Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 103 (2004) 67–79
www.elsevier.com/locate/entcs

Taclets and the KeY Prover

Martin Giese

Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden
e-mail: giese@ira.uka.de

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 are the basic building blocks for proofs in the KeY prover.

Keywords: interactive theorem proving, user interface, taclets

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 [16] 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 for JAVA CARD, known as JAVA CARD DL [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. Proofs are constructed in a sequent-style calculus.

Most of the proof rules available in the KeY system symbolically execute programs in DL formulae. For instance, the rule to cope with the if statement

doi:10.1016/j.entcs.2004.09.014

is essentially: [1]

$$\frac{e, \Gamma \vdash \langle a;\ c\rangle\, \phi, \Delta \qquad \neg e, \Gamma \vdash \langle b;\ c\rangle\, \phi, \Delta}{\Gamma \vdash \langle \mathsf{if}\ e\ \mathsf{then}\ a\ \mathsf{else}\ b;\ c\rangle\, \phi, \Delta}$$

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 augments the standard calculus for first order logic augmented with *meta variables* which allow the delayed choice of quantifier instantiations, similarly to the free variables used in first order tableau calculi. Meta variables are place-holders for ground terms which can be introduced instead of quantifier instantiations, and which are later instantiated using unification. Together with a suitable presentation mechanism, this permits to conduct efficient automated proof search and interactive proof construction using one common calculus, see [7].

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. In this paper, we describe the KeY prover from a user interface perspective, without going to deeply into technical details. We will however present the idea of taclets, which are at the basis of interactive and automated proof construction in the KeY prover.

The KeY tool can be downloaded free of charge from the KeY project home page at http://www.key-project.org/

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 as a tree structure, showing the applied rules and the case distinctions, which correspond to splits in the proof tree. Note that the labels displayed are sometimes the names of rules (or rather taclets, as will be explained later), like int_induction or hide_right. But in many cases, more useful information is printed, for instance int i; refers to the symbolic execution of a variable declaration statement, and {} to the removal of an empty block. Also, branching rules can assign useful names to the different branches, as can be seen here for an induction rule. [2]

Clicking on the proof steps displayed in the proof tree view brings up a

[1] The real rule is more involved, due to complications of the JAVA language like side effects in e, abrupt termination etc., see [2]

[2] The induction rule allows to prove an arbitrary statement inductively on the 'Base Case' and 'Step Case' branches. The proved statement can then be used on the 'Use Case' branch to prove the original goal.

