 Electronic Notes in Theoretical Computer Science 199 (2008) 107–128 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic

# Richard Bubel[1](#_bookmark0) Andreas Roth[2](#_bookmark0) Philipp Ru¨mmer[3](#_bookmark0)

*Institut fu¨r Logik, Komplexita¨t und Deduktionssysteme Universita¨t Karlsruhe, Germany*

*Department of Computer Science and Engineering Chalmers University of Technology & G¨oteborg University, Sweden*

Abstract

The interactive theorem prover developed in the KeY project, which implements a sequent calculus for JavaCard Dynamic Logic (JavaCardDL) is based on taclets. Taclets are lightweight tactics with easy to master syntax and semantics. Adding new taclets to the calculus is quite simple, but poses correctness problems. We present an approach how derived (non-axiomatic) taclets for JavaCardDL can be proven sound in JavaCardDL itself. Together with proof management facilities, our concept allows the safe introduction of new derived taclets while preserving the soundness of the calculus.

*Keywords:* taclets, lightweight tactics, dynamic logic, theorem proving

# Introduction

Background

Taclets are a new approach for constructing powerful interactive theorem provers [[4](#_bookmark48)]. First introduced as *Schematic Theory Speciﬁc Rules* [[9](#_bookmark53)], they are an efficient and convenient framework for lightweight tactics. Their most important advantages are the restricted and, thus, easy to master syntax and semantics com- pared to an approach based on meta languages like ML, and their seamless integra- tion with graphical user interfaces of theorem provers which they can be efficiently compiled [[7](#_bookmark49)] into.

Taclets contain three kinds of information, the logical content of the rule to be applied, information about side-conditions on their applicability, and pragmatic information for interactive and automatic use. Due to their easy syntax and intuitive

1 Email: [bubel@ira.uka.de](mailto:bubel@ira.uka.de)

2 Email: [aroth@ira.uka.de](mailto:aroth@ira.uka.de)

3 Email: [philipp@chalmers.se](mailto:philipp@cs.chalmers.se)

1571-0661 © 2008 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

doi:10.1016/j.entcs.2007.11.015

operational semantics, a person with some familiarity in formal methods should be able to write own taclets after a short time of study.

The interactive theorem prover developed in the KeY project [[5](#_bookmark50),[1](#_bookmark45)] is based on taclets implementing a sequent calculus for JavaCard Dynamic Logic (*Java- CardDL*) [[3](#_bookmark47)]. *JavaCard* is a subset of *Java* lacking multi-threading, garbage collec- tion and graphical user interfaces, but with additional features like transactions.

*JavaCardDL* has around three hundred axiomatic rules, this means taclets that capture the *JavaCard* semantics. Correctness of rules is crucial since new taclets can be introduced quite easily. The work presented here ensures the correctness of derived taclets for *JavaCardDL* by providing means to prove them correct relatively to the core set of *JavaCardDL* axioms (possibly enriched with further axioms for certain theories). The soundness of taclets can be proven in the calculus itself by

showing the validity of an appropriately constructed *proof obligation*. This report extends the respective approach for classical first-order logic described in [[9](#_bookmark53)] to *JavaCardDL*.

Related Work

Related to our approach are other projects for program verification like Bali [[16](#_bookmark60),[17](#_bookmark61)], where consistency and correctness of rules that cover the Java se- mantics are ensured using Isabelle, or the LOOP project [[11](#_bookmark55)] where PVS is used as foundation, and the calculus rules are thus obtained as higher order logic theorems. Complementary to the presented approach—ensuring correctness of *derived* taclets—further work has been carried out in the KeY project in order to cross-

validate selected axiomatic rules against the *Java* axiomatisation of Bali [[13](#_bookmark57),[15](#_bookmark59),[18](#_bookmark62)]

covering assignment rules (esp. for arrays) and KeY’s improved while-invariant rule

as introduced in [[6](#_bookmark51)]. Further, [[2](#_bookmark46)] describes the automatic verification of an impor- tant subset of the *JavaCardDL* calculus rules against a Maude rewriting semantics of Java.

Structure of this Paper

In Sect. [1.1](#_bookmark2) we repeat the most important concepts of classical dynamic logic and *JavaCardDL*. A formal description of taclets and a definition of the basic vocabulary used throughout the paper is given in Sect. [2](#_bookmark8). The different steps to be performed

in order to prove the correctness of derived taclets are described in Sect. [3](#_bookmark20)–[5](#_bookmark26). In Sect. [6](#_bookmark32) we give a justification of the complete procedure as main theorem. Finally, in Sect. [7](#_bookmark44) we discuss the current and future work to be done.

