An Architecture for Extensible Click'n Prove Interfaces

Holger Gast

Wilhelm-Schickard-Institut für Informatik
Sand 13, D-72076 Tübingen
University of Tübingen

Abstract. We present a novel software architecture for graphical interfaces to interactive theorem provers. It provides click'n prove functionality at the interface level without requiring support from the underlying theorem prover and enables users to extend that functionality through light-weight plugins. Building on established architectural and design patterns for interactive and extensible systems, the architecture also clarifies the relationship between the special application of theorem proving and conventional designs.

1 Introduction

Click'n prove [1], or prove-by-pointing [7], user interfaces for interactive theorem provers enable experts to edit proof scripts more efficiently and make the theorem prover more accessible to beginners. They visualize the proof state [28, 17] and interpret mouse gestures as commands that exhibit specific subterms of a goal [7, 9], perform context-sensitive rule applications [1, 6], or manipulate the proof script in a structure-oriented manner [9, 6]. Similar techniques have proven successful in software verification systems [17, 1, 22].

In previous designs, the click'n prove interface requires substantial support from the theorem prover: to interpret a mouse click as term selection, they assume that the prover augments its output with markups of the term structure [30, 8, 9, 21, 28, 3, 5]. Alternatively, the responsibility for pretty-printing terms is transferred to the interface entirely, and the prover delivers an encoding of its internal data structures [28, 30, 9, 6]. Some proposals [21, 3] extend the prover itself to handle click'n prove actions. While these approaches thus reuse the functionality available in the prover, they also require the prover to be modified for each specific interface and the invested effort cannot be transferred easily to other interfaces or provers.

In this paper, we propose an architecture for click'n prove interfaces in which the prover does not have to be aware of the interface and in particular does not have to be modified. The approach thus rests on a fundamental design rule for interactive applications [11, §2.4]: business (or application) logic and presentation logic should be strictly separated. While the business logic, i.e. the prover, remains stable over many releases, the user interface is subject to frequent changes that accommodate the needs of different user groups or allow the

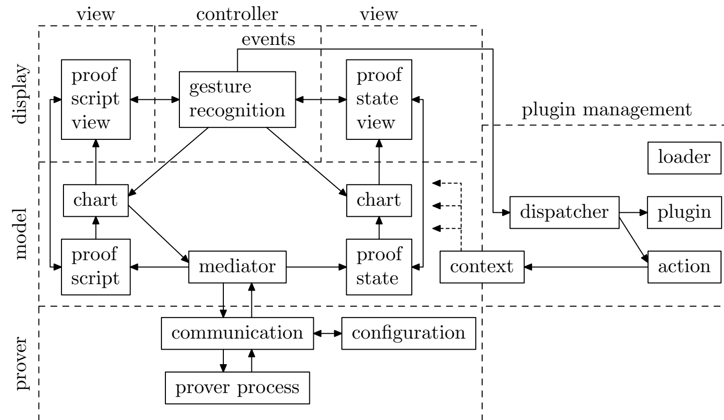

Fig. 1. System Architecture

system to run on different platforms. We use a general context-free, incremental parsing algorithm at the interface level to expose the subterm structure without breaking this design rule.

Our main objective is extensibility: previous designs required programming at the prover level to accommodate new click'n prove actions [8, 21] or were restricted to configuring a generic mechanism [29, 9, 3]. Instead, we propose that users should provide the functionality necessary for their daily work through light-weight plugins. The construction of these plugins must not require any knowledge about the theorem prover's implementation, and only modest insights into the architecture of the graphical interface. The general approach imitates the Eclipse plugin model [15], but reduces its complexity by retaining only those aspects that are immediately necessary.

The current implementation of the architecture realizes the given objectives for the Isabelle [25] theorem prover. The example plugins presented in Section 4 show how similar functionality can be provided for different provers within the established framework.

Structure of the Paper Section 2 motivates and describes the proposed architecture. Section 3 treats our parsing algorithm in more detail. Section 4 gives example plugins that provide click'n prove support for Isabelle/HOL [24]. Section 5 compares the proposal with related work. Section 6 outlines future directions and discusses design decisions. Section 7 concludes.

2 Architecture

To define our architecture, we first enumerate the essential design forces. We proceed with the division of responsibilities between the interface and the prover, and describe the extensibility mechanism. Finally, we discuss the steps necessary

to apply the architecture to specific theorem provers. The presentation stresses the standard architectural and design patterns for graphical user interfaces and extensible systems (e.g. [11, 26]) underlying our solution. In this way we hope to offer a new perspective on the commonalities and differences in software design between graphical interfaces for theorem provers and more conventional applications.

