The Design and Implementation of the GraPE Graphical Proof Editor

Max Schäfer

Abstract

We present GraPE, a graphical proof editor that strives to provide a unified point-and-click interface for a variety of existing theorem provers and supports advanced proof editing features. It strictly separates the graphical user interface responsible for displaying the derivation being worked on from the actual proof engine, yielding a flexible and powerful theorem proving environment. Unlike many other proof editors, GraPE not only supports inference rules for incrementally building a derivation but also more complex proof generation and manipulation operations, such as automatic proof search and elimination of admissible rules. GraPE provides particularly strong support for the calculus of structures and for integration with the Maude language. We show how this support can be leveraged to quickly and easily obtain a graphical theorem prover for system KSg for classical propositional logic.

1 Introduction

Graphical proof editors have long been the subject of research and implementation efforts [26, 24]. For our purposes, the existing systems can be classified into two categories:

1. graphical user interfaces designed to work with one specific theorem prover
2. independent graphical user interfaces that can be interfaced to different underlying theorem provers

Examples from the first group are abundant – suffice it to mention CtCoq [3] and Alfa [13]. Representatives of the second group are less frequent: Proof General [2] is a recent example. Other systems such as Jape [4] do maintain a separation between graphical frontend and prover, but the interface between the two is often not general enough or not very well documented, thus effectively making the two components hard to separate. Jape in particular is closely tied to the sequent calculus and adapting it to other systems such as the calculus of structures [11, 12] would likely require some fundamental architectural changes.

GraPE belongs to the second category: it consists of a general, inference system agnostic frontend responsible for displaying the current derivation and handling user interaction, which is interfaced to a theorem prover (the backend) doing the actual work of building derivations. In particular, GraPE improves on previous work in three aspects:

1. Frontend and backend are clearly separated and communicate using a simple protocol documented in this paper, which makes the implementation of new backends easy.

2. A very versatile backend based on the Maude language [7, 8], called `GraPE2Maude`, is provided, which gives access to a suite of already implemented theorem provers for systems in the calculus of structures [17].

3. Backends can not only provide simple inference rules but also more complicated proof search and manipulation procedures.

The first feature is by no means new, it has for example been discussed at length in [26]. However, none of the theorem provers we are familiar with actually provides a good implementation of this principle.

The second feature, which extends earlier work by Kahramanoğulları [16], is not part of GraPE *per se*, but is interesting in its own right. Maude provides a convenient, high-level way of implementing easily understandable yet quite fast theorem provers especially for systems in the calculus of structures. This backend makes it easy to use and modify already existing implementations, but also to experiment with new inference systems and explore them interactively.

The third point, finally, has not received the attention in the literature we think it deserves. To the best of our knowledge, existing proof editors rarely support any kind of proof manipulation beyond simple changes in graphical representation (for example reformatting a sequent calculus derivation into natural deduction style as supported by Jape). Mostly, they present a derivation as a list of goals (i.e., as yet unproved premises); the application of an inference rule results in adding zero or more new premises to this list, while discharging a previously open goal. No representation of the derivation as a whole is provided.

GraPE, on the other hand, gives the prover full access to the derivation under construction, and allows rules to change the derivation in an arbitrary way. This is the basis for implementing proof manipulation procedures to eliminate admissible rules or even full-blown cut elimination.

2 An Example: Propositional Logic with System KSg

As a concrete example of how to use GraPE we show a sample session of the program in which we utilize a Maude implementation of system KSg of the calculus of structures to prove two simple theorems of classical propositional logic.

To begin with, we give a short introduction to the calculus of structures and system KSg.

2.1 The Calculus of Structures

The calculus of structures [11, 12] is a generalization of the sequent calculus. Compared to the sequent calculus, it introduces a number of new concepts:

Structures The notions of formula and sequent are merged into the notion of *structure*, which is intermediate between a sequent and a formula.

Deep Inference In the calculus of structures, inference rules can be applied at any depth inside a structure, not only at the topmost connective as in the sequent calculus.

Proof Chains Each inference rule only has a single premise, thus there is no branching as in sequent calculus proofs, and indeed it is not needed: in the sequent calculus, the main purpose of branching rules is to bring deeper lying connectives to the top and subject them to inference rules; with deep inference, this can be accomplished more directly.

Symmetry As a consequence, inference rules and derivations are symmetric between premise and conclusion. This symmetry has many interesting proof-theoretical consequences; in particular, every inference rule can be dualized.

Equational Theory Structures are viewed modulo a congruence relation induced by an equational theory. Each inference system comes with its own equational theory, usually containing laws for, e.g., commutativity and associativity of logical operators. Two formulae which are equivalent under the equational theory can be seen as two representatives of the same structure.

Inference rules in the calculus of structures are given as schemata of the form

$$\rho \, \frac{S\{T\}}{S\{R\}}$$

where $S\{T\}$ is the rule's premise, ρ its name, and $S\{R\}$ the conclusion. Here, $S\{\ \}$ stands for a structure with a "hole", i.e. exactly one operand position inside the structure, which occurs in the scope of an even number of negations, contains the placeholder $\{\ \}$. The schema $S\{T\}$ is obtained by plugging structure schema T into that hole.

An instance of a rule schema is obtained by replacing the structure schemata T and R with actual structures respecting their schema. A chain of inference rule instances of the form

$$\begin{array}{rc} \rho_1 & \dfrac{R_0}{R_1} \\ & \overline{} \\ & \vdots \\ \rho_n & \overline{} \\ & R_n \end{array}$$

where $n \geq 0$, is called a *derivation*. In analogy to rule schemata, we also call R_0 the derivation's premise, and R_n its conclusion.

From a term rewriting point of view, the $S\{\ \}$ in the rule schema can be seen as expressing a position inside a structure. Reading the rule bottom up (the usual direction during proof search), we can then interpret it as saying that a substructure R at this position can be rewritten to a structure T. Indeed, this viewpoint can be exploited when implementing theorem provers for deep inference systems in a term rewriting language [15], as is done in our example. In keeping with term rewriting terminology, we will often refer to the structure substituted for R in a rule application as the *redex*, and to the one substituted for T as the *contractum*.

2.2 The Systems KSg and SKSg

The calculus of structures accommodates many different inference systems for many different logics (some references are given later). As examples of deep inference systems, we will take a look at system KSg and its extension SKSg, following [5] and [16].

System KSg is an inference system for classical propositional logic. Its language contains a countable set A of atoms denoted by lowercase letters. KSg structures are generated by the production

$$R ::= \mathsf{tt} \mid \mathsf{ff} \mid A \mid \overline{R} \mid [R, R] \mid (R, R)$$

Here, (R, R) stands for conjunction, $[R, R]$ for disjunction, and $\overline{R}$ for negation. The two logical constants *truth* and *falsity* are written tt and ff.

The equational theory underlying KSg is given in Figure 1. It states that conjunction and disjunction are associative and commutative with tt and ff as their respective units, and also contains the De Morgan rules and the involution law. Although conjunction and disjunction have been defined as binary operators, associativity allows us to denote them in n-ary form and save some brackets, writing (a, b, c) for $((a, b), c)$ and $(a, (b, c))$.

By application of these rules, we obtain, for example,

$$\overline{[a, \overline{b}, \mathsf{ff}]} \approx (\overline{a}, \overline{\overline{b}}, \overline{\mathsf{ff}}) \approx (\overline{a}, b, \mathsf{tt}) \approx (\overline{a}, b)$$

which means that the two formulae $\overline{[a, \overline{b}, \mathsf{ff}]}$ and $(\overline{a}, b)$ are representatives of the same structure.

In fact, the equational system allows us to always pick a representative in negation normal form (i.e., where negation is only applied to atoms) in every structure.

Associativity

$$[[R,T],U] \approx [R,[T,U]]$$
$$((R,T),U) \approx (R,(T,U))$$

Commutativity

$$[R,T] \approx [T,R]$$
$$(R,T) \approx (T,R)$$

Units

$$(\mathsf{ff},\mathsf{ff}) \approx \mathsf{ff}$$
$$[\mathsf{tt},\mathsf{tt}] \approx \mathsf{tt}$$
$$[\mathsf{ff},R] \approx R$$
$$(\mathsf{tt},R) \approx R$$

Negation

$$\overline{\mathsf{ff}} \approx \mathsf{tt}$$
$$\overline{\mathsf{tt}} \approx \mathsf{ff}$$
$$\overline{[R,T]} \approx (\overline{R},\overline{T})$$
$$\overline{(R,T)} \approx [\overline{R},\overline{T}]$$
$$\overline{\overline{R}} \approx R$$

Figure 1: The Equational Theory of System KSg

$$\boxed{\mathsf{i}{\downarrow}\,\frac{S\{\mathsf{tt}\}}{S[R,\overline{R}]} \qquad \mathsf{s}\,\frac{S([R,U],T)}{S[(R,T),U]} \qquad \mathsf{w}{\downarrow}\,\frac{S\{\mathsf{ff}\}}{S\{R\}} \qquad \mathsf{c}{\downarrow}\,\frac{S[R,R]}{S\{R\}}}$$

Figure 2: System KSg

The inference rules of system KSg are given in Figure 2; note that we sometimes omit the context brackets $\{\ \}$ if the structure schema inside the context has brackets of its own. For example, we write $S[R,\overline{R}]$ instead of $S\{[R,\overline{R}]\}$.