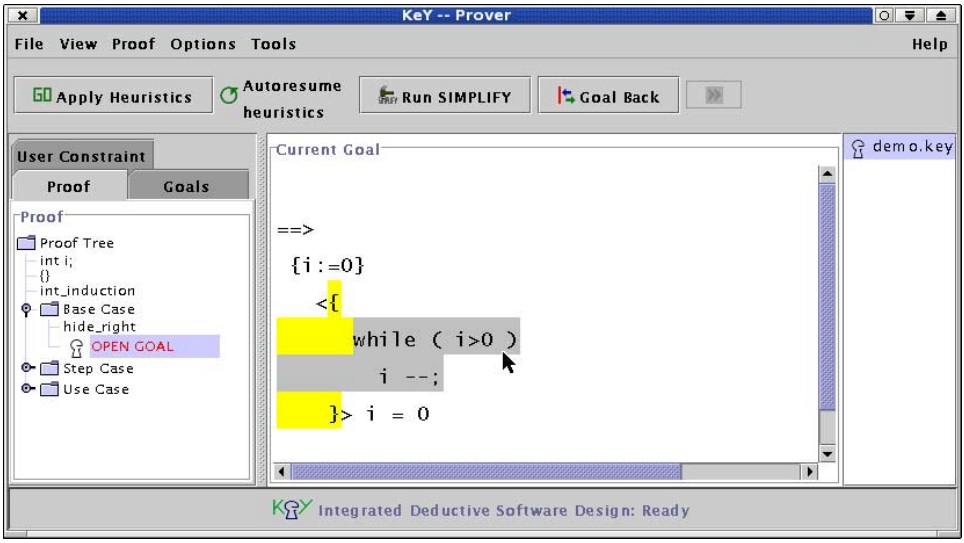

Fig. 1. The main window of the KeY prover

pop-up menu which allows the user e.g. to cut off parts of the proof at a certain point. It is also possible to fold and unfold parts of the proof tree, a very valuable feature for larger proofs. Using the "tabs" at the top of the left pane, one can also choose to display only a list of open goals, or the *user constraint*. The user constraint mainly contains the quantifier instantiations that were chosen by the user in the course of the proof, but it allows to correct faulty instantiations in existing proof without rolling back the whole proof, see [7].

The right pane contains a list of opened proofs. In this case, only one problem file `demo.key` is opened, but in general there might be a number of proof obligations from a program verification task here. It seemed preferable to allow switching between proofs this way to opening one window for each of a possibly large number of proofs.

The central part of the window displays the sequent that is currently being worked on. A formatting engine in the style of Oppen's pretty-printer [12] is used to print sequents with a structured layout. As 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. Note that we have chosen to restrict ourselves to the ASCII character set in our syntax for formulae, although quantifiers, junctors, etc. in mathematical notation would have easily been possible. It is our experience that although such a feature is welcomed by theoreticians, it rather disturbs the software engineers who are ultimately

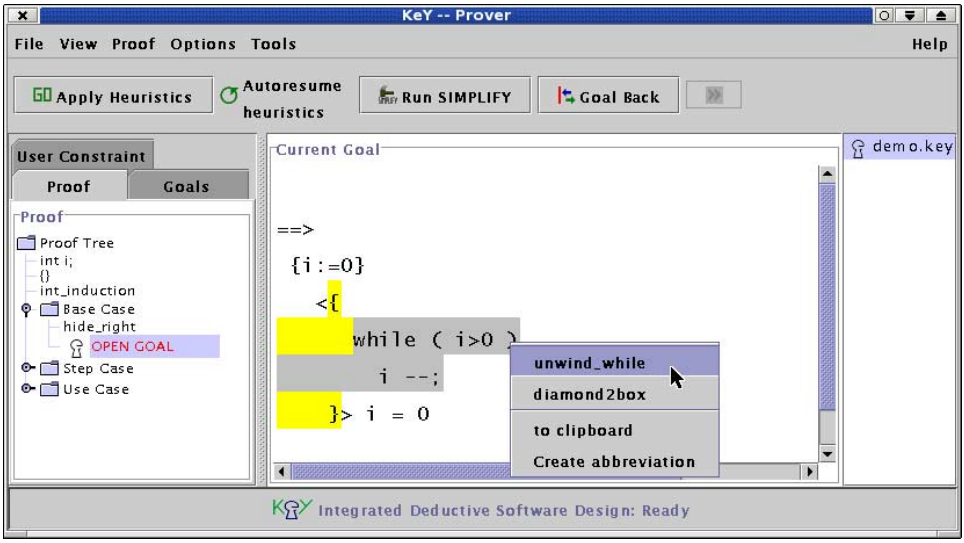

Fig. 2. Choosing a taclet to apply

the intended clients of the KeY system.

The {i:=0} in the displayed formula is an *update*, a special feature of our calculus which denotes that the following formula should be interpreted in a modified state. The formula after the update is a diamond formula containing a JAVA program. The program is surrounded by an extra pair of curly braces, mainly to simplify parsing of JAVA CARD DL formulae.

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*, which are described in the next section.

In this case, there are only two applicable taclets at that particular position in the sequent. One will unwind one execution of the loop. The other will apply duality to transform this formula into a box formula on the right of the sequent.