2.1 Design forces

The following considerations govern the decisions taken in the architecture.

Changing requirements User interfaces must often be changed to add new functionality for different user groups. In particular, users of theorem provers profit most from click'n prove support that addresses the situations they encounter most frequently. We thus envision that theories and corresponding support will be developed in parallel [29, 1].

User-level extension We expect that users will implement the support they require directly, rather than wait for the system developers to provide it.

Disjoint developer groups The theorem proving system consists of three, largely independent parts: the theorem prover, a core framework for the graphical user interface, and theory-specific click'n prove actions. We assume that these parts will be developed and maintained by three, largely disjoint groups of programmers who are familiar with their own code only.

Independent extension We expect that contributions to click'n prove functionality are most useful if they can be combined into a consistent environment in a flexible manner.

Complementary requirements The data structures of provers are optimized for the operations occurring in proof search and proof checking, but do not necessarily offer the operations required by interactive interfaces. Two examples for this are the navigation through a tree structure from nodes to their parents, and error recovery in parsing.

2.2 Separation of Application Logic and Presentation

Figure 1 exhibits the main components of our architecture and the connections between them. We now describe the left part of the figure, leaving the plugin management to Section 2.3. The system is divided into three layers: the prover layer, the model layer and the display layer.

The prover layer encapsulates the basic communication with the theorem prover. We assume that the prover provides a read-eval-print loop, which reads one textual command at a time from standard input, executes it and sends some textual result to the standard output. To accommodate differences in protocol, such as the prompt of the loop or the terminator for commands, a prover-specific *configuration* component is invoked in every communication. The prover layer provides a service to execute a single command and notifies the higher levels by callbacks when the answer has been received.

The model layer manages the data structures that are eventually displayed on the screen as dictated by the MODEL-VIEW-CONTROLLER (MVC) pattern [11, §2.4]. The components *proof script* and *proof state* store the textual representation of the current script and state as sent to and received from the prover. The management of the proof script is standard [9, 5, 2]. Incremental *chart parsers* (see Section 3) are registered as OBSERVERS [16] with both the script and the state. They maintain parse trees (or parse DAGs, in case of ambiguities) of the content. The interactions between the proof script, the prover component, and the user interface are complex. For example, when a command is sent, it must be contributed to the locked region of the script; if it fails or is undone later, the lock must be removed. This situation suggests a MEDIATOR [16] (see also [4]), which is responsible for managing the collaboration between the connected components.

The display layer is divided broadly into view and controller components, as suggested by the MVC pattern. The view components may, however, incorporate some controller functionality for entering the proof script by keyboard, and therefore implement the DOCUMENT-VIEW variant [11]. Besides the textual representation of their model data, they also access the chart for syntax highlighting, for instance to distinguish free and bound identifiers.

Gesture recognition is the central component for click'n prove functionality: it registers for mouse events, including drag&drop events, on all views and transforms them into click-and-prove events that it forwards, through the dispatcher component, to registered click'n prove actions. With these responsibilities, gesture recognition is a typical controller [11, §2.4]: It receives raw, system-dependent events and interprets them as high-level, application-specific events. Gesture recognition also implements term selection: when the user clicks to one of the views, it uses the selected character to ask the chart for the *selected token*. To accommodate ambiguous grammars, it then identifies a *selected path* as the longest path through the parse tree(s) upwards from the selected token. Finally, the *selected tree* is the smallest, i.e. the lowest, tree on the selected path. This protocol allows the user to select a specific tree by pointing to a token that is contained in none of its subtrees, for instance by pointing to the operator symbol of an expression.

2.3 Extensibility

We use the INTERCEPTOR pattern [26, §2] to achieve extensibility. In that pattern, a *framework* defines *interception points* for which extensions, or *interceptors*, can register to provide new services. The framework's behavior is specified by a finite automaton. Each of its transitions as an associated interception point, and when the automaton takes a particular transition, a descriptive *event* is sent to the registered interceptors via a *dispatcher* component. The interceptors are then given the opportunity to query the framework's state and modify its future behavior through a *context* object. The context object, as a FACADE object [16], accesses all of the components in the model layer, which is indicated by dashed

arrows in Figure 1. The concrete events of our architecture will be defined in Section 2.4.

The main benefit of the INTERCEPTOR approach to extensibility is that extensions are light-weight objects, which have to implement only a restricted interface to receive events from the framework. Their developers need to be familiar only with the framework's specification as a finite automaton and with the context object; the internal realization remains hidden. The experience with the example plugins (Section 4) suggests that users who are moderately familiar with the Java programming language will be able to contribute extensions to suit their particular needs.