Three of the given rules, namely interaction ($\mathsf{i}{\downarrow}$), weakening ($\mathsf{w}{\downarrow}$), and contraction ($\mathsf{c}{\downarrow}$), also appear in most sequent calculus systems, whereas switch (s) is peculiar to the calculus of structures: it does not normally appear in the sequent calculus, but it is part of every system in the calculus of structures, for every logic (of course, the precise notions of conjunction and disjunction involved differ between systems).

The truth constant tt is the only axiom of KSg. Every derivation with tt as its premise is called a *proof.* For example, we can use the given rule schemata to build a proof of the structure $[a,\overline{a}]$, which corresponds to the classical tautology $a \vee \neg a$:

$$\mathsf{i}{\downarrow}\,\frac{\mathsf{tt}}{[a,\overline{a}]}$$

Here, the $\mathsf{i}{\downarrow}$ rule is applied in an empty context, just as it could be done in the sequent calculus. But that does not have to be case, as this example shows:

$$\mathsf{i}{\downarrow}\,\frac{[b,\overline{b}]}{([b,\overline{b}],[a,\overline{a}])}$$

$$\mathsf{i}{\downarrow}\,\frac{S\{\mathsf{tt}\}}{S[R,\overline{R}]} \qquad \mathsf{i}{\uparrow}\,\frac{S(R,\overline{R})}{S\{\mathsf{ff}\}}$$

$$\mathsf{s}\,\frac{S([R,U],T)}{S[(R,T),U]}$$

$$\mathsf{w}{\downarrow}\,\frac{S\{\mathsf{ff}\}}{S\{R\}} \qquad \mathsf{w}{\uparrow}\,\frac{S\{R\}}{S\{\mathsf{tt}\}}$$

$$\mathsf{c}{\downarrow}\,\frac{S[R,R]}{S\{R\}} \qquad \mathsf{c}{\uparrow}\,\frac{S\{R\}}{S(R,R)}$$

Figure 3: System SKSg

Here, the context is $([b,\bar{b}],\{\})$. The rule instance used is

$$\mathsf{i}{\downarrow}\,\frac{([b,\bar{b}],\mathsf{tt})}{([b,\bar{b}],[a,\overline{a}])}$$

This instance is in fact the same as the above derivation, since by the equational theory of KSg, we have $([b,\bar{b}],\mathit{tt}) \approx [b,\bar{b}]$.

Note that all rule instances of system KSg are sound [5]; the proviso that the context hole only appear in the scope of an even number of negations is automatically ensured if formulae are always converted to their negation normal form.

Each of the rule schemata in fact expresses an implication $T \Rightarrow R$ inside a context $S\{\ \}$. Due to the duality of implication, every rule has a dual rule expressing the implication $\overline{R} \Rightarrow \overline{T}$: we can take a sound *down* rule of the form

$$\rho{\downarrow}\,\frac{S\{T\}}{S\{R\}}$$

flip it around and negate to obtain another sound *up* rule of the form

$$\rho{\uparrow}\,\frac{S\{\overline{R}\}}{S\{\overline{T}\}}$$

If we complement the four rules of system KSg with three dual up rules (switch is its own dual), we obtain a new system, called SKSg (for *symmetric* KSg), whose inference rules are shown in Figure 3. Note that such a dualization of rules is not usually possible in the sequent calculus: many of its rules are branching, and hence inherently top-down asymmetric.

In a sense, however, SKSg is not more powerful than KSg: as shown in [5], each proof containing up rules can be converted into one containing only down rules; the up rules are *admissible.*

As a final remark, it can be shown [5, 16] that the identity rules can be reduced to their atomic form, i.e. every application of the rules $\mathsf{i}{\downarrow}$ and $\mathsf{i}{\uparrow}$ can be replaced by a sequence of applications of the switch rule and the atomic identity rules

$$\mathsf{ai}{\downarrow}\ \frac{S\{\mathtt{tt}\}}{S[a,\bar{a}]} \qquad \text{and} \qquad \mathsf{ai}{\uparrow}\ \frac{S(a,\bar{a})}{S\{\mathtt{ff}\}}$$

For a thorough exposition of the calculus of structures, the reader is referred to [11], which is the original paper on the calculus of structures. Systems for classical logic in the calculus of structures are treated in [5], linear logic in the calculus of structures is investigated in [23], and modal logics are the subject of [22] and [14].

2.3 A GraPE Session

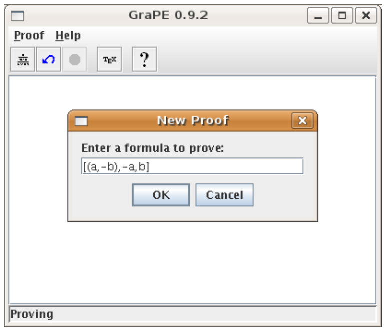
Now we will show how GraPE can be used to prove two simple propositional theorems in system KSg and system SKSg, before we go on to explain the program's inner workings.

GraPE can be downloaded from the official project homepage [25], where one can also find installation instructions. When GraPE is invoked, only the frontend is started at first. The user chooses a system description file for the system they want to use; we will show later how to create such a description for system KSg, for the moment we will simply assume that it is already available (and in fact a ready-to-use copy is included in the GraPE distribution as the file `ksg.xml`). The prover corresponding to the chosen system is then started in the background, and GraPE prompts the user for a theorem to prove.

Suppose we want to prove the simple propositional tautology

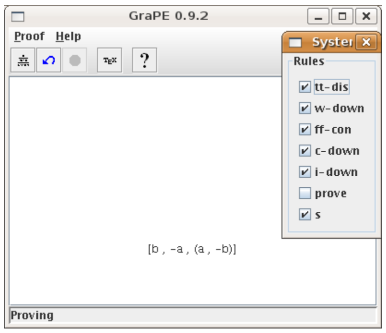
$$((a \to b) \wedge a) \to b$$

which is an instance of *modus ponens*. Written in KSg's syntax, this becomes $[(a,\bar{b}),\bar{a},b]$, and this is what we enter (for technical reasons, the overline $\bar{a}$ is entered as a minus symbol $-a$):



Enter a formula to prove:
[(a,-b),-a,b]

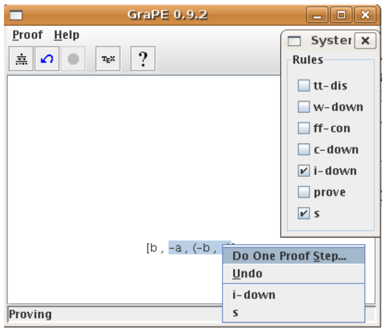
GraPE starts up the backend theorem prover, displays the derivation, which currently only consists of a single structure, and a palette for (de-)activating inference rules, the so-called *rule chooser*.



- [✓] tt-dis
- [✓] w-down
- [✓] ff-con
- [✓] c-down
- [✓] i-down
- [ ] prove
- [✓] s

```
[b , -a , (a , -b)]
```

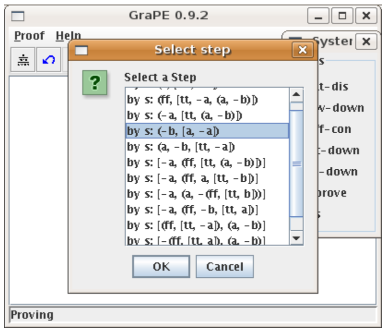
Our goal is to move the atom `a` and its negation `-a` closer together so that we can apply the interaction rule. To achieve this, we apply the switch rule to `[-a,(a,-b)]`. So we mark it and ask for possible inference steps.



- [ ] tt-dis
- [ ] w-down
- [ ] ff-con
- [ ] c-down
- [✓] i-down
- [ ] prove
- [✓] s

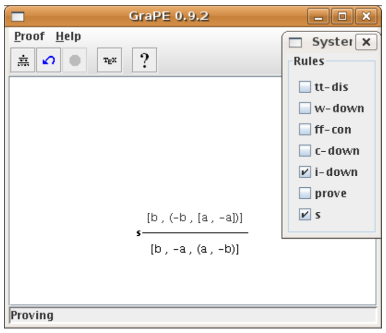
```
[b , -a , (-b ,
```

The underlying prover computes the possible steps and the results of applying them, and GraPE displays them in a list for the user to select one. In general, the number of possible steps to choose from can be quite overwhelming, so the user can decide to ignore some rules by deactivating them in the rule choosing palette – for our derivation, we have deactivated all rules except interaction and switch.

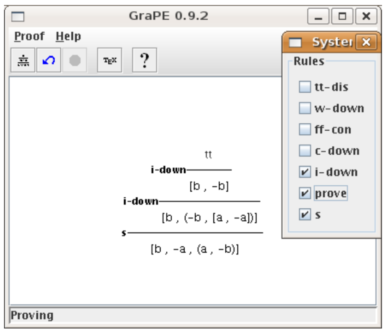


```
by s: (ff, [tt, -a, (a, -b)])
by s: (-a, [tt, (a, -b)])
by s: (-b, [a, -a])
by s: (a, -b, [tt, -a])
by s: [-a, (ff, [tt, (a, -b)])]
by s: [-a, (ff, a, [tt, -b])]
by s: [-a, (a, -(ff, [tt, b]))]
by s: [-a, (ff, -b, [tt, a])]
by s: [(ff, [tt, -a]), (a, -b)]
by s: [-(ff, [tt, a]), (a, -b)]
```

After we have selected the proof step we wish to take (namely, rewriting $[-a,(a,-b)]$ to $(-b,[a,-a])$), the derivation is augmented by this new step and then again displayed.



