HOME/etc/symbols.

Completion popups are required in situations of ambiguous completion results or where explicit confirmation is demanded before inserting completed text into the buffer.

The popup is some minimally invasive GUI component over the text area. It interprets special keys TAB, ESCAPE, UP, DOWN, PAGE_UP, PAGE_DOWN, but all other key events are passed to the underlying text area. This allows to ignore unwanted completions most of the time and continue typing quickly.

The meaning of special keys is as follows:

key	action
TAB	select completion
ESCAPE	dismiss popup
UP	move up one item
DOWN	move down one item
PAGE_UP	move up one page of items
PAGE_DOWN	move down one page of items

[3]The extra backslash avoids overlap with keywords of the buffer syntax, and allows to produce Isabelle symbols robustly in most syntactic contexts.

Movement within the popup is only active for multiple items. Otherwise the corresponding key event retains its standard meaning within the underlying text area.

Explicit completion is triggered by the keyboard shortcut `C+b` (action `isabelle.complete`). This overrides the original jEdit binding for action `complete-word`, but the latter is used as fall-back for non-Isabelle edit modes. It is also possible to restore the original jEdit keyboard mapping of `complete-word` via *Global Options / Shortcuts*.

Replacement text is inserted immediately into the buffer, unless ambiguous results demand an explicit popup.

Implicit completion is triggered by regular keyboard input events during of the editing process in the main jEdit text area (and a few additional text fields like the search criteria of the Find panel, see §2.9). Implicit completion depends on on further side-conditions:

1. The system option `jedit_completion` needs to be enabled (default).
2. Completion of syntax keywords requires at least 3 relevant characters in the text.
3. The system option `jedit_completion_delay` determines an additional delay (0.0 by default), before opening a completion popup.
4. The system option `jedit_completion_dismiss_delay` determines an additional delay (0.0 by default), before dismissing an earlier completion popup. A value like 0.1 is occasionally useful to reduce the chance of loosing key strokes when the speed of typing exceeds that of repainting GUI components.
5. The system option `jedit_completion_immediate` (disabled by default) controls whether replacement text should be inserted immediately without popup. This is restricted to Isabelle symbols and their abbreviations (§2.6) — plain keywords always demand a popup for clarity.
6. Completion of symbol abbreviations with only one relevant character in the text always enforces an explicit popup, independently of `jedit_completion_immediate`.

These completion options may be configured in *Plugin Options / Isabelle / General / Completion*. The default is quite moderate in showing occasional popups and refraining from immediate insertion. An additional completion delay of 0.3 seconds will make it even less ambitious.

In contrast, more aggressive completion works via `jedit_completion_delay = 0.0` and `jedit_completion_immediate = true`. Thus the editing process becomes dependent on the system guessing correctly what the user had in mind. It requires some practice (and study of the symbol abbreviation tables) to become productive in this advanced mode.

In any case, unintended completions can be reverted by the regular `undo` operation of jEdit. When editing embedded languages such as ML, it is better to disable either `jedit_completion` or `jedit_completion_immediate` temporarily.

2.6 Isabelle symbols

Isabelle sources consist of *symbols* that extend plain ASCII to allow infinitely many mathematical symbols within the formal sources. This works without depending on particular encodings and varying Unicode standards [7].[4]

For the prover back-end, formal text consists of ASCII characters that are grouped according to some simple rules, e.g. as plain "`a`" or symbolic "`\<alpha>`".

For the editor front-end, a certain subset of symbols is rendered physically via Unicode glyphs, in order to show "`\<alpha>`" as "α", for example. This symbol interpretation is specified by the Isabelle system distribution in `$ISABELLE_HOME/etc/symbols` and may be augmented by the user in `$ISABELLE_HOME_USER/etc/symbols`.

The appendix of [4] gives an overview of the standard interpretation of finitely many symbols from the infinite collection. Uninterpreted symbols are displayed literally, e.g. "`\<foobar>`". Overlap of Unicode characters used in symbol interpretation with informal ones (which might appear e.g. in comments) needs to be avoided! Raw Unicode characters within prover source files should be restricted to informal parts, e.g. to write text in non-latin alphabets in comments.