Our architecture complements this setup with a *loader* component, which scans a `plugins` directory on startup and registers all found extensions (see [13, 15]). We implement *lazy evaluation* [15]: the plugin declares interceptors in an XML document `plugin.xml`, which contains sufficient information to provide the user interface representation. The implementing classes are loaded only when the user triggers an action.

2.4 Events

Currently, the framework defines two events: the TREESELECTEVENT occurs when the user selects a subterm with the mouse and requests a menu with the applicable actions. The TREEDEFAULTSELECTEVENT occurs when the user double-clicks on a subterm. Both events are characterized by a *location*, and the selected *token*, *tree*, and *path* as defined in Section 2.2. The location designates the view in which the event occurred, and is currently either `script` or `state`.

Dispatching of tree events proceeds in two steps: first, all actions registered for the event are queried whether they are *enabled* for the particular event. That condition is given by a boolean predicate on the event's parameters (see Section 4). Second, one action to be invoked is selected. In the case of a TREESELECTEVENT event, the user chooses from a popup menu. For the TREEDEFAULTSELECTEVENT event, the dispatcher checks whether there is exactly one enabled action and if so, it invokes that action.

2.5 Specialization for a Theorem Prover

The framework can be applied to a given theorem prover by providing the configuration component for the communication protocol, grammars for the prover's input language and the proof state display, and plugins that react to gesture events. Since the plugins will necessarily refer to specific non-terminals and node labels in the syntax trees and will furthermore generate prover-specific commands, they cannot be reused for different provers.

In the case of Isabelle, the required grammars can be obtained in a straightforward fashion: Isabelle supports several logics, which build, however, on a meta-logic `Pure`, whose syntax for types, terms, and propositions is given on a single page in [25]. The remaining productions are declared explicitly in the theories defining the various logics, and can be extract using only the `Pure`

grammar. The syntax for commands, however, is given by (backtracking) parsing functions, such that its grammar cannot be extracted from theories but must be provided by hand. Since full context-free parsing is available, the grammars can be extracted directly from the reference manuals. For the proof state display, the output functions have been inspected.

3 Parsing the Proof Script and State

The interpretation of mouse gestures requires a suitable internal representation of the material displayed on the screen. Since the proof state and the proof script are given as text documents, a straightforward solution is to construct a parse tree (or a parse DAG, in case of ambiguities). To support the Isar notion of proof scripts as self-contained documents [23], click'n prove actions must be applicable to the script as well as the proof state. Parsing at the interface level is a major requirement of the proposed architecture, and we describe our algorithm in some detail to demonstrate that the approach is viable. The Isar input language and the proof state display of Isabelle pose three major challenges:

1. They contain nested languages, with an *outer syntax* that describes the overall structure and an *inner syntax* for terms and propositions.
2. Both languages are extensible by declarations in theories and require full context-free grammars.
3. The proper handling of the proof script requires incremental parsing.

Before giving our solution, we review briefly why existing, mainstream approaches to parsing fail to meet these challenges.

3.1 Existing Approaches

Parsing with nested languages has been studied extensively. The accepted solution is to incorporate the lexical analysis into the context-free grammar. The ASF+SDF tool [32] handles full context-free grammars using GLR parsing [31], but adding the lexical analysis results in extremely large LR-automata that preclude on-the-fly generation for extensible grammars. The Harmonia framework [18] also uses GLR parsing. Its proposed solution to nested languages is to associate scanners with non-terminals, but this extension has not been implemented so far. It is also unclear how extensible grammars could be supported. PackRat (or PEG) parsing [14] uses a backtracking recursive descent parser with memoization and integrates lexical and syntactical analysis. Since parser generation is a simple process, extensible grammars could be implemented. However, PackRat parsing only works for deterministic grammars without left-recursion.

Approaches based on structured editing (e.g. [10]), which have been used for theorem provers before [30, 9], seem to deviate too much from the usage of modern IDEs: users expect parsing to proceed in the background without restricting the possible edits, and to finish shortly after they have produced syntactically correct input.

The ProofGeneral project [5] and CtCoq [6] assign the responsibility for parsing commands to the theorem prover. Unfortunately, Isabelle currently does not produce parse trees for commands, but represents commands as `transitions`, which are (ML) functions from states to states. It thus appears that to obtain more detailed information, the parser for the Isar outer syntax would have to be rewritten almost entirely.