Proceeding in this manner, we can finally obtain a complete proof similar to this one:



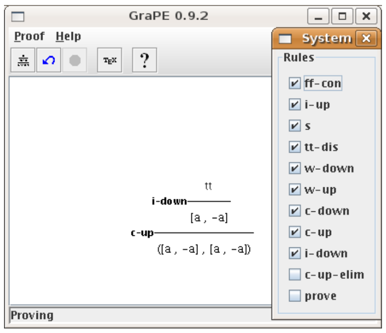
This derivation could now be exported as TEX [18] code (using the LATEX [19] macro package and the `bussproofs` proof tree style [6]). The details of how to translate an inference system's formulae into the TEX typesetting language are also part of the system description file.

This small example provides some evidence that a graphical user interface is indeed very useful when constructing proofs with deep inference: mouse-based selection of the redex to consider seems a lot more natural and convenient to us than specifying it as a character range or a position in an abstract syntax tree, as would perhaps be necessary in a command-line based proof editor. This contrasts with the sequent calculus, where inference rules can only be applied to toplevel connectives, leading to fewer application positions, and thus perhaps less to be gained from using a graphical interface.

Also note that with our implementation of system KSg we could have avoided the (admittedly marginal) burden of manually constructing the proof by relying on automatic proof search instead, which is represented by the *prove* rule. In this special case, proof search indeed finds exactly the same proof we just constructed manually.

Proof search (though not traditionally considered a form of proof manipulation) is an example of a higher-level proof transformation rule that can be implemented by the backend and presented to the user in the same manner as a simple inference rule.

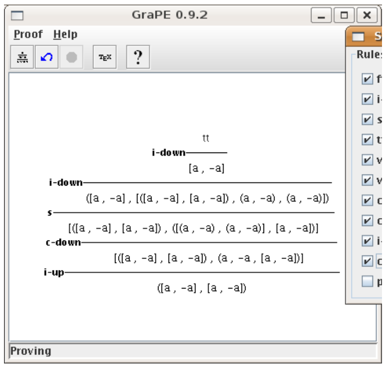
As another example, let us prove the theorem $([a, \bar{a}], [a, \bar{a}])$ in system SKSg. Using rules c↑ and i↓, a proof is quickly accomplished:



```
            tt
i-down─────────
         [a , -a]
c-up─────────────────────────
    ([a , -a] , [a , -a])
```

- [✓] ff-con
- [✓] i-up
- [✓] s
- [✓] tt-dis
- [✓] w-down
- [✓] w-up
- [✓] c-down
- [✓] c-up
- [✓] i-down
- [ ] c-up-elim
- [ ] prove

Now we can use the proof transformation rule `c-up-elim` to eliminate the instance of c↑ and replace it by an equivalent sub-derivation consisting of other rules. This yields the following derivation:



```
                        tt
            i-down──────────
                      [a , -a]
i-down──────────────────────────────────────────────────────────
     ([a , -a] , [([a , -a] , [a , -a]) , (a , -a) , (a , -a)])
s───────────────────────────────────────────────────────────────
  [([a , -a] , [a , -a]) , ([(a , -a) , (a , -a)] , [a , -a])]
c-down──────────────────────────────────────────────────────────
         [([a , -a] , [a , -a]) , (a , -a , [a , -a])]
i-up────────────────────────────────────────────────────────────
                  ([a , -a] , [a , -a])
```

While this derivation is certainly not the shortest or most elegant one, it serves to illustrate the ease with which GraPE allows to incorporate proof transformation rules.

In the rest of the paper we will give a more thorough description of the architecture and implementation of GraPE with the above sample proofs serving as our running examples.

3 The Big Picture

Let us start our exposition with a bird's eye overview of GraPE's architecture, as it is pictured in Figure 4. We already mentioned that GraPE consists of a graphical frontend and a theorem prover backend. The frontend, which is implemented in the Java programming language [10], is general and independent of any concrete inference system. The backend, on the other hand, would generally be tuned for a certain kind of (or even only one) inference system. In

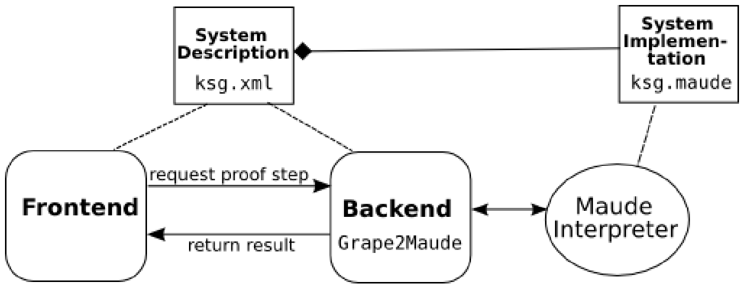


Figure 4: A Schematic Overview of GraPE's Architecture

the example scenario depicted here, the frontend is connected the `GraPE2Maude` backend which interfaces to the Maude implementation of an inference system (KSg in this case).

However, the frontend can be used with different systems implemented by different backends. For this purpose, GraPE defines a generic inference system description language and a standard protocol that frontend and backend use to communicate.

The system description needs to specify:

- the system's syntax, i.e. the available logical constants and operators; for each operator, it specifies
  - information about its abstract syntax, i.e. its arity, and whether it is commutative or associative (if applicable)
  - information about its concrete syntax, i.e. its precedence and its operator symbols
  - information about its presentation on-screen and for $\TeX$ output
- the available rules; since they are implemented by the backend, the frontend only needs to know their names
- the backend implementing this system

A more detailed description of the format of system description files, as well as a concrete example, is the subject of section 5.

The frontend itself is designed according to the MVC (*Model - View - Controller*) architectural pattern [21]. This pattern separates an application into three main components, unsurprisingly called *model*, *view*, and *controller*, which can be characterized as follows:

Model The model handles the underlying data the program works with, arranging them in data structures that correspond to the real-world or conceptual entities being modeled.

$$\begin{tikzcd}
 & & & \mathsf{c}\!\uparrow\text{-}\mathsf{elim} \arrow[dll] \arrow[drr] & & & & \\
 & \mathsf{c}\!\uparrow \arrow[dl] \arrow[dr] & & & & \mathsf{i}\!\uparrow \arrow[dl] \arrow[dr] & & \\
([a,\overline{a}],[a,\overline{a}]) & & \mathsf{i}\!\downarrow \arrow[dl] \arrow[dr] & & ([a,\overline{a}],[a,\overline{a}]) & & \mathsf{c}\!\downarrow \arrow[dl] \arrow[dr] & \\
 & [a,\overline{a}] & & \mathsf{tt} & & \cdots & & \cdots
\end{tikzcd}$$

Figure 5: Outline of the internal representation of a derivation tree

View The view renders the model into a graphical or textual representation suitable for interaction with the user.

Controller The controller handles user interaction and interprets it as requests to change the model. It forwards these requests to the model, and then updates (if necessary) the view to correspond to the changed model.

In GraPE, the entity being modeled is a derivation tree in some inference system. The inference system is modeled as a collection of objects representing operators and inference rules, the derivation tree is modeled as a thee-tiered abstract syntax tree.

The lower tier describes individual formulae with nodes corresponding to logical operators and subtrees to operands. The middle tier describes the application of inference rules with nodes corresponding to inference rules and subtrees to premises and conclusions. The higher tier, finally, describes the application of transformation rules with nodes corresponding to transformation rules and subtrees to their input and output derivations.

For example, the internal tree corresponding to the second example above, in which we first constructed a proof tree for $([a,\overline{a}],[a,\overline{a}])$ using the $\mathsf{c}\!\uparrow$ rule and then later eliminated it using the `c-up-elim` transformation rule, can be roughly pictured as in Figure 5: the left subtree of the root represents the original proof (to save some space, the full syntax trees for the individual structures have been omitted), whereas the right subtree corresponds to the $\mathsf{c}\!\uparrow$-free derivation, which due to its size is only partly depicted.

This internal derivation tree is presented to the user in the form of a GUI element, or as a piece of TEX code. In terms of the MVC pattern, these two representations are two different views of the same model. A further (though less typical) example of a view is the syntax in which the user enters formulae to be proved. The declarations necessary to specify how these three views can be constructed take, as we will see later, up the bulk of most inference system descriptions.

The final piece of our big picture of the GraPE frontend is the controller. This is the component which is responsible for handling user interaction and communicating with the backend. In GraPE, the most important kind of user interaction is when the user selects a redex and asks for rewrites. The interplay between the user and the system, as well as between different components of the system can be summarized as follows:

1. The user selects a redex using the mouse; while she is dragging the mouse, the controller makes sure that the selection always corresponds to a syntactically well-formed subformula. The information about what constitutes a well-formed subformula is obtained from the model.
2. By mouse click, the user asks for a list of possible rewrites. The controller hands the currently selected redex and the list of currently activated rules to the backend, which computes all possible rewrites.
3. The list of possible rewrites (itself a part of the model) is displayed in a dialog box (the corresponding view) for the user to choose from.
4. Once the user has made her choice, the derivation tree's model is changed to reflect it, then the view is updated to display the new tree.