* 1. *Dynamic Logic*

Classical Dynamic Logics

The family of dynamic logics (DL) [[10](#_bookmark54)] belongs to the class of multi-modal logics. As programs are first-class citizens in DL formulas, DL is well-suited for program analysis and reasoning purposes. For the sake of simplicity and as a consequence of

Γ ▶ ⟨ if (*ψ*) {p; while (*ψ*) {p} } else {}⟩*φ,* Δ Γ ▶ ⟨while (*ψ*) {p}⟩*φ,* Δ

(1)

Γ*,ψ* ▶ ⟨p⟩*φ,* Δ Γ*,* ¬*ψ* ▶ ⟨q⟩*φ,* Δ Γ ▶ ⟨ if (*ψ*) {p} else {q}⟩*φ,* Δ

(2)

Γ ▶ ⟨p⟩⟨q⟩*φ,* Δ

Γ{x←z}

*.* {x←z}

{x←z}

(3)

*,* x =t

▶ *φ,* Δ

(4)

Γ ▶ ⟨p;q⟩*φ,* Δ Γ ▶ ⟨x=t⟩*φ,* Δ

Table 1

DPDL/DQDL Axiomatisation (excerpt). z is a new variable.

using a non-concurrent and real world programming language, we will only consider deterministic programs.

Let p be an arbitrary program and *φ* a first-order or dynamic logic formula, then

* ⟨p⟩*φ* (“diamond p *φ*”): p terminates and after the execution of p formula *φ* holds
* [p]*φ* (“box p *φ*”): if p terminates then after the execution of p formula *φ* holds

are typical representatives of DL formulas. Deterministic propositional dynamic logic (DPDL) is defined over a signature Σ = (*At*0*, Prg* 0*, Op*), where *At*0*, Prg* 0 are enumerable sets of propositional variables and atomic programs (resp.). Besides the classical propositional operators ¬*,* → the operator set *Op* contains box [p] and diamond ⟨p⟩ modalities for each program p. The set of formulas is the smallest set defined inductively over *At*0 and *Prg* 0:

* all classical propositional formulas are formulas in DPDL
* if *φ, ψ* are DPDL formulas then *φ* → *ψ* and ¬*φ* are DPDL formulas
* if p ∈ *Prg* is a program and *φ* a formula in PDL then ⟨p⟩*φ* and [p]*φ* are DPDL formulas
* the set *Prg* of programs is the smallest set satisfying
  + 1. *Prg* 0 ⊆ *Prg*
    2. if p*,* q ∈ *Prg* and *ψ* ∈ DPDL then

‘p;q’ (concatenation), ‘ if (*ψ*) {p} else {q}’ and ‘while (*ψ*) {p}’ are programs.

The semantics can be defined in terms of Kripke frames (*S,* (*ρ*p)p∈*Prg* ) with a set *S* of states, and transition relations *ρ*p : *S* → *S* which define the semantics of each program p ∈ *Prg* . The relations *ρ*p have to adhere to certain conditions w.r.t. the program constructors (;, if −else, etc.) from the definition above, for example, program composition *ρ*p;q = *ρ*q ◦ *ρ*p.

An excerpt from an axiom system for DPDL in terms of sequent calculus rules is given in Table [1](#_bookmark4).

DPDL is useful to reason about program properties induced by the program

constructors. However, as a consequence of constructing programs from atomic programs without any fixed semantics, they lack possibilities to talk about individ- ual programs and, thus, about functional properties.

Like the step from propositional to first-order logic, one extends DPDL to deter- ministic quantified dynamic logics (DQDL). DQDL extends the propositional part to full first-order logic (with equality and a universe *D*), and on the program side it replaces the atomic programs with assignments of the form v=t, where v is a vari- able and t an arbitrary term. In general, each program state *s* ∈ *S* is assigned a first order structure (*D, I*) and a variable valuation *σ* : *Var* → *D* respecting *ρx*=*t*(*s*)= *s*' with *σ*' = *σt*(*D,I*)*,σ* .

*x*

Again a relatively [4](#_bookmark6) complete calculus can be given, the corresponding assign- ment rule is shown in Table [1](#_bookmark4).