3.2 Incremental Chart Parsing

Chart parsing [20] works with general context-free grammars and can handle incremental parsing [33, 27]. Since it does not require a generation phase, it is a light-weight solution to extensible languages. Perhaps the best-known chart parser is Earley's algorithm [12], which Isabelle uses for its inner syntax.

To define chart parsing, let $G = (N, T, P, S)$ be a grammar with non-terminals N, terminals (or tokens) T, productions P, and a start symbol S. We assume that the right-hand side of a production either contains only non-terminals or consists of a single terminal symbol, in which case it is called a *pre-terminal* production. Let s be an input string that lexical analysis has split into tokens $t_1 \ldots t_n$. The *chart* is a directed graph (V, E) where the vertices $V = \{0, \ldots n\}$ mark the positions between the tokens, before the first, and after the last token. The edges E are triples $(v, v', A \to \alpha \cdot \alpha')$ of two vertices v, v' and an *item*, such that $A \to \alpha\alpha'$ is a production in P. The dot indicates the current position in the parsing process. An edge is *active* if $\alpha' \neq \varepsilon$, and *inactive* otherwise.

The chart is initialized by adding a pre-terminal edge $e_i = (i-1, i, T_i \to t_i)$ for each token t_i and an edge $(0, 0, S \to \cdot\alpha)$ for each production $S \to \alpha \in P$. Then, the following steps are applied until no new edges are produced:

predict For each edge $(v, v', A \to \alpha \cdot B\alpha')$ in the chart and each production $B \to \beta \in P$, add an edge $(v', v', B \to \cdot\beta)$.

combine For each active edge $(v, v', A \to \alpha \cdot B\alpha')$ and inactive edge $(v', v'', B \to \beta\cdot)$ in the chart, add an edge $(v, v'', A \to \alpha B \cdot \alpha')$

There are only finitely many possible edges and the process terminates with a chart that contains an edge $(v, v', A \to \alpha\cdot)$ iff there is a derivation $A \Rightarrow \alpha \Rightarrow^* t_{v+1} \ldots t_{v'}$. To enumerate all parse trees, it is sufficient to mark edges with a unique identifiers, and to modify the *combine* step to record in the result the identifier of the referenced, inactive edge. An edge e *depends on* an edge e' [33] iff e references e', or e depends on some edge e'' that depends on e'. It is straightforward to extend the framework to priority grammars [25, §7.1], such that associativity and precedence of operators can be encoded.

Wirén's algorithm for incremental parsing [33] takes a chart and a single modification that consists in the addition, deletion, or replacement of some token t. Multiple changes are processed in order. The algorithm splits the chart at the point of modification by removing all edges spanning the modification and renumbering the vertices to accommodate the insertion or deletion of t. After adding the pre-terminal edge for a newly inserted token, the normal parsing algorithm is run to complete the chart for the modified input.

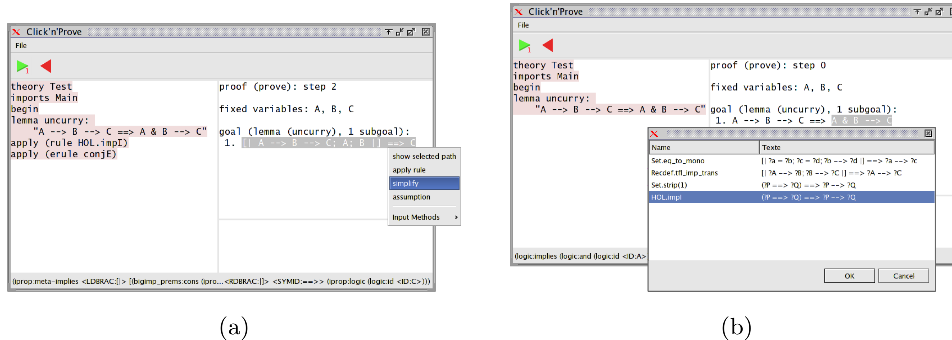

Fig. 2. Invoking Actions by Mouse Gestures

Our algorithm improves on Wirén's in that edges reference equivalence classes of edges, where two edges are equivalent if they have the same start and end vertices and their items have the same left-hand side. With this definition, the algorithm generates a packed representation of the parse forest that results from ambiguous grammars. This packed representation also allows for more efficient incremental parsing by lazy edge removal: when some edge e becomes invalid due to a modification, we run the chart parser locally to produce a new edge e' that is equivalent to e. If such an edge is found, the dependent edges of e do not have to be processed at all, and parsing finishes.