This general description will be fleshed out with more details below using our running examples from the introduction.

4 Implementing System KSg

Before we show how to describe system KSg for GraPE, we first present its implementation as a Maude module. This section closely follows the corresponding sections from [16] and [7].

4.1 A Short Introduction to Maude

Maude is a language and interpreter for declarative programming in rewriting logic and membership equational logic. The basic unit of specification and programming is the module. In Maude, there are three kinds of modules: functional modules, system modules, and object oriented modules. Our implementation will only use functional and system modules. A module can import another module by means of the `protecting` declaration.

From a programming point of view, a functional module is an equational style functional program with user-definable syntax, in which a number of sorts, their elements, and functions on those sorts are defined. Functions can be given definitions through equations viewed as simplification rules to be applied in the left to right direction. These rules are assumed to form a terminating and confluent term rewriting system.

A system module, on the other hand, can be seen as a declarative style concurrent program, again with user-definable syntax. System modules can include rewrite rules, which are, however, not assumed to be terminating or confluent.

Maude's interactive interpreter shell provides the `reduce` command, which uses a functional module's equational rules to reduce a given term to normal form. This is, of course, not possible for a system module. Here one can use the `search` command, which for two given terms searches for a sequence of rewritings transforming one into the other by breadth-first search.

An important aspect of Maude is its support for reflection: modules, sorts, terms, and rewrite rules can be represented in the Maude language itself as values of sort `Module`, `Sort`, `Term`, and `Rule`, respectively. These values can be used, inspected, and decomposed using builtin accessor functions; for our purposes, the two most interesting ones are `metaReduce` and `metaSearch`, which mimic the functionality of the interpreter commands `reduce` and `search`, but can be used inside other function definitions.

4.2 System KSg in Maude

We first give an implementation of system KSg in Maude, which exploits the above-mentioned parallel between deep inference and term rewriting: KSg structures are represented as Maude terms of sort `Structure` with the logical operators defined as Maude operators, the equational theory is implemented using equational logic, and the inference rules correspond to rewrite rules.

4.2.1 Representing Structures

For example, the negation, conjunction, and disjunction operators and the two constants for KSg are declared as follows:

```
op tt    : -> Structure .
op ff    : -> Structure .
op -_    : Structure -> Structure .
op {_,_} : Structure Structure -> Structure [assoc comm id: tt] .
op [_,_] : Structure Structure -> Structure [assoc comm id: ff] .
```

Note that operand positions are indicated by underscores (_) in the declaration. Operator attributes like commutativity and associativity or the existence of units are provided as so-called operator attributes inside square brackets. This not only frees the user of having to explicitly give equations for these properties, but also ensures that Maude will use its built-in optimized algorithms for associative and commutative matching when rewriting such structures. Also note that the conjunction operator has to be written with curly brackets, since round parentheses have a special syntactic status in Maude.

Given the above declarations, the formula

$$[\overline{[\overline{a}, b]}, \overline{a}, b]$$

can be represented as the Maude term

```
[-[- a, b], [- a, b]]
```

4.2.2 Representing the Equational Theory

Now assume the following two equations, one De Morgan law and the involution law, are given:

```
eq - [ S:Structure , T:Structure ] =
                          { - S:Structure , - T:Structure } .
eq - - S:Structure = S:Structure .
```

Then the following short command line transcript shows how to reduce a term to normal form

```
Maude> reduce [-[- a, b], [- a, b]] .
reduce in SKSg : [- [b,- a],[b,- a]] .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
Result Structure: [b,[- a,{a,- b}]]
```

The result corresponds, of course, to the negation normal form $[(a,\bar{b}),\bar{a},b]$ of the input formula.

4.2.3 Representing the Inference Rules

If we further define the switch rule as a rewrite rule

```
rl [switch] : [T:Structure,{R:Structure, S:Structure}] =>
                       {S:Structure,[R:Structure, T:Structure]} .
```

we can search for possible rewrites of our above term:

```
Maude> search [b,[- a,{- b,a}]] =>1 S:Structure
(...)
Solution 11 (state 11)
states: 12 rewrites: 22 in 0ms cpu (0ms real) (~ rewrites/second)
S:Structure --> [b,{- b,[a,- a]}]
(...)
```

This result shows that the term `[b,[-a,{-b,a}]]` (corresponding to structure $[a,\bar{a},(a,\bar{b})]$) can be rewritten to `[b,{-b,[a,-a]}]` (corresponding to $[b,(\bar{b},[a,\bar{a}])]$) in one step. Incidentally, this was the first rewrite step chosen in our introductory example derivation.

Further Maude commands exist to display more details about a particular rewrite; the curious reader is referred to [16].

The complete definition of KSg's syntax, the equational theory, and the inference rules are given in Figure 6, with one functional module (`KSg-Signature`) for the sort and operator definitions, another functional module (`KSg-NNF`) for the equational theory, and a system module (`KSg-Inf`) for the inference rules.

This implementation closely follows the formal definition given earlier, if we keep in mind that the Maude rewrite rules correspond to a bottom up reading of the inference rules.

```
fmod KSg-Signature is

  sorts Atom Unit Structure .
  subsort Atom < Structure .
  subsort Unit < Structure .

  op tt    : -> Unit .
  op ff    : -> Unit .
  op -_    : Structure -> Structure .
  op [_,_] : Structure Structure -> Structure [assoc comm id: ff] .
  op {_,_} : Structure Structure -> Structure [assoc comm id: tt] .

  ops a b c d e f g h i j : -> Atom .

endfm

fmod KSg-NNF is

  protecting KSg-Signature .

  vars R T : Structure .

  eq - tt = ff .
  eq - ff = tt .
  eq - [ R , T ] = { - R , - T } .
  eq - { R , T } = [ - R , - T ] .
  eq - - R  = R .

endfm

mod KSg-Inf is

  protecting KSg-Signature .

  var A : Atom .
  var R T U  : Structure .

  rl  [i-down]  : [ A , - A ]        => tt .
  rl  [s]       : [ { R , T } , U ]  => { [ R , U ] , T } .
  rl  [w-down]  : R                  => ff .
  rl  [c-down]  : R                  => [ R , R ] .
  rl  [tt-dis]  : tt                 => [ tt , tt ] .
  rl  [ff-con]  : ff                 => { ff , ff } .

endm
```

Figure 6: Maude Implementation of System KSg

There are, however, two notable differences: the i↓ rule is only applied to atoms, and two equations of the equational theory are instead implemented as rewrite rules, namely `tt-dis` and `ff-con`.

Regarding the first difference, applying i↓ only to atoms (known as "atomic interaction") yields a more straightforward implementation. To see why, consider the following (incorrect) definition of full i↓:

```
rl [i-down] : [R:Structure, - R:Structure] => tt .
```

We would like this rule to apply, say, for `R = {a,b}`. But since we only deal with structures in negation normal form, `-R` would be `[-a,-b]`, and the term `[R,-R]`, now of the form `[{a,b},[-a,-b]]`, would not match the left hand side of the rule. For atoms, such a situation can never occur, and the simple implementation works.

Correcting this problem is not impossible, but makes the code more complicated. And as mentioned in the introduction, using only atomic interaction does not restrict the range of possible derivations.

For the second difference, recall that Maude treats the equations in functional modules as rewrite rules from the left to the right. If we were to give the equation

```
eq { ff , ff } = ff
```

Maude would, for example, reduce the term `[a,{ff,ff}]` to `[a,ff]` and further to `a`.

However, it would never use the equation from the right to the left, converting `ff` to `{ff , ff}`, since this would clearly lead to a non-terminating rewrite system. Thus we have to provide this case as an explicit rewrite rule.

In the light of these two differences, we should actually call this system KSg' to indicate it is not the original system KSg anymore. Still, we can construct all derivations in KSg' that we could construct in KSg, so we will not distinguish between the two.

4.3 System SKSg in Maude

If we want to extend the above modules to an implementation of system SKSg, a complication arises: while we can reuse the modules for the signature and the equational theory, some of the up rules cannot be implemented as Maude rewrite rules.

Take, for example, the rule w↑. We might want to implement it as

```
rl [w-up] : tt => R:Structure .
```

This definition, however, is rejected by Maude: variables can never be introduced on the right hand side of a rewrite rule. Intuitively, this makes sense, as the interpreter would not know what to substitute for `R` when applying this rule. A possible solution would be to *don't-know* non-deterministically select a structure and later backtrack if it turns out not to be the right choice; this is, however, not currently implemented in Maude.

The same problem arises with the definition of $\mathsf{i}\uparrow$. Both of these rules have been omitted from our Maude implementation, which is a simple extension of the KSg implementation obtained by adding a rewrite rule `c-up` for inference rule $\mathsf{c}\uparrow$. This yields a modified version of SKSg, which we call system SKSg'.

The difference between the two systems is not as big as one might think: as shown in [5], all the "up" rules are admissible, i.e. any proof that uses them can be transformed into one that does not. For $\mathsf{w}\uparrow$ and $\mathsf{c}\uparrow$, an even stronger result is shown: any *derivation* (not just any *proof*) that uses them can be transformed into one that does not.

Hence, although the difference between SKSg' and SKSg is bigger than between KSg' and KSg, we still will not normally distinguish between them for the purposes of this presentation.

