Taclets and the KeY Prover

Extended Abstract

Martin Giese

Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden
giese@cs.chalmers.se

Abstract. We give a short overview of the KeY prover, which is the proof system belonging to the KeY tool [1], from a user interface perspective. In particular, we explain the concept of *taclets* which is the basic building block for proofs in the KeY prover.

1 Introduction

The goal of the ongoing KeY Project [1] is to make the application of formal methods possible and effective in a real-world software development setting. One of the main products of the KeY Project is the KeY Tool, which allows the specification and verification of Java Card [8] programs. The KeY Prover is an integrated interactive and automated theorem prover that is used in the KeY tool to reason about programs and specifications.

The logic employed by the KeY prover is a dynamic logic (DL) for Java Card [2]. This can be viewed as a kind of first order multi-modal logic, where modal operators are indexed by programs. A diamond formula $\langle\pi\rangle\,\phi$ means that there is a terminating execution of the program π after which ϕ holds, a box formula $[\pi]\,\phi$ means that ϕ holds after every terminating execution.

Most of the proof rules available in the KeY system symbolically execute programs in DL formulae. There are also some induction rules to reason about loops and recursion. The "core" of the calculus is however first-order, in the sense that there is no quantification over functions or sets, no lambda abstraction, etc. The prover uses a sequent-style calculus, which is augmented with *meta variables* to allow the delayed choice of quantifier instantiations, similarly to the free variables used in first order tableau calculi.

As program verification cannot be done fully automatically for realistic programs, it was important to make the interactive user interface of the KeY prover intuitive and powerful.

The KeY tool can be downloaded free of charge from the KeY project home page at `http://i12www.ira.uka.de/~key`.

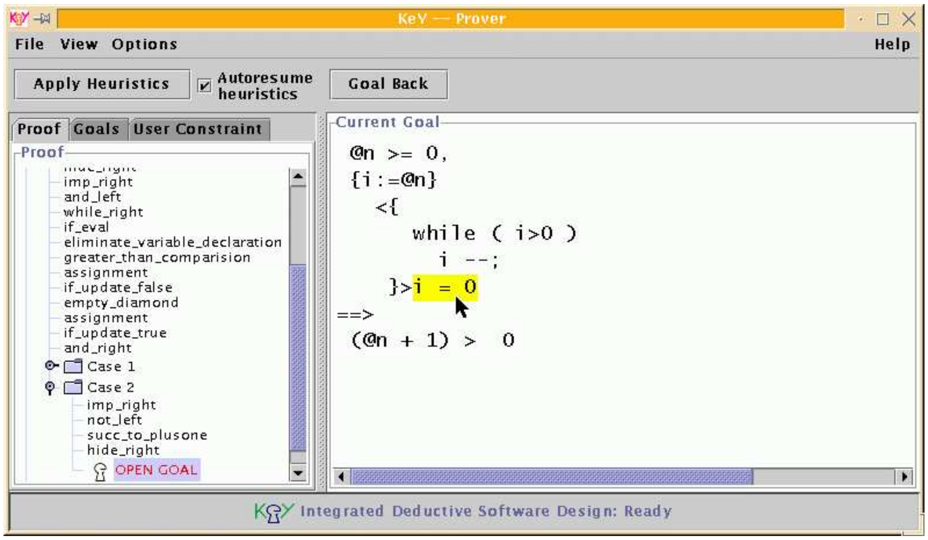

Fig. 1. The main window of the KeY prover

2 The Prover Window

In Fig. 1, the main window of the KeY prover is shown. In the left part of the window, the whole proof tree is displayed, showing the applied rules and the case distinctions, which correspond to splits in the proof tree. Using the "tabs", one can also choose to display only a list of open goals, or the *user constraint*. The user constraint allows the user to control the instantiation of meta variables. The proof steps displayed in the proof view have pop-up menus which allow the user e.g. to cut off parts of the proof at a certain point.

The right part of the window displays the sequent that is currently being worked on. A formatting engine in the style of Oppen's pretty-printer [7] is used to print sequents with a structured layout. As is visible in the figure, the formula, sub-formula, term, etc. that is currently under the mouse pointer is highlighted. Highlighting, in conjunction with layout, helps the user in understanding the structure of a complex formula.

Clicking on an operator in a formula displays a pop-up menu giving a choice of rule applications possible for that sub-formula, see Fig. 2. In the KeY prover, the rules from which proofs are built are combined with the information of how the user should interact with these rules, to form entities called *taclets*, see next section.