To accommodate nested languages, we introduce *lexer switching*. A switch to lexer l is marked by $\uparrow^l$ in the right-hand side of productions. The chart structure is generalized such that a token t can be surrounded be arbitrary vertices v, v', which are not necessarily numbered consecutively. When the chart parser encounters an edge $(v, v', A \to \alpha \cdot \uparrow^l \alpha')$, it searches for a token t that is located in the input string at the end of token v' and has been produced by lexer l. If v'' is the vertex immediately before t, the algorithm creates an edge $(v, v'', A \to \alpha \uparrow^l \cdot \alpha')$, such that parsing proceeds with the pre-terminal edge of t. For efficiency, tokens are generated lazily when some production requests them.

4 Plugins

This section demonstrates how plugins can implement click'n prove functionality based on the framework proposed in Sections 2 and 3. The framework itself is developed in Java using the SWT and JFace libraries [13] for the user interface. The examples are chosen to illustrate the services available from the framework, but do not provide a comprehensive click'n prove interface for Isabelle. Since the framework API is still under development, we cannot yet present a complete specification. We include the actual code of the plugins to substantiate the claim that only modest experience is necessary to produce them.

4.1 Simplification

Isabelle's simplifier is used in tactic-style proofs to rewrite the first goal of the current state with a set of equality theorems declared as simplifier rules. In the click'n prove interface it should therefore be available in the context menu when the user points to the first goal of the proof state.

Figure 2(a) shows a screenshot: the left-hand side panel contains the proof script in which the locked region is marked by a red background. The right-hand side is split between the proof state on top and a panel for error messages below. By pointing the mouse to the `-->` operator, the user has selected the goal term as indicated by the highlight. The term is also shown in prefix notation in the status line. A click on the right mouse button has brought up the action menu, where the simplifier action is available.

When the user selects the menu entry, the dispatcher component invokes the `run()` method of the following class, which implements the interface associated with the TreeSelectEvent (Section 2.4). The `run()` method uses the context object to insert a new command into the proof script.

```
public class Simplify implements TreeSelectActionDelegate {
  public void run(ClickProveContext ctx, TreeSelectEvent ev) {
    ctx.insertAndSubmitGeneratedCommand("apply (simp)");
  }
}
```

To make the simplifier action appear in the context menu, the following XML element is included in the plugin descriptor (Section 2.3). It specifies the internal identifier of the action, the label for the menu entry and the class containing the implementation. Furthermore, the element declares the *enable* condition for the action: the location of the tree select event must be the proof state and the selected path must include a subgoal node, which is directly below a `subgoals` node representing the list of all subgoals, which again is directly below the `ps_goals` node, which captures the lower part of Isabelle's proof state. Paths are read from the bottom to the top, paralleling the nesting of parse trees. The XML fragment thus expresses the condition that the user has selected some term within in the first goal of the proof state.

```
<tree-event
  id="tactic.simplify"
  label="simplify"
  class="tactic.Simplify">
  <enable>
    <and>
      <location id="state"/>
      <path><node nt="ps_goals"/>
            <node nt="subgoals" label="cons"/>
            <node nt="subgoal"/>
            <any/>
      </path>
    </and>
  </enable>
</tree-event>
```

The conditions on the selected path expressible in XML are modeled after regular expressions: `node` requires a node with a specific label and/or a non-terminal; the attribute `pos="`*i*`"` indicates that the node must be the *i* th child of the next node on the selected path. The condition `any` matches an arbitrary sequence of nodes; `alt`,`maybe`,`repeat`, and `seq` represent the regular operators for alternative, option, Kleene star, and sequence.

4.2 Rule application

Our second example is an action that searches for the rules applicable to the conclusion of the first goal. It demonstrates that with more collaboration from the prover, more sophisticated click'n prove actions become possible. Isabelle provides a command `find_theorems`, which searches the current proof context for theorems fulfilling a given condition. The condition `intro` finds rules that are applicable to the first goal. Figure 2(b) shows the resulting screenshot: the user has pointed to the first goal and selected from the context menu the action `apply rule`, which has queried the theorem database and opened the dialog.