4.4 Representing Transformation Rules

The techniques used so far are sufficient for implementing inference rules. In order to implement transformation rules that affect entire derivations, however, we need additional support for reifying derivations so that these rules can operate on them. We show how this can be done by extending the above approach.

We additionally introduce a Maude operator `_>[_]>_` which allows us to express derivations; for example, the second introductory example derivation

$$\mathsf{c}\uparrow\,\frac{\mathsf{i}\downarrow\,\dfrac{\mathit{tt}}{[a,\overline{a}]}}{([a,\overline{a}],[a,\overline{a}])}$$

would be represented as

```
              {[a, - a], [a, - a]}
>['c-up]>     [a, - a]
>['i-down]>   premise(tt)
```

where inference rules are referenced by their names written in Maude's *quoted identifier* syntax.

This representation could now be used for any inference in system SKSg; however, we would like it to be uniform over different inference systems with different logical operators. Hence we make use of the meta-level facilities of Maude and declare the derivation operator to be an operator on `Terms` (independent of the concrete sort used to represent formulae in a specific module); this suggests the following declarations for the derivation operator:

```
sort Derivation .

op premise : Term -> Derivation .
op _>[_]>_ : Term Qid Derivation -> Derivation .
```

Now our above example can be rewritten to use `Terms`; the concrete syntax of `Terms` is very cumbersome to read, but fortunately there is a Maude builtin function `upTerm`, which converts a given object-level term into its meta-level representation. The example thus becomes:

```
               upTerm({[a,- a], [a,- a]})
  >['c-up]>    upTerm([a,- a])
  >['i-down]>  premise(upTerm(tt))
```

Such a representation allows us to formulate the `prove` rule for system SKSg as a rewrite rule similar in spirit to the inference rules:

```
rl [prove]   : premise(T) =>
              derivationOfSearchPath('SKSg-Inf, T, upTerm(tt)) .
```

To see how this rule works, consider the following prefix of the above example derivation:

```
               upTerm({[a,- a], [a,- a]})
  >['c-up]>    premise(upTerm([a,- a]))
```

The `prove` rule can be applied to this derivation, matching the premise. The utility function `derivationOfSearchPath` now constructs a sequence of rewrites from `[a,- a]` to `tt` using the meta-level function `metaSearchPath` mentioned before, and returns it as a new derivation:

```
               upTerm([a,- a])
  >['i-down]>  premise(upTerm(tt))
```

This new derivation is then spliced into the original derivation, replacing its premise and yielding the complete derivation we have seen before.

5 Generic Description of Inference Systems

Having described the implementation of system KSg in Maude, we are now ready to give its system description. The description of any inference system needs to bridge the gap between the internal abstract representation of a derivation (the model) and the external representation for the user (the view). In particular, it defines

- the abstract syntax of formulae as represented internally,
- the concrete syntax used for manual formula input by the user,
- the available rules of inference and transformation,
- the graphical display format of formulae and derivations,
- the backend to use,

- translation rules for exporting derivations as TeX documents.

All of these definitions are bundled together into an XML [9] file which is accessible for both the frontend and the backend.

5.1 Describing Syntax and Display Format

5.1.1 General Operator Attributes

Formula syntax is described by a list of operator declarations. GraPE supports unary and binary operators. Unary operators can be prefix (like the negation operator "$\neg$"), postfix, or outfix (like the parentheses "( )"). Binary operators can be infix (like the classical conjunction operator "$\wedge$") or outfix (like system KSg's conjunction operator "(,)"). Constants are treated as operators without operands.

Each operator has a unique numerical precedence, which is a natural number, where higher numbers indicate less tight binding, and an internal ID. If not given explicitly, operators are automatically assigned numerically decreasing precedences in the order of listing, that is operators declared earlier bind less tight; if no explicit internal ID is given, one is synthesized.

For each of an operator's operand positions, one can specify whether this position contains a single operand (the default) or a list or multiset of operands, and what maximum precedence to expect in this position.

For instance, assume that we declare the classical negation operator $\neg$ to have a precedence of 1; if we declare its operand position to also have a maximal precedence of 1, the negation operator can have another negated expression as its argument (as in $\neg\neg a$). If, however, the operand position has a maximal precedence value of 0, the operand cannot have $\neg$ as its topmost operator anymore.

As another example, consider the turnstile operator $\vdash$ from one-sided sequent calculus. It is a unary operator, but its (single) operand is in fact a multiset of operands. This fact can be directly expressed by declaring the operand position to be a comma-separated multiset, ensuring that $\vdash a, \neg a$ is indeed a valid expression.

5.1.2 Graphical Representation of Formulae

It was mentioned above, that the internal abstract syntax tree representation of the current derivation (and with it the representation of every individual formula) lies at the heart of the model. This model can produce a view of itself, that is a textual or graphical representation of the data it contains. In fact, there is not a single view, but there are *three* different views:

1. the **input** view: this is the representation users employ when manually entering formulae via the keyboard. In this representation, graphically more complex operators can be rendered as "ASCII art"; for example, the classical conjunction operator $\wedge$ might be entered as /\.

2. the **output** view: this is the representation GraPE employs when displaying formulae on the screen, for example inside the derivation tree. It should look as natural as possible to the user, using the standard operator symbols whenever feasible.
3. the **TeX** view: this representation can be used to export formulae for use inside a LaTeX document.

The system description must provide enough information for the model to be able to generate any of these three views. Therefore, it must, for every operator symbol, specify three representations: two Unicode strings for the input and output views, and one fragment of TeX code for the third view. Often, these representations will, however, be the same; hence GraPE provides suitable defaults as specified below.

5.1.3 Individual Operator Attributes

Apart from the general attributes described above, each kind of operator has special attributes relating to its syntactic representation and display. We give here a quite detailed list; the reader might consider skipping it on first reading and refer back to it when needed.

- for unary prefix operators:

  symbol the operator symbol's output view

  input the operator symbol's input view; by default, this is the same as **symbol**

  tex the code pattern to be used when creating the TeX view; instances of the TeX argument placeholder `#1` will be replaced by the code generated for the (one and only) argument

  To avoid ambiguities, the operand position's maximal precedence must never exceed the operator's own precedence.

  For example, the negation operator for system `KSg` is declared by

  ```
  <unary-prefix symbol="-" tex="\overline{#1}"/>
  ```

  All other attributes are given their default values; in fact, the complete definition would be

  ```
  <unary-prefix id="-_" symbol="-" input="-"
                tex="\overline{#1}" prec="14"
                op="singleton" opprec="14"/>
  ```

  declaring the negation operator to be a unary prefix operator with internal id `-_`, output symbol $-$, input symbol `-`, TeX code `\overline{#1}` and precedence 14. Its operand is declared to be a singleton of maximal precedence 14.

Note that the precedence value of fourteen is computed because there are two other operators and twelve constants (two logical constants and ten predefined propositional letters), among which the unary negation has the least binding power. The operand position's precedence of 14 means that applications of this operator can be stacked (if it were, for example, 13, then only expressions with tighter binding operators could occur as its argument).

As another example, take the turnstile operator from one-sided sequent calculus (bundled with the GraPE distribution as system `GS1p`). Here, the declaration is

```
<unary-prefix symbol="&vdash;" input="|-"
              tex="\vdash #1" operand="multiset,"/>
```

Thus, the operator symbol is to be input as `|-`; the graphical representation is `&vdash;`, which is a character entity predefined by GraPE that resolves to the Unicode [1] character code for the turnstile. The operand position, finally, is specified as being a `"multiset,"`, i.e. a comma-separated multiset.

- for unary postfix operators: the possible attributes are the same as for prefix operators. To avoid ambiguities, the operand position of a unary postfix operator can only be a singleton of precedence not greater than the operator itself.

- for unary outfix operators: Similar to the other unary operators; note that we now do not have a single operator symbol but two (the start delimiter and the end delimiter). Each operator symbol has of course three different views, yielding a total of six view-related attributes (**startsym**, **endsym**, **instart**, **inend**, **texstart**, and **texend**).

  For the normal grouping parentheses, for example, we declare

  ```
  <unary-outfix startsym="(" endsym=")" grouping="yes"/>
  ```

  Instead of the pair of `startsym`/`endsym` declarations, we could also have given a single `symbol` attribute, that has to be exactly two characters long, the first being the start and the second the end delimiter.

  The `grouping` attribute is special for unary outfix operators. Sometimes, formulae constructed by the prover may be syntactically ambiguous and their correct interpretation has to be forced by bracketing. In general, we cannot assume that the bracketing operator will always be the normal parenthesis (in `KSg`, for example, this has quite a different meaning). Hence the user has to explicitly specify a grouping operator if one is needed.

As with other outfix operators, the operand position's precedence defaults to the highest precedence occurring in the system description, i.e. any kind of expression can occur as an operand.

- for binary infix operators:

  symbol the operator symbol's output view (as for unary prefix operators)

  input the operator symbol's input view; by default, this is the same as **symbol**

  tex specifies the TEX code to use for the operator; this is *not* a code pattern as above, but simply a binary operator to be put in between operands

  assoc whether the operator is associative; the default is `yes`

  comm whether the operator is commutative; again, the default is `yes`

  leftprec, leftop specifies the type and precedence of the left operand position

  rightprec, rightop same for the right operand position

  It should be noted that the **comm** attribute only describes whether the *frontend* should treat this operator as commutative. In many sequent calculus systems, for instance, the commutativity of the classical conjunction operator $\wedge$ is not implicit as in system KSg, but built into the structural rules of the system. For such systems, the frontend should not know about the operator's commutativity, and it would hence be declared with `comm="no"`.