[4] Raw Unicode characters within formal sources would compromise portability and reliability in the face of changing interpretation of special features of Unicode, such as Combining Characters or Bi-directional Text.

Encoding. Technically, the Unicode view on Isabelle symbols is an *encoding* in jEdit (not in the underlying JVM) that is called `UTF-8-Isabelle`. It is provided by the Isabelle/jEdit plugin and enabled by default for all source files. Sometimes such defaults are reset accidentally, or malformed UTF-8 sequences in the text force jEdit to fall back on a different encoding like `ISO-8859-15`. In that case, verbatim "`\<alpha>`" will be shown in the text buffer instead of its Unicode rendering "α". The jEdit menu operation *File / Reload with Encoding / UTF-8-Isabelle* helps to resolve such problems, potentially after repairing malformed parts of the text.

Font. Correct rendering via Unicode requires a font that contains glyphs for the corresponding codepoints. Most system fonts lack that, so Isabelle/jEdit prefers its own application font `IsabelleText`, which ensures that standard collection of Isabelle symbols are actually seen on the screen (or printer).

Note that a Java/AWT/Swing application can load additional fonts only if they are not installed on the operating system already! Some old version of `IsabelleText` that happens to be provided by the operating system would prevents Isabelle/jEdit from its bundled version. This could lead to missing glyphs (black rectangles), when the system version of `IsabelleText` is older than the application version. This problem can be avoided by refraining to "install" any version of `IsabelleText` in the first place (although it might be occasionally tempting to use the same font in other applications).

Input methods. In principle, Isabelle/jEdit could delegate the problem to produce Isabelle symbols in their Unicode rendering to the underlying operating system and its *input methods*. Regular jEdit also provides various ways to work with *abbreviations* to produce certain non-ASCII characters. Since none of these standard input methods work satisfactorily for the mathematical characters required for Isabelle, various specific Isabelle/jEdit mechanisms are provided.

Here is a summary for practically relevant input methods for Isabelle symbols:

1. The *Symbols* panel with some GUI buttons to insert certain symbols in the text buffer. There are also tooltips to reveal the official Isabelle representation with some additional information about *symbol abbreviations* (see below).

2. Copy / paste from decoded source files: text that is rendered as Unicode already can be re-used to produce further text. This also works between

different applications, e.g. Isabelle/jEdit and some web browser or mail client, as long as the same Unicode view on Isabelle symbols is used uniformly.

3. Copy / paste from prover output within Isabelle/jEdit. The same principles as for text buffers apply, but note that *copy* in secondary Isabelle/jEdit windows works via the keyboard shortcut `C+c`, while jEdit menu actions always refer to the primary text area!

4. Completion provided by Isabelle plugin (see §2.5). Isabelle symbols have a canonical name and optional abbreviations. This can be used with the text completion mechanism of Isabelle/jEdit, to replace a prefix of the actual symbol like `\<lambda>`, or its backslashed name `\lambda`, or its ASCII abbreviation `%` by the Unicode rendering.

   The following table is an extract of the information provided by the standard `$ISABELLE_HOME/etc/symbols` file:

symbol	**backslashed name**	**abbreviation**
λ	`\lambda`	`%`
$\Rightarrow$	`\Rightarrow`	`=>`
$\Longrightarrow$	`\Longrightarrow`	`==>`
$\bigwedge$	`\And`	`!!`
$\equiv$	`\equiv`	`==`
$\forall$	`\forall`	`!`
$\exists$	`\exists`	`?`
$\longrightarrow$	`\longrightarrow`	`-->`
$\wedge$	`\and`	`&`
$\vee$	`\or`	`|`
$\neg$	`\not`	`~`
$\neq$	`\noteq`	`~=`
$\in$	`\in`	`:`
$\notin$	`\notin`	`~:`

   Note that the above abbreviations refer to the input method. The logical notation provides ASCII alternatives that often coincide, but deviate occasionally. This occasionally causes user confusion with very old-fashioned Isabelle source that use ASCII replacement notation like `!` or `ALL` directly in the text.

   On the other hand, coincidence of symbol abbreviations with ASCII replacement syntax syntax helps to update old theory sources via explicit completion (see also `C+b` explained in §2.5).