Example 1.1 *For the universe D* = N

*. .* ∗

*.* (*close*)

*of natural numbers, the DQDL formula*

*.*

⟨x=3;y=x;⟩y =x *can be proven valid with the*

*rules of Table* [*1*](#_bookmark4) *as shown on the right.*

x = 3*,* y = *x* ▶ y =x

*~~. .~~*

([4](#_bookmark3))

x =3 ▶ ⟨y=x;⟩y =x

*~~.~~* ([4](#_bookmark3))

▶ ⟨x=3;⟩⟨y=x;⟩y =x

*~~.~~* ([3](#_bookmark3))

▶ ⟨x=3;y=x;⟩y =x

*JavaCardDL*

The step from academic languages as described in the previous paragraphs to real world programming languages like *JavaCard* [[8](#_bookmark52),[14](#_bookmark58)] leads to several complications. In the next few paragraphs, we introduce some features of *JavaCardDL* [[3](#_bookmark47)]. First some preliminaries:

* Formulas must not occur in *JavaCardDL* programs, instead *Java* expressions of type boolean are used as guards.
* The set of variables *Var* = *PVar* *LVar* is the disjoint union of program vari- ables *PVar* and logical variables *LVar*. In contrast to logical variables, program variables can occur in programs as well as in formulas, but cannot be bound by

*.*

quantifiers. For instance, let *x* ∈ *LVar* and o*,* u ∈ *PVar*, then ∀*x.*⟨o=u;⟩*x* =u is a

well-formed *JavaCardDL* formula, whereas ∀*x.*⟨o=*x*;⟩*x* =*.* u is not.

* All states have the same universe *D*, and predicates are assumed to have the same meaning in all states (they are *rigid* ).

A sequent calculus covering *JavaCard* has to cope with aliasing, side-effects, abrupt termination as result of thrown exceptions, breaks, continues or returns and more. The KeY approach follows the symbolic execution paradigm, thus a majority of the calculus rules realises a *JavaCard* interpreter reducing expressions

and statements stepwise to side-effect free assignments.

Example 1.2 *An easy-to-use decomposition rule similar to (*[*3*](#_bookmark3)*) is not available in*

*JavaCardDL due to abrupt termination. For example*

*.*

▶ ⟨l :{ if (v == 0) { break l; } else { v = 0; } v = 3;} ⟩v =3

4 Usually DQDL is interpreted in an arithmetic structure.

*cannot be decomposed to*

*.*

▶ ⟨l :{ if (v == 0) { break l; } else { v = 0; } }⟩⟨v *=* 3⟩v = 3*,*

*as this is obviously not equivalent for* v= 0*.*

Decomposition was essential for DPDL and DQDL in order to reduce the com- plexity of programs stepwise to atomic programs or assignments, which can be handled by calculus rules without the need of a dedicated rule for each program.

*JavaCardDL* therefore introduces the notion of a *ﬁrst active statement* to which

a rule applies, and a program context ‘ .. ◦1 ...’ whose inactive prefix ‘ .. ’ matches

on all preceding labels, opening braces or try blocks. Consider the following rule:

*. .*

#b = true ▶ ⟨ .. {#sta1} ...⟩*φ* #b = false ▶ ⟨ .. {#sta2} ...⟩*φ*

▶ ⟨ .. if (#b) {#sta1} else {#sta2} ...⟩*φ*

(5)

where #b is a side-effect free boolean expression and #sta1, #sta2 are arbitrary

*JavaCard* statements.

Example 1.3 (Example [1.2](#_bookmark5) continued) *Applying rule (*[*5*](#_bookmark7)*) to*

*.*

▶ ⟨l :{ if (v == 0) { break l; } else { v = 0; } v = 3;}⟩v=3

*where* ◦1 *corresponds to the program between ‘* l :{*’ (inactive program preﬁx) and ‘*v = 3;}*’ (suffix of the program context) now yields the two sequents*

*. .*

1. (v==0) = true ▶ ⟨l :{ { break l; } v = 3; }⟩v=3 *and*

*. .*

1. (v==0) = false ▶ ⟨l :{ { v = 0; } v = 3; }⟩v=3

# Taclets

Taclets are lightweight, stand-alone tactics with simple syntax and semantics. Their introduction was motivated by the observation that only few basic actions in proof construction are sufficient to implement most rules for first-order modal logic. These are:

* + to recognise sequents as an axiom, and to close the according proof branch,
  + to modify at most one formula per rule application,
  + to add a finite (and fixed) number of formulas to a sequent,
  + to let a proof goal split in a fixed number of branches,
  + to restrict the applicability according to context information.