  For example in system GS1p, the conjunction operator is defined as

  ```
      <binary-infix symbol="&wedge;" input="/\" tex="\wedge"
                    comm="no"/>
  ```

  Note again the use of a predefined entity, `&wedge;`, for the operator symbol.

  For associative operators, the specifications for the two operand positions have to agree, since for example in the formula $a \wedge b \wedge c$, the subformula b is both the right operand of the first, and the left operand of the second conjunction.

- for binary outfix operators: all of the attributes for binary infix operators can be used, with the additional feature that the operator symbol consists of a start delimiter, a separator, and an end delimiter; as with unary outfix operators, these can be given either as three different attributes or as one three-character attribute

- for constants: constants have attributes `symbol`, `input`, and `tex` with the same meaning as for the other operators. For convenience, there is also an

```
<syntax>
  <unary-prefix  symbol="-" tex="\overline{#1}"/>
  <binary-outfix symbol="[,]"/>
  <binary-outfix id="con" symbol="(,)"/>
  <constant      symbol="tt"/>
  <constant      symbol="ff"/>
  <constants     symbols="a b c d e f g h i j"/>
</syntax>
```

Figure 7: XML definition of KSg's syntax

operator type `constants`, which allows to define several constants with similar attributes at once; this can be used, for example, to define a list of propositional variables (which are treated as constants by the GraPE parser).

The complete syntax specification for system KSg is given in Figure 7. Observe that the conjunction operator is given a special identifier, because it needs to be referred to from the backend-specific part of the description file, which will be given later.

For the reader who feels uncomfortable with this rather casual explanation of the operator declarations, we give a procedure for transforming a sequence of operator declarations into a context-free grammar in appendix A.

5.2 Describing Rules

The next section of the description file is devoted to describing the available rules. We distinguish between inference rules, which act upon single formulae, and transformation rules, which act upon whole derivations.

Since the matching and application of rules is entirely the responsibility of the underlying prover, the frontend only needs to know each rule's name (which will also be the one displayed by the GUI) and how to represent the name in TEX. Furthermore, it can be specified whether a rule should be enabled when the program starts up. By default, inference rules start out enabled, whereas transformation rules are disabled.

In contrast to formulae, whose textual representation is largely user-definable, the graphical representation of inference rules is always the same, following the well-established proof tree paradigm. For transformation rules, on the other hand, only the result of the transformation is represented on-screen, hiding the input proof tree.

The rule description section for system KSg is given in Figure 8.

5.3 Describing the Backend

Finally, we need to tell GraPE which backend to use and how to find it, and provide some parameters the backend needs. This part is very much depen-

```
<rules>
  <inference-rule name="i-down"   tex="$\mathsf{i}\!\downarrow$"/>
  <inference-rule name="s"        tex="$\mathsf{s}$"/>
  <inference-rule name="w-down"   tex="$\mathsf{w}\!\downarrow$"/>
  <inference-rule name="c-down"   tex="$\mathsf{c}\!\downarrow$"/>
  <inference-rule name="ff-con"   tex="$\mathsf{ff-con}$"/>
  <inference-rule name="tt-dis"   tex="$\mathsf{tt-dis}$"/>
  <transformation-rule name="prove"/>
</rules>
```

Figure 8: XML definition of KSg's rules

```
<backend class="grape.backend.grape2maude.GraPE2Maude">
  <load name="ksg.maude"/>
  <inferencer name="KSg-Inf"/>
  <normalizer name="KSg-NNF"/>
  <maudename id="con" name="{_,_}"/>
  <result-sort name="Structure"/>
</backend>
```

Figure 9: XML definition of KSg's backend

dent on the particular backend used, so we just give and explain the necessary declarations for KSg with the Grape2Maude backend (see Figure 9).

The `class` attribute of the `backend` element is required for all backend declarations; it tells the frontend which class implements the backend so that it can dynamically load this class at runtime. In our case, the backend class is part of the GraPE distribution, but that does not have to be the case. The user can write their own backend in Java, compile it to a `.class` file and insert its class name here, provided it conforms to the `Prover` interface specified below.

The next three lines tell Maude which files to load and the names of the inferencer and normalizer modules; the first one is supposed to contain the inference rules, whereas the second one is used to normalize formulae (compare with the module definitions given above in Figure 6).

Because the conjunction's syntax is different in Maude than it is in the frontend, we specify a `maudename` override, which uses the identifier introduced earlier to refer to the conjunction operator.

Finally, the `result-sort` declaration gives the name of the Maude sort that is used to represent formulae.

6 Putting It All Together

Armed with our knowledge about the implementation of system KSg, let us now revisit our initial example to complete our exposition of GraPE's internals.

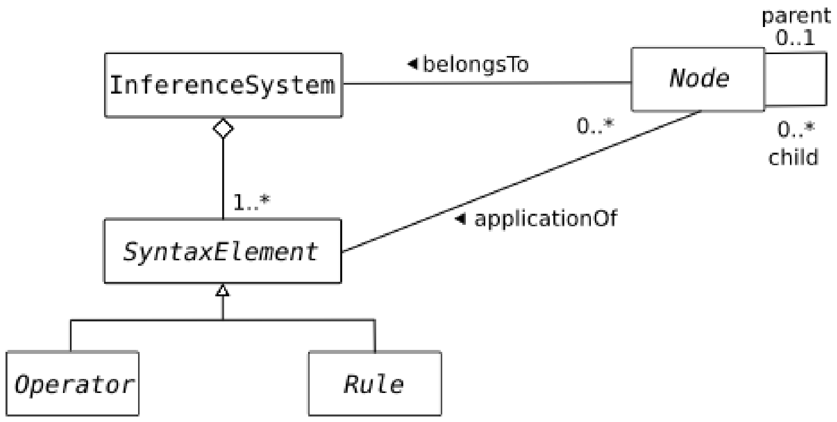


Figure 10: Class diagram of the relation between nodes, inference systems, and syntax elements

$$\begin{tikzcd}
 & & & {[\_,\_]} \arrow[dll] \arrow[d] \arrow[dr] & \\
 & (\_,\_) \arrow[dl] \arrow[dr] & & -\_ \arrow[d] & b \\
a & & -\_ \arrow[d] & a & \\
 & & b & &
\end{tikzcd}$$

Figure 11: Abstract syntax tree for $[(a, \bar{b}), \overline{a}, b]$

When the user selects the description file `sksg.xml`, GraPE reads it, creates a representing object for every operator or rule, and gathers them together in an object representing the inference system; this relationship is represented in the left part of the UML [20] class diagram in Figure 10. The operator and rule objects (which we will collectively call *syntax elements* in the following) serve two important purposes:

They implement the parser which parses formulae input by the user, and they can also build the graphical representation of a derivation. For example, if the user enters the formula `[(a,-b),-a,b]`, it is parsed by the operator objects into the syntax tree in Figure 11.

This syntax tree is implemented as a tree of `Node` objects, where each node contains a reference to its parent and a list of children, along with a reference to the operator whose application it represents and the surrounding inference

```
⟨⟨Interface⟩⟩
Prover
setParam (in name : String, in value : String) : void
start ( ) : void
normalize (in derivation : Node) : Node
findRewrites (in redex : Node, in rules : Collection⟨Rule⟩)
            : Collection⟨Node⟩
abort ( ) : void
```

Figure 12: The `Prover` interface

system (again, see Figure 10).

Now the backend is loaded dynamically according to the specifications of the description file. Since its actual type will not be known until run time, GraPE requires any backend to implement the interface `Prover` given in Figure 12. This interface guarantees the existence of five methods which together form the communication protocol between frontend and backend:

Method `setParam` is used to pass backend-specific settings given in the description file on to the prover. For example, the names of the module files to load as specified by the `load-file` directives in the description file are passed to the backend via `setParam` invocations. Once this setup is completed, `start` is called to initialize the prover.

Before displaying the freshly started derivation for the first time, GraPE first normalizes it by passing it through the backend's `normalize` method. The `GraPE2Maude` backend implements this normalization as reduction using the equations from the specified `normalizer` module, i.e. `KSg-NNF` in our case. The formula $[(a, \bar{b}), \bar{a}, b]$ used in the introductory example is already in normal form, hence the normalization step does not change anything.

The normalized syntax tree (the model in terms of the MVC architecture) is then converted into graphical representation (the view), which just consists of the formula

$$[(a, -b), -a, b]$$

The conversion from abstract syntax tree to graphical representation is the second important purpose of the operator objects: since they store all the information about the operator type and symbols, they can construct the graphical view of a given node, provided the views of the node's children have already been constructed.

In the example above, first the views of the constant nodes `a` and `b` are constructed, which are given in the system description (a and b, respectively). Next, the unary negation nodes can have their views constructed: both of them have the same operator object, namely the object representing unary negation. This object, when given the view of the operand (i.e., the string a or b), constructs a new view by prepending the operator symbol $-$ to it, yielding $-a$ and $-b$. Now the conjunction operator can process its two operands' views to produce $(a, -b)$, which is then fed to the disjunction operator to yield the view of the whole formula, $[(a, -b), -a, b]$.