The set of available rules is designed in such a way as to minimize the amount of keyboard interaction. For instance, propositional reasoning is done simply by selecting an appropriate rule for the formula to be affected. The usual way to instantiate a quantifier is to click *not* on the quantified formula, but on the term for which an instance is required. If one selects the taclets inst_all or inst_ex, one then gets a dialog in which the quantified formula to be instantiated can be chosen among the universal formulae on the left, resp. the existential formulae on the right of the sequent. The same ef-

fect can also be achieved using a *drag and drop* gesture. The user clicks on the term and drops it onto one of the top level quantifiers occurring in the sequent. Only when an instance is needed that is not yet present in the sequent, will the user select the instantiation rule that allows to enter a term directly using the keyboard. A further reduction of the complexity of user interactions stems from the fact that most quantifier instantiations in usual theorem provers are required to obtain instances of axioms and lemmmas. In the KeY prover, these are typically encoded as taclets, see Sec. 3, which can obtain most of the required information from the context in which they are applied.

A number of useful functionalities may be reached via the set of toolbar buttons at the top of the window. The 'Apply Heuristics' and 'Autoresume heuristics' buttons refer to the automated application of taclets explained in the next section. The 'Run SIMPLIFY' button extracts the arithmetic formulae from the goal and tries to close it using the external theorem prover Simplify [11]. 'Goal Back' removes the last proof step on the current branch. Finally, the fast-forward symbol is used to control the proof reuse mechanism that is currently being implemented, and that will allow to reuse earlier proof attempts to show properties after the program has changed, e.g. due to fixing a bug.

3 Taclets

As this paper is focussed on user interface aspects, we can give only a brief introduction to the concept of taclets. For an in-depth, technical discussion of taclets, that also takes account of the particular difficulties associated with a calculus for JAVA CARD DL, see [3].

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.[3] 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

[3] Taclets have been introduced under the name of *schematic theory specific rules (STSR)* by Habermalz [10], see Sec. 5

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

The clause heuristics(simplify) indicates that this rule should be part of the *heuristic* named simplify. A heuristic is simply a named set of taclets. The user can interactively change which heuristics should be active at a certain time. Pushing the 'Apply Heuristics' button applies all taclets that belong to some activated heuristic as long as possible. If the 'Autoresume heuristics' button is checked, heuristics are automatically applied after each interactive taclet application. It is sometimes convenient to switch this behaviour off temporarily, to apply several interactive steps in sequence.

As further examples of taclets, here are the quantifier instantiation taclets mentioned earlier:

inst_all { if (all u.b ==>) find (t) add ({u t}(b) ==>)}

inst_ex { if (==> ex u.b) find (t) add (==> {u t}(b))}

These rules are presented when the user clicks on some term t if there is a universal resp. existential quantifier on the left, resp. right of the sequent. If there are several, the user can pick which one to instantiate. The syntax {u t}(b) denotes the result of substituting t for all occurrences of u in b. [4]

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 nonetheless powerful enough to permit comfortable interactive theorem proving [10]. For the automated execution of heuristics, the idea is that any possible taclet application will eventually be executed (fairness), but certain taclets may be preferred by attaching priorities to them.

[4] The actual rules are slightly more complicated due to difficulties with non-rigid terms in first-order modal logics.

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. For instance, the cancellation law for addition which states that $x + z = y + z$ implies $x = y$ can be coded in a taclet

$$\mathsf{find}(\mathsf{x} + \mathsf{z} = \mathsf{y} + \mathsf{z})\ \mathsf{replacewith}(\mathsf{x} = \mathsf{y})$$

This taclet would be attached only to equalities where there actually are identical summands on both sides.

In principle, nothing prevents the formulation of a taclet that represents an unsound proof step. It is possible however, to automatically generate a first-order proof obligation from a taclet, representing its logical content. 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. For instance, the proof obligation for the cancellation law taclet above would simply be:

$$a + c = b + c \rightarrow a = b$$

for some new constants a, b, c. For the inst_all taclet, one gets:

$$(\forall x.p(x)) \rightarrow p(c)$$

where p is a new function symbol and c is a new constant. In this respect, taclets are different from many systems based on higher-order logic, where the justification of a derived rule is done in some meta-logic. Proof obligations for taclets are in the same logic as the one the taclets act upon. See [3] for a detailed description of how proof obligations are computed.