These are the only actions which taclet constructs are provided for. This restriction turns out to reduce the complexity for users of a proof system significantly [[4](#_bookmark48)].

Taclets by Example

Taclets describe rule schemas in a concise and readable way. A simple example rewrites terms 1 + 1 with 2. In taclet notation such a rule schema is written as:

find(1 + 1) replacewith(2)

In a taclet—in addition to the logical content of the described rule—an opera- tional meaning is encoded: If a user of a taclet-based prover selects the term of the find-part (i.e. 1+ 1) of a taclet and chooses the taclet for application, the find-part is replaced with (an instantiation of) the replacewith-term (i.e. 2).

In this simple form, the rule schemas described by taclets are not expressive enough for practical use; *schema variables* and more constructs besides find and replacewith make them powerful enough to fulfil the requirements posed above.

Schema Variables and Instantiations

Expressions [5](#_bookmark10) in taclets may contain elements from a set *SV* of *schema variables*. An *instantiation ι*(v) of a schema variable v ∈ *SV* is a concrete expression that must fulfil certain conditions depending on the kind of the schema variable (see below). We may, e.g., define a schema variable i such that *ι*(i) must be a term of an integer sort.

Expressions *e* in taclets containing schema variables from *SV* are called *schematic expressions* over *SV* . The instantiation map *ι* can canonically be ex- tended to schematic expressions:

*ι*(*op*(*e ,... ,e* )) = ⎧⎨ *ι*(*op*) if *n* =0 and *op* ∈ *SV*

(6)

1 *n*

⎩

*op*(*ι*(*e*1)*,... , ι*(*en*)) otherwise

Thus, *e* describes a set of concrete expressions:

{*ι*(*e*) | *ι* is an instantiation map for every v ∈ *SV* }

For instance, a taclet find(i + i) replacewith(2 ∗ i) contains schematic terms over

{i}. Applied on a sequent containing the term 3 + 3, i is instantiated with *ι*(i)= 3 and the taclet replaces *ι*(i + i)=3 + 3 with *ι*(2 ∗ i)= 2 ∗ 3 in the new goal.

Taclet Syntax

The clause replacewith(2) is an example of a *goal template*, this means the description of how a goal changes by applying the taclet. More than one goal template may be part of a taclet, separated by semicolons, which describes that a goal is split by the taclet. If there is no goal template in a taclet, applications close the proof branch. Additionally, goal templates may contain the following clauses:

5 By *expression* we denote syntactic elements like terms or formulas, but in the context of *JavaCardDL*

also *Java* programs.

* + While in the example taclets above the find- and replacewith-parts consisted of terms, they can also be sequents. *All* find- and replacewith-parts of a taclet must *either* be terms or sequents. These sequents indicate that the described expression must be a top-level formula in either the antecedent or succedent, e.g. a taclet find( ▶ *φ* → *ψ*) replacewith(*φ* ▶ *ψ*) (over the schema variables {*φ, ψ*}) is applicable only to top-level formulas in the succedent. A sequent in the find-part must have either an empty antecedent or succedent.
  + Taclet applications can *add* formulas to the antecedent or succedent. This is denoted by the keyword add followed by a schematic sequent (similarly to replacewith).
  + Taclets support the dynamic enlargement of the taclet rule base by adding new taclets using the keyword addrules. We omit this feature in the present paper, although a treatment similar to what is shown here is possible.

Often, more requirements on the sequents that a taclet should be applicable to is needed. Such side conditions are described by the following optional taclet con- stituents:

* + A taclet that contains an if followed by a schematic sequent *context* is only directly applicable if *context* is a “sub-sequent” of the sequent the taclet is applied to. If this is not the case, the taclet is however still applicable but, by an automatic cut, it is required to show the if-condition.
  + Predefined clauses in a varcond-part describe conditions on the instantiations of schema variables. The most important ones are:
    - v not free in s, which disallows logical variables *ι*(v) to occur unbound in

*ι*(s).

* + - v new depending on s, which introduces a new skolem symbol *ι*(v) (possibly depending on free “meta variables” occurring in *ι*(s)).

The complete syntax of taclets is reiterated here as an overview:

[if (*context*)] [find (*f* )] [varcond (*c*1,. .. ,*ck*)] [replacewith (*rw*1)] [add (*add*1)];

(7)

. .

[replacewith (*rwn*)] [add (*addn*)]

For *i* = 1 *... n*, *context* and *addi* stand for a schematic sequent, *f* and *rwi* for a schematic term, formula, or sequent but all of the same kind. *c*1*,... , ck* are variable conditions.