Control symbols. There are some special control symbols to modify the display style of a single symbol (without nesting). Control symbols may be applied to a region of selected text, either using the *Symbols* panel or keyboard shortcuts or jEdit actions. These editor operations produce a separate control symbol for each symbol in the text, in order to make the whole text appear in a certain style.

style	symbol	shortcut	action
superscript	`\<^sup>`	`C+e UP`	`isabelle.control-sup`
subscript	`\<^sub>`	`C+e DOWN`	`isabelle.control-sub`
bold face	`\<^bold>`	`C+e RIGHT`	`isabelle.control-bold`
reset		`C+e LEFT`	`isabelle.control-reset`

To produce a single control symbol, it is also possible to complete on `\sup`, `\sub`, `\bold` as for regular symbols.

2.7 Automatically tried tools

Continuous document processing works asynchronously in the background. Visible document source that has been evaluated already may get augmented by additional results of *asynchronous print functions*. The canonical example is proof state output, which is always enabled. More heavy-weight print functions may be applied, in order to prove or disprove parts of the formal text by other means.

Isabelle/HOL provides various automatically tried tools that operate on outermost goal statements (e.g. **lemma**, **theorem**), independently of the state of the current proof attempt. They work implicitly without any arguments. Results are output as *information messages*, which are indicated in the text area by blue squiggles and a blue information sign in the gutter (see figure 2.4). The message content may be shown as for other output (see also §2.3). Some tools produce output with *sendback* markup, which means that clicking on certain parts of the output inserts that text into the source in the proper place.

The following Isabelle system options control the behavior of automatically tried tools (see also the jEdit dialog window *Plugin Options / Isabelle / General / Automatically tried tools*):

- `auto_methods` controls automatic use of a combination of standard proof methods (*auto*, *simp*, *blast*, etc.). This corresponds to the Isar command **try0**.

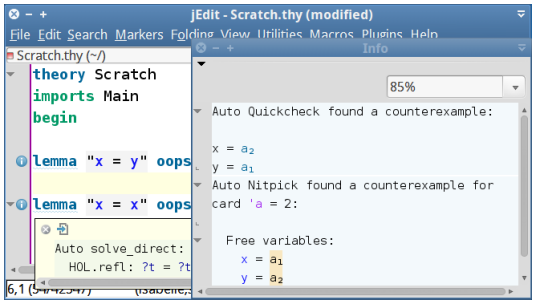

Figure 2.4: Results of automatically tried tools

The tool is disabled by default, since unparameterized invocation of standard proof methods often consumes substantial CPU resources without leading to success.

- `auto_nitpick` controls a slightly reduced version of **nitpick**, which tests for counterexamples using first-order relational logic. See also the Nitpick manual [2].

  This tool is disabled by default, due to the extra overhead of invoking an external Java process for each attempt to disprove a subgoal.

- `auto_quickcheck` controls automatic use of **quickcheck**, which tests for counterexamples using a series of assignments for free variables of a subgoal.

  This tool is *enabled* by default. It requires little overhead, but is a bit weaker than **nitpick**.

- `auto_sledgehammer` controls a significantly reduced version of **sledgehammer**, which attempts to prove a subgoal using external automatic provers. See also the Sledgehammer manual [1].

  This tool is disabled by default, due to the relatively heavy nature of Sledgehammer.

- `auto_solve_direct` controls automatic use of **solve_direct**, which checks whether the current subgoals can be solved directly by an existing theorem. This also helps to detect duplicate lemmas.

  This tool is *enabled* by default.

Invocation of automatically tried tools is subject to some global policies of parallel execution, which may be configured as follows:

- `auto_time_limit` (default 2.0) determines the timeout (in seconds) for each tool execution.
- `auto_time_start` (default 1.0) determines the start delay (in seconds) for automatically tried tools, after the main command evaluation is finished.

Each tool is submitted independently to the pool of parallel execution tasks in Isabelle/ML, using hardwired priorities according to its relative "heaviness". The main stages of evaluation and printing of proof states take precedence, but an already running tool is not canceled and may thus reduce reactivity of proof document processing.