In addition to the character string to be displayed on-screen, the view also contains a list of all markable ranges inside the formula, i.e. of all character ranges that correspond to possible redices. A possible redex is either a subtree of the syntax tree or a contiguous run of subtrees of an associative operator's node. In the example above, $(a, -b)$ is a markable range and a possible redex (since it is a subtree), and so is $(a, -b), -a$, which corresponds to the possible redex $[(a, -b), -a]$, a contiguous run of subtrees. On the other hand, [(is not a markable range. After displaying a formula's view on the screen, the controller makes sure that the user can only select markable ranges with the mouse.

A final refinement are the so-called *active characters*: when displaying a formula with a binary commutative operator, this operator's symbol is made mouse-sensitive. A mouse click by the user then results in swapping the two operands of this operator, i.e. exchanging the position of the two corresponding subtrees in the model and then updating the view. In the example, all the commata are made mouse-sensitive, since both the conjunction and the disjunction operator were declared as being commutative.

Now let us retrace the steps of a typical user interaction as given in our "Big Picture" overview:

1. First, the user selects a redex. As mentioned, the system only allows selections which actually correspond to sensible redices, that is either a complete subtree or two or more contiguous subtrees of an associative operator node. Note that GraPE does not check whether there actually is a rewrite for a selected redex; this can be computationally expensive (as it involves communication with the backend), and is only done when the user actually requests a rewrite.

2. Assume the user selects the range corresponding to the possible redex

   $$[(\mathtt{a}, -\mathtt{b}), -\mathtt{a}]$$

   and then requests a rewrite. The frontend produces a `Node` object for the redex, assembles a list of all currently activated inference and transformation rules, and hands both to the backend's `findRewrites` method.

   The backend then checks whether there are any rewrites for this redex using the given collection of rules, and returns all of them as a collection of `Rewrite` objects.

   The internal representation of a rewrite, given as a UML class diagram in Figure 13, is more or less straightforward: the representing object needs to encapsulate information about the redex, the rule used, and the contracta.

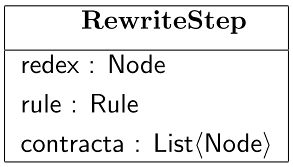


Figure 13: Class `RewriteStep`

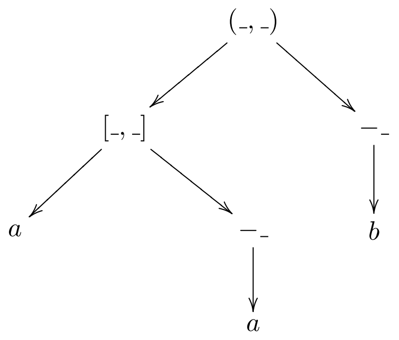


Figure 14: Abstract syntax tree for $([a, -a], -b)$

Note the plural here: a single rewrite can yield a *list* of results. This is simply a convenience for uniformly implementing branching rules. In system KSg, every rewrite will always have a single contractum; in system GS1p, however, some rules result in rewrites with two contracta.

3. The list of possible rewrites is displayed in a dialog box for the user to choose from. Again, we see here the influence of the ubiquitous MVC pattern: the `Rewrite` objects are part of the model, their graphical representation inside the dialog box is a view, and the code that interprets the user's selection takes over the role of the controller.

   In our example, the user requested an application of the switch rule, with the redex being rewritten to $([a, -a], -b)$; the syntax tree of this formula is given in Figure 14.

4. This tree is then plugged into the position formerly occupied by the redex, yielding the tree in Figure 15.

   This whole syntax tree is the premise of the rule application, whereas the conclusion is the syntax tree we saw before in Figure 11. The two of them are joined together by a rule node labeled with s to indicate an application of the switch rule, yielding the new derivation tree in Figure 16, which is our new model.

Applying further inference rules results in the addition of further inference nodes (i.e., nodes labeled with inference rules). The leftmost subtree of each

$$\begin{tikzcd}
 & & & {[\_,\_]} \arrow[dl] \arrow[dr] & \\
 & & {(\_,\_)} \arrow[dl] \arrow[dr] & & b \\
 & {[\_,\_]} \arrow[dl] \arrow[dr] & & -\_ \arrow[d] & \\
a & & -\_ \arrow[d] & b & \\
 & & a & &
\end{tikzcd}$$

Figure 15: Abstract syntax tree for $[([a, -a], -b), b]$

$$\begin{tikzcd}
 & & & & & & \mathsf{s} \arrow[dlll] \arrow[drr] & & & \\
 & & & {[\_,\_]} \arrow[dll] \arrow[d] \arrow[dr] & & & & & {[\_,\_]} \arrow[dl] \arrow[dr] & \\
 & {(\_,\_)} \arrow[dl] \arrow[dr] & & -\_ \arrow[d] & b & & & {(\_,\_)} \arrow[dl] \arrow[dr] & & b \\
a & & -\_ \arrow[d] & a & & & {[\_,\_]} \arrow[dl] \arrow[dr] & & -\_ \arrow[d] & \\
 & & b & & & a & & -\_ \arrow[d] & b & \\
 & & & & & & & a & &
\end{tikzcd}$$

Figure 16: Example for a derivation tree containing an inference node

inference node is a formula, namely the rule's conclusion, while the further subtrees might also be formulae or might themselves be inference nodes. Thus, the application of inference rules results in the tree growing at the lower right.

Applying a transformation rule, on the other hand, creates a new root node, whose first child (the input of the transformation) is the old derivation, and whose second child is the new derivation, i.e. the output of the transformation. Is this way, GraPE keeps a complete history of all applied inference and transformation rules, and the application of any rule can simply be undone by replacing the corresponding rule node with its leftmost child.

7 Discussion

We have described the design and implementation of GraPE, a graphical proof editor with support for generic description of inference systems, multiple prover backends, and an extended notion of proof manipulation rule, including both simple inference steps and more complicated proof search or proof transformation rules. We have shown how the distinction between a frontend handling graphical derivation display and user input and a backend doing tho actual work of theorem proving makes the editor very flexible and usable with a wide range of different inference systems, and how the use of Maude as a backend makes the implementation of new inference systems quite easy.

Although we think that GraPE is already very general, there are still some features missing, for example:

- First-order systems: The inference systems mentioned above are all for propositional logic; in fact, the syntax description rules of GraPE are not powerful enough to accommodate the more complicated structure of first-order formulae. We are planning to extend GraPE with the ability to handle first-order syntax in one of the next versions.

- Top-down derivation construction: This feature is sometimes very helpful in finding derivations, and it would nicely complement the existing bottom-up approach. Supporting it would probably entail some extensions to the current handling of the internal derivation tree.

- Automatic generation of description files and Maude modules: Since a lot of information is duplicated in the Maude implementation and the system description, it would be nice to have a utility that generates at least a skeleton of one from the other. Command line utilities to do this are currently under development.

- Other display styles (besides proof trees and chains): This is a more long-term goal; we would like to be able to use, for example, natural deduction or proof nets.

- For the `GraPE2Maude` backend, we would like to implement more sophisticated proof manipulations.

We think that these features can be integrated into GraPE as an extension of currently existing features without having to change any of the basic design decisions.

In the end, however, only the feedback of our users will show whether we have succeeded in developing a useful system.

References

[1] Joan Aliprand, Julie Allen, and Joe Becker, editors. *The Unicode Standard, Version 4.0*. Addison Wesley Publishing Company, 2003.

[2] David Aspinall et al. *Proof General Version 3.5*. Website at `http://proofgeneral.inf.ed.ac.uk/`.

[3] Janet Bertot and Yves Bertot. *CtCoq: A system presentation*. In Automated Deduction (CADE-13), volume 1104 of Lecture Notes in Artificial Intelligence, pages 231–234. Springer-Verlag, July 1999.

[4] R. Bornat and B. A. Sufrin. *Animating formal proof at the surface: the Jape proof calculator*. In The Computer Journal, 43(3), pages 177–192, 1999.

[5] Kai Brünnler. *Deep Inference and Symmetry in Classical Proofs*. PhD Thesis, Dresden University of Technology, 2003.

[6] Sam Buss. *bussproofs.sty LaTeX style*. Website at `http://www.math.ucsd.edu/∼sbuss/ResearchWeb/bussproofs/`.

[7] Manuel Clavel et al. *Maude Manual (Version 2.2)*. Technical report, Computer Science Laboratory, SRI International, 2006. Available online at `http://maude.cs.uiuc.edu/maude2-manual/`.

[8] Manuel Clavel et al. *The Maude 2.0 system*. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA-99)*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag LNCS 1631.

[9] *Extensible Markup Language (XML)*. Website at `http://www.w3.org/XML`.

[10] James Gosling et al. *The Java^{TM} Language Specification, Second Edition*. Addison-Wesley Professional, 2000.

[11] Alessio Guglielmi. *A system of interaction and structure*. Technical Report WV-02-10, Technical University of Dresden, 2002. To appear on *ACM Transactions on Computational Logic*.

[12] Alessio Guglielmi. *Deep Inference and the Calculus of Structures*. Website at `http://alessio.guglielmi.name/res/cos/`.

[13] T. Hallgren and A. Randa. *An Extensible Proof Text Editor*. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning*, number 1955 in LNAI, pages 70–84. Springer-Verlag, 2000.