No provision is currently made in the user interface for the interactive *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 (see [8]), using the Swing GUI library (see [17]). 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 *Observer* design pattern (see [5]). Every change in the data structures representing the proof tree triggers an event for which the concerned user interface components wait. While this is not the fastest conceivable technique, it has helped to provide a good modularization of the system.

4.1 Highlighting

To assist the user in selecting the formula or term a taclet should act upon, the KeY prover highlights the whole sub-formula or term the mouse pointer is over as it moves over the sequent. For instance, in the formula

```
p & (q | r & s)
```

,

given that conjunction (`&`) has priority over disjunction (`|`), the right conjunct `(q | r & s)` is going to be highlighted when the pointer is over the `|` or one of the parentheses, `r & s` will be highlighted when the pointer is over the right `&`, and the whole formula if it is over the left `&`. If the pointer is over one of the symbols `p`, `q`, `r`, or `s`, only that symbol is highlighted. The implementation of this feature relies on 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. The position tables have the same tree structure as the represented terms, so the time to find the position corresponding to a character is linear in the depth of the term. This has so far proved to be fast enough. An additional feature of the position tables is that they store only *offsets* of sub-terms for each position, instead of absolute positions in the string representation of the sequent. This makes it possible to reuse the position table for a formula that is not affected by a taclet application, provided its layout does not change. This optimization has not yet been implemented in the KeY system though. Once the text range to be highlighted has been calculated using the position tables, the actual painting is done using the standard highlighting functionality provided by the JAVA libraries.

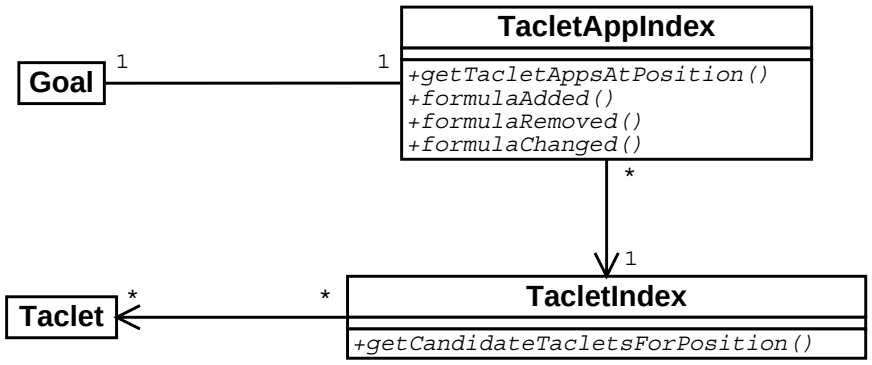

Fig. 3. Indexing Data Structures for Taclets

4.2 The Taclet Application Index

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. The first ingredient for this is of course the position table, which yields a handle on the logic data structures corresponding to the mouse position. The actual list of applicable taclets is computed from this using a number of indexing data structures, see Fig. 3. Considering that the taclet set for JAVA CARD DL comprises hundreds of taclets, it is clearly not an option to iterate through the whole set of taclets while the user waits for the menu. Instead, for every open goal, a *taclet application index* is kept, that stores all taclet applications possible in a sequent at any position. A *taclet application* consists of a taclet along with a position where it is applied and a number of schema variable bindings determined by the position.[5] The taclet application index is organized in such a way that quick access to the applicable taclets is possible based on the position in the sequent. Only taclet applications that are actually possible are stored. Regard for instance the taclet quoted in Sec. 3, which has to be applied on an implication in the antecedent. Only for such positions is a taclet application going to be put in the taclet application index, and only then will it be displayed to the user. The nice thing about the taclet application index is that most of a sequent usually remains unchanged between taclet applications, and accordingly most of the taclet applications remain valid. It is sufficient to remove taclet applications referring to changed formulae and to add some for new formulae after each proof step. This optimization is in the course of being implemented. In the current version of the KeY prover, the taclet application index is simply recalculated before each

[5] There may be unbound schema variables left; the instantiations of those are asked for interactively.

proof step, which has so far been fast enough.