We display the proof tree and the current sequent together in one window, using a split pane, for the following reason. It makes perfect sense to work on several proofs at a time. For instance, during construction of a new proof, one might want to consult an older one for reference. It is also conceivable, though not yet implemented, to cut and paste parts of proofs. In such a setting, having

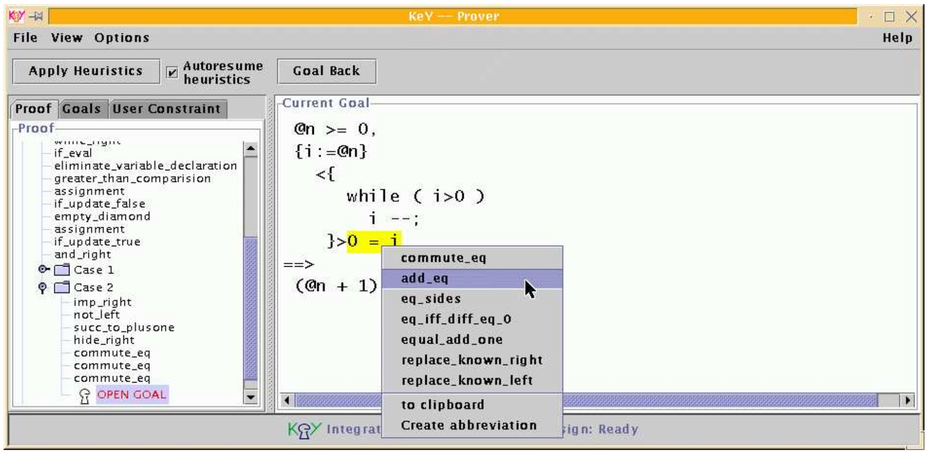

Fig. 2. Choosing a taclet to apply

separate windows for proof trees and sequents would make it hard for the user to see which belongs to which.

3 Taclets

Most existing interactive theorem provers are "tactical theorem provers". The tactics for which these systems are named are programs which act on the proof tree, mostly by many applications of primitive rules, of which there is a small, fixed set. The user constructs the proof by selecting the tactics to run. Writing a new tactic for a certain purpose, e.g. to support a new data type theory, requires expert knowledge of the prover.

In the KeY prover, both tactics and primitive rules are replaced by the *taclet* concept.[1] A taclet combines the logical content of a sequent rule with pragmatic information that indicates how and when it should be used. In contrast to the usual fixed set of primitive rules, taclets can easily be added to the system. They are formulated as simple pattern matching and replacement schemas. For instance, a very simple taclet might read as follows:

find (b −> c ==>) if (b ==>) replacewith(c ==>) heuristics(simplify)

This means that an implication b −> c on the left side of a sequent may be replaced by c, if the formula b also appears on the left side of that sequent.

Apart from this "logical" content, the keyword find indicates that the taclet will be attached to the implication and not to the formula b for interactive

[1] Taclets have been introduced under the name of *schematic theory specific rules (STSR)* by Habermalz [6]. The concept of interactive theorem proving through direct manipulation of formulae was inspired by the theorem prover InterACT [4].

selection, i.e. it will appear in the pop-up menu when the implication is clicked on.

Taclets can be part of *heuristics*. The clause heuristics(simplify) indicates that this rule should be part of the heuristic named simplify, meaning that it should be applied automatically whenever possible if that heuristic is activated. The user can interactively change which heuristics should be active at a certain time.

While taclets can be more complex than the typically minimalistic primitive rules of tactical theorem provers, they do not constitute a tactical programming language. There are no conditional statements, no procedure calls and no loop constructs. This makes taclets easier to understand and easier to formulate than tactics. In conjunction with an appropriate mechanism for heuristic application, they are nevertheless powerful enough to permit comfortable interactive theorem proving [6]. For the automated execution of heuristics, the idea is that any possible taclet application will eventually be executed (fairness), but certain taclets may preferred by attaching priorities to them.

Also note that taclets are rather lightweight entities. It is for instance absolutely possible to introduce dozens of *ad-hoc* taclets to reason about some specific data type in an intuitive way. The set of taclets should and can be designed in such a way that usual human reasoning about some application domain is reflected by the available taclets. An important consequence of attaching taclets to operators is that the taclets for a certain data type will almost all be attached to operators of the according type. For instance, taclets for reasoning about numbers are attached to operators like + or >=, etc. This means that when the user clicks on a specific operator, only those taclets will be visible that are relevant for that operator in that context. This significantly reduces the burden on the user that is usually associated with a large set of rules.