[14] Robert Hein and Charles Stewart. *Purity through unravelling*. In Paola Bruscoli, François Lamarche, and Charles Stewart, editors, *Structures and Deduction*, pages 126–143. Technical University of Dresden, 2005.

[15] Steffen Hölldobler and Ozan Kahramanoğulları. *From the Calculus of Structures to Term Rewriting Systems*. Technical Report WV-04-03, Dresden University of Technology, 2004.

[16] Ozan Kahramanoğulları. *Deep Inference: Implementation, Reducing Nondeterminism, Language Design*.
Draft of PhD Thesis, available on the web at `http://www.wv.inf.tu-dresden.de/∼ozan/Papers/ozansthesis.pdf`, 2006.

[17] Ozan Kahramanoğulları. *The Calculus of Structures in Maude*. Website at `http://www.computational-logic.org/∼ozan/maude_cos.html`.

[18] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1994.

[19] Leslie Lamport. *LATEX: a Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 1994.

[20] Meilir Page-Jones. *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley, 2000.

[21] Trygve Reenskaug. *THING-MODEL-VIEW-EDITOR: an Example from a planningsystem*. Xerox PARC technical note, 1979.

[22] Charles Stewart and Phiniki Stouppa. *A systematic proof theory for several modal logics*. Technical Report WV-03-08, Dresden University of Technology, 2003.

[23] Lutz Strassburger. *Linear Logic and Noncommutativity in the Calculus of Structures*. PhD Thesis, Dresden University of Technology, 2003.

[24] Donald Syme. *A New Interface for HOL - Ideas, Issues, and Implementation*. The Computer Laboratory, University of Cambridge, 1995.

[25] *The GraPE Graphical Proof Editor*. Website at `http://grape.sourceforge.net`.

[26] Laurent Thery et al. *Real Theorem Provers Deserve Real User-Interfaces*. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, Washington D.C., December 1992.

A Transforming Operator Declarations into a Context-Free Grammar

To supplement the intuitive interpretation of the operator declarations given in section 5, we show now how to construct a context free grammar $G = (N, \Sigma, S, P)$ with non-terminals N, alphabet Σ, start symbol S, and productions P, from a list of such declarations.

Let $D = (D_1, \ldots, D_n)$ be a list of operator declarations given in the format specified above, with $\Pi = (\pi_1, \ldots, \pi_n)$ as the list of their respective precedences; let $\pi_m = \max_{1 \leq j \leq n} \pi_j$. Remember that the precedences are pairwise distinct.

First, we define the set of terminal symbols Σ. Start out with $\Sigma = \emptyset$. Then for each operator declaration D_i, $1 \leq i \leq n$,

- if D_i declares a unary prefix or postfix operator, add its operator symbol to Σ
- if D_i declares a unary outfix operator, add its start and end symbols to Σ
- if D_i declares a binary infix operator, add its operator symbol to Σ
- if D_i declares a binary outfix operator, add its start, separator, and end symbols to Σ
- if D_i declares a constant, add its symbol to Σ
- for any list or multiset operand position, add the separator symbol to Σ

The set of non-terminals is $N = \{N_j, LOp_j, Op_j, ROp_j, Rest_j \,|\, 1 \leq j \leq \pi_m\} \cup \{N_{err}\}$; the start symbol is N_{π_m}.

It remains to define the set of productions P. Initially, let P contain precisely one production for every non-terminal N_i and one for N_{err}.

For $i > 0$, this production is $N_i \rightarrow N_{i-1}$, for $i = 0$, it is $N_0 \rightarrow N_{err}$.

The production for N_{err} is $N_{err} \rightarrow N_{err}$, indicating a parse error (i.e., the parsed input did not match the constructed grammar).

Now for every operator declaration D_i, $1 \leq i \leq n$:

- If D_i declares a unary prefix operator $\star$, add the production

  $$N_{\pi_i} \rightarrow \star Op_{\pi_i}$$

  If the operand position of this operator is a singleton with precedence k, add the production

  $$Op_{\pi_i} \rightarrow N_k$$

  If the operand position is a list or multiset with separator "," and precedence k, add the two productions

  $$\begin{array}{lcl} Op_{\pi_i} & \rightarrow & N_k \\ Op_{\pi_i} & \rightarrow & N_k \, , \, Op_{\pi_i} \end{array}$$

- If D_i declares a unary postfix operator †, remove the production for N_{π_i} from P. Remember that a postfix operator's operand position can only ever be a singleton with a precedence less or equal to π_i. If the precedence equals π_i, add the three productions

$$\begin{array}{lcl} N_{\pi_i} & \rightarrow & N_{\pi_i-1} \mathit{Rest}_{\pi_i} \\ \mathit{Rest}_{\pi_i} & \rightarrow & \dagger \\ \mathit{Rest}_{\pi_i} & \rightarrow & \dagger \mathit{Rest}_{\pi_i} \end{array}$$

  If we were to naively follow the schema used with prefix operators, we would get a production of the form $N_{\pi_i} \rightarrow N_{\pi_i}\dagger$, i.e. a left-recursive rule. To make parsing easier, we want to avoid this kind of rule.

  If the operand's precedence is not equal to π_i, let it be k. Then add the two productions

$$\begin{array}{lcl} N_{\pi_i} & \rightarrow & N_k\dagger \\ N_{\pi_i} & \rightarrow & N_k \end{array}$$

- If D_i declares a unary outfix operator with start symbol "(", and end symbol ")", add the production

$$N_{\pi_i} \rightarrow (\ Op_{\pi_i}\ )$$

  The productions for the operand position are the same as with unary prefix operators.

- if D_i declares a binary infix operator $\circ$, remove the production for N_{π_i} from P. If the operator is associative, add

$$\begin{array}{lcl} N_{\pi_i} & \rightarrow & Op_{\pi_i} \circ N_{\pi_i} \\ N_{\pi_i} & \rightarrow & Op_{\pi_i} \end{array}$$

  and add operand position productions for Op_{π_i} as above (note that in this case, the two operand positions have the same attributes and hence can be treated by one production).

  If the operator is not associative, add

$$\begin{array}{lcl} N_{\pi_i} & \rightarrow & LOp_{\pi_i} \circ ROp_{\pi_i} \\ N_{\pi_i} & \rightarrow & LOp_{\pi_i} \end{array}$$

  and for each of LOp_{π_i} and ROp_{π_i}, add operand position productions as for Op_{π_i} before.

- if D_i declares an associative binary outfix operator with start symbol "(", separator symbol ",", and end symbol ")", add

$$\begin{array}{lcl} N_{\pi_i} & \rightarrow & (\ Op_{\pi_i}\ ,\ \mathit{Rest}_{\pi_i}\ ) \\ \mathit{Rest}_{\pi_i} & \rightarrow & Op_{\pi_i}\ ,\ \mathit{Rest}_{\pi_i} \\ \mathit{Rest}_{\pi_i} & \rightarrow & Op_{\pi_i} \end{array}$$

$$
\begin{array}{lcl}
N_{14} & \rightarrow & -\,N_{13} \\
N_{14} & \rightarrow & N_{13} \\
N_{13} & \rightarrow & [\,N_{14}\,,\mathit{Rest}_{13}\,] \\
N_{13} & \rightarrow & N_{12} \\
\mathit{Rest}_{13} & \rightarrow & N_{14} \\
\mathit{Rest}_{13} & \rightarrow & N_{14}\,,\mathit{Rest}_{13} \\
N_{12} & \rightarrow & (\,N_{14}\,,\mathit{Rest}_{12}) \\
N_{12} & \rightarrow & N_{11} \\
\mathit{Rest}_{12} & \rightarrow & N_{14} \\
\mathit{Rest}_{12} & \rightarrow & N_{14}\,,\mathit{Rest}_{12}
\end{array}
\qquad
\begin{array}{lcl}
N_{11} & \rightarrow & \texttt{tt} \\
N_{11} & \rightarrow & N_{10} \\
N_{10} & \rightarrow & \texttt{ff} \\
N_{10} & \rightarrow & N_{9} \\
N_{9} & \rightarrow & \texttt{a} \\
N_{9} & \rightarrow & N_{8} \\
\vdots & & \\
N_{0} & \rightarrow & \texttt{j} \\
N_{0} & \rightarrow & N_{err} \\
N_{err} & \rightarrow & N_{err}
\end{array}
$$

Figure 17: Context-free Grammar for the Syntax of KSg

For non-associative binary outfix operators, add instead

$$N_{\pi_i} \rightarrow (\ LOp_{\pi_i}\ ,\ ROp_{\pi_i}\ )$$

The productions for Op_{π_i} in the associative case and LOp_{π_i} and ROp_{π_i} in the non-associative case are handled as for infix operators.

- if D_i declares a constant κ, add

$$N_{\pi_i} \rightarrow \kappa$$

Obviously, the tuple $G = (N, \Sigma, S, P)$ is a context-free grammar, since the above procedure never introduces a rule with more than one non-terminal on the left-hand side.

In fact, G is almost an LL(1) grammar, that is it can be parsed by a top-down parser with one token of lookahead. The single exception is the rule for N_{err}, which is left-recursive.

We do not formally prove this statement but rather refer the reader to the GraPE source code, where a deterministic parser with one token of lookahead is constructed for any set of operator declarations. It treats the N_{err} case as an error, and hence recovers LL(1)-parsability.

As an example, we show in Figure 17 the grammar obtained from the operator declarations for KSg.