Users should experiment how the available CPU resources (number of cores) are best invested to get additional feedback from prover in the background, by using a selection of weaker or stronger tools.

2.8 Sledgehammer

The *Sledgehammer* panel (figure 2.5) provides a view on some independent execution of the Isar command **sledgehammer**, with process indicator (spinning wheel) and GUI elements for important Sledgehammer arguments and options. Any number of Sledgehammer panels may be active, according to the standard policies of Dockable Window Management in jEdit. Closing such windows also cancels the corresponding prover tasks.

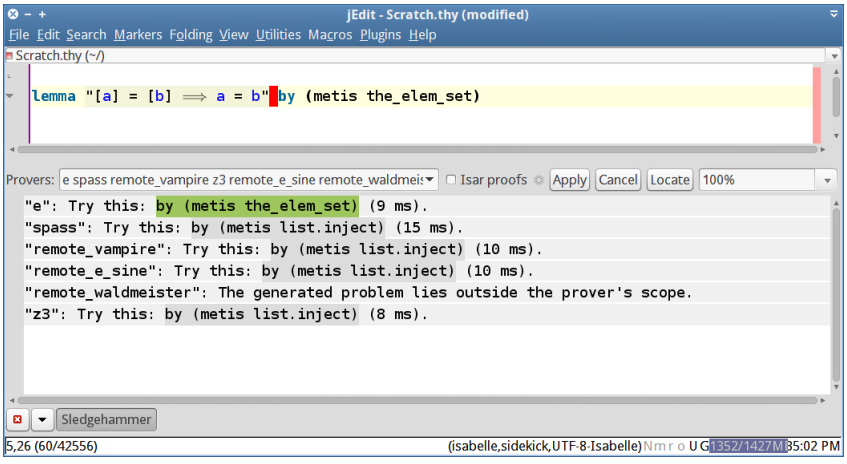

Figure 2.5: An instance of the Sledgehammer panel

The *Apply* button attaches a fresh invocation of **sledgehammer** to the command where the cursor is pointing in the text — this should be some pending proof problem. Further buttons like *Cancel* and *Locate* help to manage the running process.

Results appear incrementally in the output window of the panel. Proposed proof snippets are marked-up as *sendback*, which means a single mouse click inserts the text into a suitable place of the original source. Some manual editing may be required nonetheless, say to remove earlier proof attempts.

2.9 Find theorems

The *Find* panel (figure 2.6) provides an independent view for the Isar command **find_theorems**. The main text field accepts search criteria according to the syntax *thmcriterium* given in [4]. Further options of **find_theorems** are available via GUI elements.

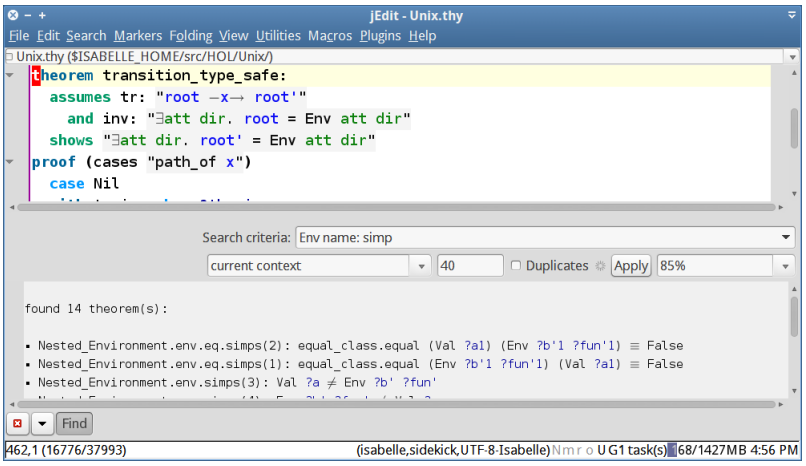

Figure 2.6: An instance of the Find panel

The *Apply* button attaches a fresh invocation of **find_theorems** to the current context of the command where the cursor is pointing in the text, unless an alternative theory context (from the underlying logic image) is specified explicitly.