Additionally—though out of scope of this paper—taclets can be assigned to one or several rule sets, which makes them available to be automatically executed by strategies. For a homogenous treatment in this paper *f* and *rwi* are declared to be never empty: we assume that skipping replacewith is a shorthand for *rwi* = *f* , a skipped find means *f* = ▶ *false*, and *false* always occurs in succedents of sequents.

A skipped if- or add-part means *context* = ▶ or *addi* = ▶ (resp.).

Schema Variable Types

While the above definitions have been general enough to be applied to every first- order modal logic, we are now focusing on special schema variables for *JavaCardDL*. Let *SVtac* denote the schema variables contained in a taclet *tac*. Schema variables v ∈ *SVtac* are assigned to one out of a predefined list of types, each having special

properties concerning admissible instantiations *ι*(v). An instrument to define these properties is to introduce *preﬁx sets* (denoted by Π*l*(v), Π*pv*(v), and Π*jmp*(v)) for schema variables v. A selection of the most relevant schema variable types is given below. If v is of type

* Variable, then v is assigned a *sort*, *ι*(v) must be of that sort. Moreover, *ι*(v) must be a logical variable. For v /= v' ∈ *SV* : if v' is a Variable schema variable then *ι*(v) /= *ι*(v'). *ι*(v) must not occur bound in *ι*(v'') for all v'' ∈ *SV* .
* Term, then v is assigned a *sort*, *ι*(v) must be of that sort.

v is assigned a set Π*l*(v) ⊆ *SV* of schema variables. Π*l*(v) is defined to be the smallest set with, for all constituents of *tac*, if v occurs in the scope of a Variable schema variable v' ∈ *SV* then v' ∈ Π*l*(v) except there is a variable condition v' not free in v declared in *t*. v is assigned a set Π*pv*(v) which is the smallest set of program variables that occur but are not declared in *tac* or are declared above [6](#_bookmark12) every occurrence of v.

We require from instantiations *ι*: If, for some v' ∈ *SVtac*, *ι*(v') is a logical variable that occurs unbound in *ι*(v) then v' ∈ Π*l*(v); if *ι*(v') is a program variable that occurs undeclared in *ι*(v) then v' ∈ Π*pv*(v).

* Formula, then as for Term, v is assigned Π*l*(v) and Π*pv*(v). v must fulfil the same conditions concerning these sets.
* Statement, then *ι*(v) is a *JavaCard* statement. Again, v is assigned Π*pv*(v) and it must satisfy the same conditions as above concerning this set.

v is assigned a set Π*jmp*(v) consisting of *JavaCard* statements break, continue,

break *l*, continue *l* for all labels *l*, if v is enclosed with a suitable jump target. If *jst* is a break or continue statement of *ι*(v) with a target not in *ι*(v) then *jst* ∈ Π*jmp*(v). [7](#_bookmark13) Usually Statement schema variables have names starting with #

to distinguish them from regular *Java* elements.

* ProgramVariable, then *ι*(v) is a local program variable or class attribute of *Java*. v is assigned a *Java* type and *ι*(v) must be of that type. Again, names of this kind of variable start with #.
* ProgramContext, then *ι*(v) is a program transformation [8](#_bookmark14) *pt* that takes a *Java* program element *α* and delivers a new program element *pt*(*α*), such that *pt*(*α*) is a sequence of statements of which the first one contains *α* and has only opening

6 If we consider *tac* as abstract syntax tree.

7 For a complete treatment of *JavaCardDL* it is furthermore necessary to consider return-statements, which are left out in this paper

8 Thus being an exception from the statement above that *ι*(v) must be an expression.

braces, opening try blocks, etc., in front. For this case, the continuation of the instantiation map ([6](#_bookmark9)) is then modified to

*ι*(*op*(*e*1*,... , en*)) := *pt*(*ι*(*e*1)) if *n* =1 and *op* is a ProgramContext schema variable.

Usually, v is denoted by .. *e*1 ... containing the schematic *Java* program *e*1,

as introduced in Sect. [1.1](#_bookmark2).