The *enable* condition for the new action is similar to that for simplification. The following code sequence is taken from the implementation of rule application. It demonstrates how commands that do not belong to the proof script can be sent to the prover and how the resulting output can be retrieved by a callback method.

```
TheoremSelectDialog sel
        = new TheoremSelectDialog(ctx.getShell());
Command c = new TextCommand("find_theorems intro");
ActiveCommand cmd = ctx.submitSilentCommand(c);
cmd.addListener(new ReportFound(ctx, sel));
if (sel.open() == Window.OK) {
   String thm = sel.getSelectedTheorem();
   if (thm != null) {
    ctx.insertAndSubmitGeneratedCommand("apply (rule "+thm+")");
  }
}
```

The call to `submitSilentCommand` sends the given `find_theorems` command to the prover for execution, and returns immediately with an `ActiveCommand` object that represents the command to be executed.[1] The `ActiveCommand` object notifies registered Observers [16] when the prover completes processing. In the example, a `ReportFound` object scans the output and inserts the found theorems into the open `TheoremSelectDialog`. When the user selects one of those theorems, it is applied to the current state.

[1] This behavior is essential because the `run()` method is invoked within the GUI event-thread and no new events are accepted before it returns – the user interface is "frozen". The call to `sel.open()` does not return immediately either, because the dialog is modal, but this situation is handled internally by the SWT library.

4.3 Quick Introduction and Elimination

The search for applicable rules using the action from Section 4.2 may take a few seconds, which is unacceptable when the desired rule is "obviously" the standard introduction or elimination rule. The `QuickIntro` and `QuickElim` classes therefore hold an extensible mapping from non-terminal/label pairs to the standard rules. For the Isabelle/HOL logic, for example, `QuickElim` maps (`logic`, `exists`) to `exE` and (`logic`, `and`) to `conjE`. The following code fragment is registered for the TreeDefaultSelectEvent, i.e. a double-click on one of the premises of the first goal.

```
Object nt = ev.getTree().getNonTerminalID();
String l = ev.getTree().getLabel();
String rl = rules.get(new NTLabelPair(nt, l));
if (rl == null) {
  ctx.showErrorDialog("No quick elim rule for "+nt+":"+l);
} else {
  ctx.insertAndSubmitGeneratedCommand("apply (erule+" "+rl+")");
}
```

Since the `QuickIntro` is defined analogously, we must give a precise *enable* condition for `QuickElim` to ensure mutual exclusion. `QuickElim` is applicable only under the first subgoal's top-level meta-implication, and here to any one of the list of premises or the single premise, if there is just one.

```
<path>
  <node nt="subgoal"/>
  <maybe><node nt="logic" label="meta-all"/></maybe>
  <node label="meta-implies"/>
  <alt>
    <seq><node pos="0"/>
         <repeat><node nt="bigimp_prems"/></repeat>
         <node/>
    </seq>
    <node pos="0"/>
  </alt>
</path>
```

4.4 Picking a fact in Isar

The following action demonstrates the use of *tree navigators* to access the parse DAG, starting from the selected tree. Bertot [6] shows that a similar access to the syntax tree is sufficient to implement proof-by-pointing [7].

The grammar for Isar proof scripts [23] includes named assumptions and intermediate facts of the form *id*`:` `"`*proposition*`"`. To use a fact in a proof, it must be "picked" with the command `from`. The following simple action, registered for TreeDefaultSelectEvent, enables the user to generate the command by double-clicking anywhere within the named fact.

```
sel = ev.getTree().navigate()
  .up("props",null)
  .child(0).check("thmdecl_opt", "some").child(0).single();
Token t = sel.getToken(0);
ctx.insertGeneratedCommand("from "+t.getText());
```

A `TreeNavigator` object represents a set of positions reached in the current navigation. The `navigate()` method of a tree node returns the tree node itself. The navigator's `up()` method searches the ancestor nodes for a non-terminal/label combination and returns a navigator for these. The next steps in the example navigate down to the first child, check that a name is present (it is optional in the grammar), and finally access the name. Since grammars can be ambiguous, tree navigation in general yields several results and the `TreeNavigator` class encapsulates the backtracking search. The user can retrieve the results by the statement `for (TreeNode p:` *navigator*`)`. Alternatively, the method `single()` returns a single result if it exists or throws an exception.

The proof script is parsed incrementally, such that more sophisticated versions of the `Pick` action could examine – through the context object – the preceding commands to determine the exact command to be generated: after a `from` command, the prover is in *chain* mode and does not accept a second `from`. Instead, the first `from` must be augmented with a new `and` clause. After a goal statement, i.e. in *show* mode, a `using` command should be issued.

5 Related Work

User interfaces for theorem provers have attracted the attention of numerous researchers. In the following, we therefore focus on the main points of comparison: the collaboration with the prover that enables click'n prove functionality, and the mechanisms for extensibility. The existing solutions for parsing the proof script have been discussed in Section 3.1.