4.3 The Taclet Index

The reason why one can afford to recompute the taclet application index after each proof step is of course that another indexing data structure permits to do this efficiently: The *taclet index*. 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. For each taclet in this set, it is then checked whether all conditions for the application are actually 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. For instance, many of the taclets currently used in the KeY prover serve the symbolic execution of programs. Therefore, we make sure that the indexing can differentiate between taclets for various kinds of Java statements. We use a hash table indexed by the top operator of the formula or term in question, 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 using the heuristics, performance ranges between 20 rule applications per second for the more complicated symbolic execution taclets to about 500 per second for simple propositional logic on a current Linux workstation.

The performance of the taclet index 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. In particular the matching part might become faster than with the current approach of comparing two term data structures.

It is not clear whether this will become necessary, as the system performs quite satisfactorily so far.

5 Related Work

Taclets were first introduced under the name of *schematic theory specific rules (STSR)* by Habermalz [9,10]. The concept of interactive theorem proving by pointing the mouse at the the formula a rule should act upon was strongly inspired by the theorem prover InterACT [6]. That theorem prover allowed reasoning about abstract data types algebraically specified by sets of conditional equations. It provided only a relatively hard-wired set of rules however. Domain-specific reasoning was only possible through the application of conditional equations. The interesting aspect was that the user could click on the term where an equation should be applied. With taclets, it becomes possible to do domain-specific reasoning in a way that matches human reasoning in the domain and not the underlying specification language.

An idea for using mouse gestures to control a theorem prover, known as "Proof by Pointing" has already been suggested earlier by Bertot, Kahn, and Théry [4]. The peculiarity of that approach is that a single mouse click on some sub-formula can trigger a whole series of rule applications that decompose a formula until the selected sub-formula is on the top level of the sequent. Proof by pointing is limited to a fixed sequent calculus, with no domain-specific rules at all.

Semantically, taclets bear an obvious resemblance to tactics and/or derived rules in systems based on higher-order logic like Isabelle [14] or PVS [13], but also to concepts from the proof planning world like the methods of the ΩMEGA system [15]. Indeed, in a taclet-based theorem prover, taclets often play the role of derived rules or tactics, in that they can a complex deduction as a single entity. They also encode knowledge about domain-specific reasoning like methods. Taclets differ from the named concepts in that they

- include an operational semantics for both automated and interactive application, and
- do *not* provide any programming constructs, and thus
- can be justified with respect to other taclets by reasoning in the object logic, and not in some meta-logic.

The taclet mechanism was carefully designed to display all these properties.

6 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.

The KeY system has repeatedly been used in undergraduate education, and students have been able to show simple properties of JAVA programs using the prover. We take this to be an indication that our user interface is good enough to allow it to be used by non-experts after a reasonable amount of coaching. Some larger case studies conducted by more experienced members of the KeY group are presented in [1].

Acknowledgement

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

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. *Software and System Modeling*, 2004. To appear.

[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] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen Schlager. Taclets: a new paradigm for writing theorem provers. *Revista De La Real Academia De Ciencias Exactas, Fisicas Y Naturales*, 2004. to appear.

[4] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *Proc. Intl. Symp. on Theoretical Aspects of Computer Software*, LNCS 789, pages 141–160. Springer, 1994.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] 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.

[7] Martin Giese. Integriertes automatisches und interaktives Beweisen: Die Kalkülebene. Diploma Thesis, Fakultät für Informatik, Universität Karlsruhe, June 1998.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1997.

[9] Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln*. PhD thesis, Universität Karlsruhe, 2000.

[10] 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.

[11] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also published as Xerox PARC Research Report CSL-81-10.

[12] Derek C. Oppen. Pretty-printing. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[13] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *LNCS*, pages 411–414. Springer, July/August 1996.

[14] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[15] J. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.P. Wirth, and J. Zimmer. Proof development with ΩMEGA. In A. Voronkov, editor, *Proceedings of the 18th Conference on Automated Deduction (CADE–18)*, volume 2392 of *LNAI*, pages 144 –149, Copenhagen, Denmark, 2002. Springer Verlag, Germany.

[16] Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997.

[17] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison Wesley, 1999.