Chapter 3

Miscellaneous tools

3.1 SideKick

The *SideKick* plugin of jEdit provides some general services to display buffer structure in a tree view.

Isabelle/jEdit provides SideKick parsers for its main mode for theory files, as well as some minor modes for the `NEWS` file, session `ROOT` files, and system `options`.

Moreover, the special SideKick parser `isabelle-markup` provides access to the full (uninterpreted) markup tree of the PIDE document model of the current buffer. This is occasionally useful for informative purposes, but the amount of displayed information might cause problems for large buffers, both for the human and the machine.

3.2 Timing

Managed evaluation of commands within PIDE documents includes timing information, which consists of elapsed (wall-clock) time, CPU time, and GC (garbage collection) time. Note that in a multithreaded system it is difficult to measure execution time precisely: elapsed time is closer to the real requirements of runtime resources than CPU or GC time, which are both subject to influences from the parallel environment that are outside the scope of the current command transaction.

The *Timing* panel provides an overview of cumulative command timings for each document node. Commands with elapsed time below the given threshold are ignored in the grand total. Nodes are sorted according to their overall timing. For the document node that corresponds to the current buffer, individual command timings are shown as well. A double-click on a theory node or command moves the editor focus to that particular source position.

It is also possible to reveal individual timing information via some tooltip for the corresponding command keyword, using the technique of mouse hovering with `CONTROL`/`COMMAND` modifier key as explained in §2.4. Actual display of

timing depends on the global option `jedit_timing_threshold`, which can be configured in "Plugin Options / Isabelle / General".

The *Monitor* panel provides a general impression of recent activity of the farm of worker threads in Isabelle/ML. Its display is continuously updated according to `editor_chart_delay`. Note that the painting of the chart takes considerable runtime itself — on the Java Virtual Machine that runs Isabelle/Scala, not Isabelle/ML. Internally, the Isabelle/Scala module `isabelle.ML_Statistics` provides further access to statistics of Isabelle/ML.

3.3 Isabelle/Scala console

The *Console* plugin of jEdit manages various shells (command interpreters), e.g. *BeanShell*, which is the official jEdit scripting language, and the cross-platform *System* shell. Thus the console provides similar functionality than the special buffers `*scratch*` and `*shell*` in Emacs.

Isabelle/jEdit extends the repertoire of the console by *Scala*, which is the regular Scala toplevel loop running inside the *same* JVM process as Isabelle/jEdit itself. This means the Scala command interpreter has access to the JVM name space and state of the running Prover IDE application: the main entry points are `view` (the current editor view of jEdit) and `PIDE` (the Isabelle/jEdit plugin object).

For example, the subsequent Scala snippet gets the PIDE document model of the current buffer within the current editor view:

```
PIDE.document_model(view.getBuffer).get
```

This helps to explore Isabelle/Scala functionality interactively. Some care is required to avoid interference with the internals of the running application, especially in production use.

3.4 Low-level output

Prover output is normally shown directly in the main text area or secondary *Output* panels, as explained in §2.3.

Beyond this, it is occasionally useful to inspect low-level output channels via some of the following additional panels:

- *Protocol* shows internal messages between the Isabelle/Scala and Isabelle/ML side of the PIDE editing protocol. Recording of messages starts with the first activation of the corresponding dockable window; earlier messages are lost.

  Actual display of protocol messages causes considerable slowdown, so it is important to undock all *Protocol* panels for production work.

- *Raw Output* shows chunks of text from the `stdout` and `stderr` channels of the prover process. Recording of output starts with the first activation of the corresponding dockable window; earlier output is lost.

  The implicit stateful nature of physical I/O channels makes it difficult to relate raw output to the actual command from where it was originating. Parallel execution may add to the confusion. Peeking at physical process I/O is only the last resort to diagnose problems with tools that are not fully PIDE compliant.

  Under normal circumstances, prover output always works via managed message channels (corresponding to `writeln`, `warning`, `error` etc. in Isabelle/ML), which are displayed by regular means within the document model (§2.3).

- *Syslog* shows system messages that might be relevant to diagnose problems with the startup or shutdown phase of the prover process; this also includes raw output on `stderr`.