Bertot, Théry, and Kahn [30, 7, 9, 6] have introduced and developed the proof-by-pointing paradigm. They expect the prover to send output as trees, which they render on the screen using the PPML formalism of CENTAUR [10]. The translation of mouse gestures to proof commands takes place at the interface level [6]; the editor for the proof script is structure-oriented, requiring the user to manipulate abstract syntax trees, but also providing extensive editing support.

A similar implementation strategy is used in [8, 21, 3], where the prover marks each subterm in the output with its path p in the overall syntax tree. Bertot et al. [8] interpret a mouse click by sending a command `pbp` p (for *proof-by-pointing*) to the prover, which generates a command, executes it, and sends it back to the interface for inclusion in the proof script. Aspinall and Lüth [3] generate commands from templates at the interface level.

The Jape editor for proof documents [29] includes a theorem prover that appears to share its data structures with the interface, such that mouse gestures can be readily interpreted. In the KIV [17] and Jive [22] verification systems, the graphical front-end is likewise coupled tightly to the built-in prover. The interface

for the distributed system Ωmega [28] expects the prover to transmit its internal data structures, which enables elaborate and fine-grained visualization.

Extensibility has been achieved by two means: generic algorithms that can be configured for specific application domains, and broker architectures that provide a common infrastructure to which provers and front-ends can attach to exchange messages.

Bertot and Théry [9] generalize the proof-by-pointing algorithm to accommodate new connectives. The implementation in [8] likewise uses an extensible repository of functions that handle specific connectives, but they must be programmed at the prover level.

The Jape [29] editor can be configured by the inference rules of concrete logics. The declaration of a rule also includes the gesture that is to invoke the rule. The set of available gestures is large, but fixed, such that, for example, specific dialogs cannot be programmed.

Lüth and Wolff [21] introduce the *notepad* metaphor. The user manipulates *objects* displayed on the notepad by drag&drop gestures, which are interpreted uniformly as function application. The system can thus be used for any *application* that provides concrete representation types for objects and applicable operations. Aspinall and Lüth [3] implement a generic interface that can be configured with the command templates to be filled out for drag&drop gestures.

The $\mathcal{LOUI}$ interface [28] for Ωmega represents rules and tactics introduced by theories by new menu entries. If a tactic requires parameters such as the instantiations for a variable, a generic dialog is opened for the user to enter the missing data.

Broker [11] architectures allow loosely coupled, possibly distributed components to interact by message passing. The Ωmega prover [28] and current versions of the ProofGeneral [5, 4] provide an infrastructure for prover and display components to communicate. The XML-based PGIP protocol [5] abstracts over specific provers and defines a communication standard. The message passing interface in broker architectures requires the individual components to maintain internal state, such that programming extensions, especially with click'n prove functionality, involves a substantial effort.

6 Future Work and Discussion

We are currently developing the implementation of the architecture and a click'n prove interface for Isabelle. The example plugins in Section 4 show the current state and exhibit further immediate requirements: proof script management for multiple documents [9, 5], a View Handler [11] for switching between visualizations, and context-sensitive syntax highlighting. An open strategic question is whether our architecture should build on the Eclipse platform [13, 15] (following [4]) to import the available view management and plugin mechanisms. There are mainly three considerations: a user contributing click'n prove actions should not have to know the Eclipse plugin model in detail, the graphical front-end is

to remain light-weight, and special plugins for single theories may need to be loaded after startup and from locations outside of the installation directory.

A major direction is the integration of new views. Apart from visualizing the goal structure (see for example [19]), we intend to provide the notepad metaphor [21] for offline-calculations, which would be performed by silent prover commands (Section 4.2). This extension introduces a DragAndDropEvent with *selected source* and *selected destination* objects as parameters. We expect that the editing of Isar proof scripts [23] will be simplified if intermediate results in calculational reasoning and auxiliary facts can be generated by forward reasoning on the notepad.

The second major goal is the definition of a stable API for plugins and a more comprehensive set of events. Useful extensions can be found in the motivation of the Interceptor pattern [26]. For instance, mathematical notation can be encoded and decoded by a plugin if the framework provides interceptable events Load and Save for proof scripts, Send for commands, and Receive for answers.

A question that can be answered only with more experience is whether the access to syntax trees is sufficient to implement all desirable click'n prove actions. Bertot [6] provides a partial answer: once a grammar for the term structure is completed, proof-by-pointing can be implemented. However, the Isabelle parsing model [25, §8] defines four steps to obtain the internal tree representation: generation of parse trees, application of parse translations and macro expansion, and finally type inference. We contend that it will be sufficient that parse trees are available, because users necessarily manipulate this externally visible form, thus expecting support only at this level. The lack of type information implies, however, that no disambiguation of parse DAGs can take place and the available actions cannot depend on types.