Example 2.1 *The following taclet performs a cut with the condition that the fo- cused term (*t*) equals* 0 *and replaces it in the respective goal by* 0*. We declare* t *as* Term *schema variable of an integer sort.*

find(t) replacewith(0)

*.*

add(t =0 ▶ );

*.*

(8)

replacewith(t) add( ▶ t = 0)

*As an example that represents a rule of JavaCardDL, we take a taclet that re- places the postﬁx increment operator applied to a program variable (*x*)* behind *a statement (*#sta*) with an equivalent statement using assignment and the* + *opera- tor, and leaves the formula (φ) behind the diamond unchanged.* #sta *is a* Statement *schema variable and φ a* Formula *schema variable.*

find(⟨#sta x++;⟩*φ*) replacewith(⟨#sta x=x+1;⟩*φ*) (9)

*Finally, the following taclet splits a proof for an* if *statement with the condition* x==0 *(where* x *is a concrete local variable) and produces goals, reducing the formula to the statements of the appropriate branch and the if condition put to the correct side of the sequent.* #sta1 *and* #sta2 *are* Statement *schema variables and φ is a* Formula *schema variable.*

find(⟨l: if (x==0) #sta1 else #sta2⟩*φ*)

replacewith(⟨l: #sta1 ⟩*φ*)

replacewith(⟨l: #sta2 ⟩*φ*)

*.*

add(x =0 ▶ );

*.*

add( ▶ x = 0)

(10)

Semantics

Taclets have a precise operational semantics, which is described in detail in [[4](#_bookmark48)], and which we have sketched informally above. For the purposes of this paper it is sufficient to fix the logical meaning of a taclet in the traditional rule schema notation.

We denote the union of two sequents and the subset relation between two se- quents as follows:

Γ1 ▶Δ1 ∪ Γ2 ▶Δ2 := Γ1*,* Γ2 ▶Δ1*,* Δ2

Γ1 ▶Δ1 ⊆ Γ2 ▶Δ2 iff Γ1 ⊆ Γ2 and Δ1 ⊆ Δ2

First, we assume that *f* is a schematic sequent, i.e. the taclet *tac* can only be applied to top-level formulas. By the operational semantics of taclets [[4](#_bookmark48)], *tac* represents the rule schema:

*rw*1 ∪ *add*1 ∪ (Γ ▶Δ) *... rwk* ∪ *addk* ∪ (Γ ▶Δ)

*f* ∪ (Γ ▶Δ) (11)

where Γ ▶Δ is an arbitrary sequent with *context* ⊆ *f* ∪ (Γ ▶Δ).

Similarly, if *f* is a schematic term or formula (*seq*[*e*] denotes a sequent with an arbitrary but for a rule fixed occurrence of an expression *e*):

*seq*[*rw*1] ∪ *add*1 ∪ (Γ ▶Δ) *... seq*[*rwk*] ∪ *addk* ∪ (Γ ▶Δ)

*seq*[*f* ] ∪ (Γ ▶Δ)

where Γ ▶Δ is an arbitrary sequent with *context* ⊆ *seq*[*f* ] ∪ (Γ ▶Δ).

In Sect. [4](#_bookmark21), the notion of *meaning formulas* is derived that makes the meaning of these rule schemas induced by taclets more precise.

Due to their simplicity and operational meaning, taclets can be schematically compiled into the GUI of taclet-based interactive theorem provers: In the KeY system a mouse click over an expression displays only those taclets whose find-part can be matched with the expression in focus. This drastically reduces the cognitive burden on the user. For an extensive account on user interaction see [[4](#_bookmark48)].

# Outline of Bootstrapping Taclets

After having introduced basic notions and notations, we can focus on the task of how to ensure correctness of derived taclets. We aim to prove their soundness within the *JavaCardDL* calculus itself. Our approach is based on [[9](#_bookmark53)] which has already

provided this kind of bootstrapping for classical first-order logic.

Given a taclet *tac*, we first derivea *meaning formula M* (*tac*) (see Sect. [4](#_bookmark21)), which is supposed to be valid if and only if all possible applications of *tac* are correct proof steps. For example, consider the following taclet *tac*0:

find(*true* ∧ *φ* ▶ ) replacewith(*φ* ▶)

with a Formula schema variable *φ*. The corresponding meaning formula is

*M* (*tac*0)= ¬*φ* → ¬(*true* ∧ *φ*) or equivalently (*true* ∧ *φ*) → *φ*

Intuitively, the meaning formula states that if a formula in an antecedent is replaced, the new formula must be at most as strong as the old one. If this can be proven for all instantiations of *φ*, i.e. for all formulas, then obviously *tac*0 is sound. Unfortunately, meaning formulas contain schema variables (here: *φ*) and are

thus no *JavaCardDL* formulas. Moreover, we have to quantify somehow over all

formulas. Skolemisation of schema variables (see Sect. [5](#_bookmark26)) helps us, however, not

having to leave our original logic and not having to employ higher order logics on the object level. Skolemisation of meaning formula *M* (*tac*0) leads to