In principle, nothing prevents the formulation of a taclet that represents an unsound proof step. It is possible however, to generate a first-order proof obligation from a taclet, at least for taclets not involving DL. If that formula can be proved using a restricted set of "primitive" taclets, then the new taclet is guaranteed to be a correct derived rule.

No provision is currently made in the user interface for the *construction* of taclets. They are given in the textual form shown above and read into the system by a parser. In future versions, a possibility to define taclets within the user interface might be added to the system.

4 Implementation

The KeY prover is implemented in the Java programming language [5], using the Swing [9] GUI library. The coordination between the displayed proof tree, the current sequent, etc. and the underlying logical data structures follows the *Model, View, Controller* architecture, making intensive use of the *Listener* design pattern (see [3]). While this is not the fastest conceivable technique, it has helped to provide a good modularization of the system.

Highlighting and generation of position-dependent pop-up menus depends on having a fast mechanism to find the term position corresponding to a certain character in the displayed sequent. This is achieved using *position tables*, which record the start and end of nested formulae and terms in every sub-formula/term of the sequent. Position tables are built by the pretty-printer during layout, at a low additional cost, and they are very efficient. There is no perceivable delay due to highlighting when the mouse is moved over the sequent.

For a pleasant user experience, it is also important that the available taclets at a certain position are displayed with minimal delay when the user clicks somewhere. This is achieved using a number of indexing data structures. For every open goal, a *taclet application index* is kept, that stores all taclet applications possible in a sequent at any position. It is organized in such a way that quick access to the applicable taclets is possible based on the position in sequent. Only taclet applications that are actually possible are stored. Regard for instance the taclet given in the previous section, which requires the presence of b in the antecedent. If that formula is not present, a corresponding taclet application will not be put into the taclet application index, and thus will not be displayed to the user. In the current implementation, the taclet application index is simply recalculated before each interaction, but it would be possible to cache most taclet applications between taclet applications, as most of the sequents remain unchanged.

In order to calculate the taclet application index efficiently, a *taclet index* is used. This contains the set of all available taclets, and provides an operation to determine a set of candidates that might be applicable, given some formula and its position in a sequent. The idea is to go through all sub-formulae of a newly introduced formula in a sequent and ask the taclet index for a (hopefully small) set of potentially applicable taclets. It is then checked whether all conditions of the taclet actually are satisfied, and if so, a corresponding taclet application is put into the taclet application index. What indexing mechanism is sensible for the taclet index is of course dependent on the set of taclets in use. Many of the taclets currently used in the KeY prover serve the symbolic execution of programs. We use a hash table indexed by the top operator, and in case of program modalities, by the type of the first executable statement in the program in question. This gives very acceptable performance for interactive use: the time required to apply a rule, to build the new taclet application index and to layout and display the new sequent lies mostly below half a second. The standard set of taclets usually worked with comprises several hundred taclets for propositional and predicate logic, integers, sets and above all for Java Card. When taclets are applied automatically by the "heuristics", performance ranges between 10 rule applications per second for the more complicated symbolic execution taclets to several hundred per second for simple propositional logic.

The performance might become unacceptable in the future, due for instance to an enlarged taclet base. In that case, our course will be to progressively optimize the indexing data structures. In fact, this has already been done twice in the past: originally there was no taclet index at all. As the number of predicate logic rules grew, hashing on the top function symbol was introduced. Finally, with the addition of DL rules, indexing on program statements became necessary.

Another conceivable future optimization is to compile taclets: As taclets have a quite operational semantics, it would be possible to produce Java byte code for the actions of a taclet, instead of the current interpretive approach. It is not clear weather this will become necessary, as the system performs quite satisfactorily so far.

5 Conclusion

We have briefly described the KeY prover from a user interface perspective. In particular, we have introduced the concept of *taclets*, which consist of the logical content of a sequent rule, paired with pragmatic information on how and when to apply it. We have also given a short overview of some of the non-trivial implementation issues involved.

Acknowledgments

The author is indebted to Richard Bubel for providing some of the technical details and to Wolfgang Ahrendt for helpful comments on a draft version of this paper.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, February 2003.
2. Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In Isabelle Attali and Thomas P. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.
4. R. Geisler, M. Klar, and F. Cornelius. *InterACT*: An interactive theorem prover for algebraic specifications. In *Proc. AMAST'96, 5th International Conference on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 563–566. Springer, July 1996.
5. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1997.
6. Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000. `http://i12www.ira.uka.de/~key/doc/2000/stsr.ps.gz`.
7. Derek C. Oppen. Pretty-printing. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

8. Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997.
`ftp://ftp.javasoft.com/docs/javacard/JC20-Language.pdf`.
9. Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison Wesley, 1999.