7 Conclusion

We have presented an architecture for click'n prove interfaces to interactive theorem provers that differs from previous proposals in two aspects: it requires no support from the prover beyond a textual interface with a read-eval-print loop, and it is extensible by light-weight plugins that register to be notified for defined events. Our architecture is based entirely on established patterns for interactive and extensible systems, thus linking the special application of interfaces for theorem provers to more widespread software designs.

We use a parser accepting general context-free grammars to analyze both the proof state and the proof script. Since the parser works incrementally, users can select subterms from the script during editing, but they are not constrained to manipulating only well-formed syntax trees.

The experience with the current implementation suggests that users moderately familiar with the framework will be able to provide new click'n prove functionality with little effort. Actions may analyze the structure of the proof state and proof script in detail to decide which commands to generate, they may invoke the prover for more sophisticated queries, and may use arbitrary services of the GUI toolkit for specialized communication with the user.

References

1. Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.
2. David Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, number 1785 in Lecture Notes in Computer Science, 2000.
3. David Aspinall and Christoph Lüth. ProofGeneral meets IsaWin. In *User Interfaces for Theorem Provers (UITP '03)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2003. (to appear).
4. David Aspinall, Christoph Lüth, and Ahsan Fayyaz. Proof general in eclipse: System and architecture overview. In *Eclipse Technology Exchange Workshop at OOPSLA 2006*, 2006.
5. David Aspinall, Christoph Lüth, and Daniel Winterstein. Parsing, editing, proving: The pgip display protocol. In *International Workshop on User Interfaces for Theorem Provers 2005 (UITP'05)*, 2005.
6. Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11(33):225–243, 1999.
7. Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, number 789 in LNCS, pages 141–160. Springer-Verlag, 1994.
8. Yves Bertot, Thomas Kleymann-Schreiber, and Dilip Sequeira. Implementing proof by pointing without a structure editor. Technical Report ECS-LFCS-97-368, Department of Computer Science, Edinburgh University, October 1997.
9. Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *J. Symbolic Computation*, 25:161–194, 1998.
10. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGPLAN Notices*, 24(2):14–24, February 1989.
11. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns*, volume 1. Wiley & Sons, 1996.
12. Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–103, February 1970.
13. The Eclipse workbench. `http://www.eclipse.org`.
14. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM Press.
15. Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns and Plugin-ins*. The Eclipse series. Addison-Wesley, 2004.
16. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
17. Dominik Haneberg, Simon Bäumler, Michael Balser, Holger Grandy, Frank Ortmeier, Wolfgang Reif, Gerhard Schellhorn, Jonathan Schmitt, and Kurt Stenzel. The user interface of the KIV verification system - a system description. In *Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005)*, 2005.

18. The Harmonia research project. http://harmonia.cs.berkeley.edu/.
19. Martin Homik, Andreas Meier, and Christoph Benzmüller. Designing a proof GUI for non-experts: Evaluation of an experiment. Presentation at UITP '05, 2005.
20. Martin Kay. Algorithm schemata and data structures in syntactic processing. In *Readings in natural language processing*, pages 35–70. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1986.
21. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 19(2):167–189, 1999.
22. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS '00, Tools and Algorithms for the Construction and Analysis of Software*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.
23. Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 2003.
24. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2002.
25. Lawrence C. Paulson. *Isabelle – A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg, 1994.
26. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented Software Architecture: Patterns for concurrent and networked objcts*, volume 2. Wiley & Sons, 2000.
27. Rolf Schwitter. Incremental chart parsing with predictive hints. In *Proceedings of the Australasian Language Technology Workshop*, pages 1–8, University of Melbourne, Australia, 2003.
28. Jörg Siekmann and Stephan Hess. $\mathcal{L}\Omega\mathcal{UI}$: Lovely Ωmega user interface. *Formal Aspects of Computing*, 3:1–17, 1999.
29. Bernard Sufrin and Richard Bornat. User interfaces for generic proof assistants: Interpreting gestures. In *User Interfaces for Theorem Provers '96*, 1996.
30. Laurent Théry, Yves Bertot, and Gilles Kahn. Real theorem provers deserve real user-interfaces. In *Proceedings of the fifth ACM SIGSOFT symposium on software development environments*, pages 120–129, 1992.
31. M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.
32. Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2002.
33. Mats Wirén. Interactive incremental chart parsing. In *Proceedings of the fourth conference on European chapter of the Association for Computational Linguistics*, pages 241 – 248, Manchester, England, 1989.