*M*Sk(*tac*0)= (*true* ∧ *φ*Sk) → *φ*Sk*,*

where *φ*Sk is a new nullary predicate. We call these formulas *M*Sk(*tac*) *taclet proof obligations*. *M*Sk(*tac*) is a *JavaCardDL* formula (with a slightly extended vocabu- lary) and can be loaded into our interactive theorem prover. If the proof obligation

can be proven successfully then correctness of the taclet is ensured for all possible applications according to the definition of the meaning formula. The proof of the corresponding theorem is given in [[12](#_bookmark56)] and sketched in Sect. [6](#_bookmark32).

On a semantic level, this theorem can be justified by arguing that if an ap- plication of the taclet *tac* leads to an incorrect proof, a suitable interpretation *D* can be constructed such that the meaning formula *M* (*tac*) is not satisfied under *D* (which is a direct consequence of the definition of meaning formulas) and thus *M* (*tac*) could not have been proven. This semantic argumentation works fine for

first-order logics [[9](#_bookmark53)], but when *JavaCardDL* comes into play, the complete complex

*JavaCard* semantics would have to be incorporated in the reasoning.

Instead, we take a syntactic approach getting the *JavaCard* semantics via the *JavaCardDL* calculus for free. The basic idea is to show that an application of a taclet *tac* can always be replaced by a transformed proof of *M*Sk(*tac*).

# Meaning Formulas of Taclets

The basis for our reasoning about the correctness of taclets is a *meaning formula* [[9](#_bookmark53)] derived in this section. It is declared to be the meaning of a taclet independently from concrete taclet application mechanisms, thus providing a very flexible way to address soundness issues. In fact we define a taclet application mechanism to be correct if (and only if) taclets with valid meaning formulas are translated into sound rules. [9](#_bookmark23) To show that a taclet is correct it is thus sufficient to prove the validity of its meaning formula.

For the whole section we define (Γ ▶Δ)∗ := Γ → Δ, in particular ( ▶ *φ*)∗ = *φ* and (*φ* ▶)∗ = ¬*φ*. Furthermore, in this section by the validity of a sequent we mean the validity of (Γ ▶Δ)∗. We define a (sequent) calculus *C* to be *sound* if only valid sequents are derivable in *C*. We conceive rules

*P*1 *.. . Pn Q*

as relations between tuples of sequents (the premisses) and single sequents (the conclusion) and define that a rule *R* ∈ *C* is sound if for all tuples (⟨*P*1*,... , Pk*⟩*, Q*) ∈ *R*:

if *P*1*,... , Pk* are valid, then *Q* is valid. (12)

9 As a schematic formula, the meaning formula is by definition valid iff all instances of the formula are valid.

For the calculus *C* we can state:

Lemma 4.1 *C is sound if all rules R* ∈ *C are sound.*

The rules *Rtac* we are interested in are defined through taclets *tac* over a set *SV* of schema variables in the form as defined in ([7](#_bookmark11)). Assuming first that the find- part is a sequent, taclets induce the rule schema ([11](#_bookmark19)). To apply Lem. [4.1](#_bookmark24), for each instantiation *ι* of *SV* , ([12](#_bookmark22)) must be shown for *k* = *n*, *Pi* = *ι*(*rwi* ∪ *addi* ∪ Γ ▶ Δ) (*i* = 1 *... n*), and *Q* = *ι*(*f* ∪ Γ ▶ Δ). Since the formulas of Γ ▶ Δ which are not in *context* are arbitrary and not influenced by the rule application we can simply omit them and show the lemma for *Pi* = *ι*(*rwi* ∪ *addi* ∪ *context* ) (*i* = 1 *... n*) and *Q* = *ι*(*f* ∪ *context* ). We assume that *tac* does not introduce skolem functions,

i.e. does not contain such a variable condition. Then by the deduction theorem, the global condition ([12](#_bookmark22)) can be strengthened to the local implication, namely that

*P* ∗ ∧ *...* ∧ *P* ∗ → *Q*∗ must be valid.