  A limited amount of syslog messages are buffered, independently of the docking state of the *Syslog* panel. This allows to diagnose serious problems with Isabelle/PIDE process management, outside of the actual protocol layer.

  Under normal situations, such low-level system output can be ignored.

Chapter 4

Known problems and workarounds

- **Problem:** Lack of dependency management for auxiliary files that contribute to a theory (e.g. **ML_file**).

  Workaround: Re-load files manually within the prover, by editing corresponding command in the text.

- **Problem:** Odd behavior of some diagnostic commands with global side-effects, like writing a physical file.

  Workaround: Copy / paste complete command text from elsewhere, or discontinue continuous checking temporarily.

- **Problem:** No way to delete document nodes from the overall collection of theories.

  Workaround: Ignore unused files. Restart whole Isabelle/jEdit session in worst-case situation.

- **Problem:** Keyboard shortcuts `C+PLUS` and `C+MINUS` for adjusting the editor font size depend on platform details and national keyboards.

  Workaround: Rebind keys via *Global Options / Shortcuts*.

- **Problem:** The Mac OS X keyboard shortcut `COMMAND+COMMA` for application *Preferences* is in conflict with the jEdit default shortcut for *Incremental Search Bar* (action `quick-search`).

  Workaround: Rebind key via *Global Options / Shortcuts* according to national keyboard, e.g. `COMMAND+SLASH` on English ones.

- **Problem:** Mac OS X system fonts sometimes lead to character drop-outs in the main text area.

  Workaround: Use the default `IsabelleText` font. (Do not install that font on the system.)

- **Problem:** Some Linux / X11 input methods such as IBus tend to disrupt key event handling of Java/AWT/Swing.

  Workaround: Do not use input methods, reset the environment variable `XMODIFIERS` within Isabelle settings (default in Isabelle2013-2).

- **Problem:** Some Linux / X11 window managers that are not "re-parenting" cause problems with additional windows opened by Java. This affects either historic or neo-minimalistic window managers like `awesome` or `xmonad`.

  Workaround: Use regular re-parenting window manager.

- **Problem:** Recent forks of Linux / X11 window managers and desktop environments (variants of Gnome) disrupt the handling of menu popups and mouse positions of Java/AWT/Swing.

  Workaround: Use mainstream versions of Linux desktops.

- **Problem:** Full-screen mode via jEdit action `toggle-full-screen` (default shortcut `F11`) works on Windows, but not on Mac OS X or various Linux / X11 window managers.

  Workaround: Use native full-screen control of the window manager (notably on Mac OS X).

- **Problem:** Full-screen mode and dockable windows in *floating* state may lead to confusion about window placement (depending on platform characteristics).

  Workaround: Avoid this combination.

Bibliography

[1] J. C. Blanchette. *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL.* http://isabelle.in.tum.de/doc/sledgehammer.pdf.

[2] J. C. Blanchette. *Picking Nits: A User's Guide to Nitpick for Isabelle/HOL.* http://isabelle.in.tum.de/doc/nitpick.pdf.

[3] M. Wenzel. *The Isabelle/Isar Implementation.* http://isabelle.in.tum.de/doc/implementation.pdf.

[4] M. Wenzel. *The Isabelle/Isar Reference Manual.* http://isabelle.in.tum.de/doc/isar-ref.pdf.

[5] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009).* ACM Digital Library, 2009.

[6] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. S. Coen and D. Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010), FLOC 2010 Satellite Workshop*, ENTCS. Elsevier, July 2010.

[7] M. Wenzel. Isabelle as document-oriented proof assistant. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Conference on Intelligent Computer Mathematics / Mathematical Knowledge Management (CICM/MKM 2011)*, volume 6824 of *LNAI*. Springer, 2011.

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

[9] M. Wenzel. READ-EVAL-PRINT in parallel and asynchronous proof-checking. In *User Interfaces for Theorem Provers (UITP 2012)*, EPTCS, 2013.

[10] M. Wenzel. Shared-memory multiprocessing for interactive theorem proving. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*. Springer, 2013.

[11] M. Wenzel and S. Berghofer. *The Isabelle System Manual*. http://isabelle.in.tum.de/doc/system.pdf.