1. *n*

Since *ι*, as defined by ([6](#_bookmark9)), treats propositional junctors as a homomorphism

and the operator (·)∗ is a homomorphism regarding the union of sequents up to propositional transformations, this formula can now be simplified as follows:

*n*

* 1. *n*

*i*=1

*P* ∗ ∧ *...* ∧ *P* ∗ → *Q*∗ = *ι*(*rwi* ∪ *addi* ∪ *context* )∗ → *ι*(*f* ∪ *context* )∗ (13)

*n*

= *ι* (*rw*∗ ∨ *add*∗) → (*f* ∗ ∨ *context* ∗) *.* (14)

*i*

*i*

*i*=1

If ([14](#_bookmark24)) is proven for all instantiations *ι*, then the rule *Rtac* represented by *tac* is sound.

In the next definition our previously made assumptions are revoked: the variable condition *svi* new depending on*...* introduces new skolem functions. If *P*1*,... , Pn* contain skolem symbols that do not occur in *Q*, the interpretation of the symbols can be regarded as universally quantified in ([12](#_bookmark22)) by the usual definition of ‘valid’. Because of their negation in ([13](#_bookmark24)), they are existentially bound in the meaning for- mula. Moreover, taclets that have terms or formulas instead of sequents as find-part and replacewith-parts are reduced to a rule that adds an equivalence *f* ↔ *rwi* or equation *f* = *rwi* to the antecedent.

Definition 4.2 (Meaning Formula) *Each taclet tac, as declared in (*[*7*](#_bookmark11)*), is as- signed an* unquantified meaning formula *tac*∗*, which is deﬁned by:*

⎧⎪ *n*

(*rw*∗ ∨ *add*∗) → (*f* ∗ ∨ *context* ∗) *if f is a sequent*

*tac*∗ :=

*i*=1

*i*=1 *i* *i*

⎨ *n . i* ∗ ∗

*f* = *rw*

→ *addi*

→ *context*

*if f is a term*

⎪

⎪⎪⎩ *n*

*i*=1

*i*

(*f* ↔ *rwi*) → *add*∗ → *context* ∗ *if f is a formula*

*Suppose* sv1*,... ,* sv*k* ∈ *SVtac are all schema variables, which tac contains a variable condition* sv*i* new depending on*... for. M* (*tac*) := ∃x1 *...* ∃x*k.φ is deﬁned to be the* meaning formula *of tac where φ is obtained from tac*∗ *by replacing each* sv*i with*

*a new* Variable *schema variable* x*i with the same sort as* sv*i.*

Example 4.3 (Example [2.1](#_bookmark15) continued) *The taclets tac*1*, tac*2*, and tac*3 *deﬁned through (*[*8*](#_bookmark16)*), (*[*9*](#_bookmark17)*), and (*[*10*](#_bookmark18)*), resp., have (after applying some propositional equivalence transformations) the following meaning formulas:*

*. . . .*

*M* (*tac*

)= t =0 ∧ t =0 ∨

t = t ∧ ¬(t = 0)

(15)

1

*M* (*tac*2)= ⟨#sta x++;⟩*φ* ↔ ⟨#sta x=x+1;⟩*φ* (16)

*M* (*tac*3)= (⟨l: if (x==0) #sta1 else #sta2⟩*φ* ↔ ⟨l: #sta1 ⟩*φ*)

(17)

∧ x =0

*.*

∨ (⟨l: if (x==0) #sta1 else #sta2⟩*φ* ↔ ⟨l: #sta2 ⟩*φ*)

*.*

∧ ¬(x = 0)

# Construction of Proof Obligations

Except for trivial taclets, the meaning formula *M* (*tac* ) of a taclet *tac* contains schema variables, which is at least inconvenient for proving *M* (*tac* ). Variables of these types, however, do not occur bound within the formula (resp., when consid- ering validity, they can be regarded as implicitly universally quantified), and hence it is possible to replace them in a suitable way without altering the validity of the meaning formula:

* Schema variables for logical variables or program variables can simply be replaced with new concrete variables. It has to be taken in account, however, that when instantiating a schematic expression it is possible that two different schema vari- ables of type ProgramVariable are instantiated with the same concrete variable (which is not possible for Variable schema variables by the definitions of Sect. [2](#_bookmark8)). By the presence or absence of such collisions, the set of instances of a schematic expression is divided into (finitely many) classes, which all have to be considered to capture the meaning of the schematic expression.
* Schema variables for terms, formulas or *Java* statements can be replaced with suitable “skolem” symbols, which are similar to the atomic programs of DPDL for Statement schema variables. To model the notion of abrupt termination, which does not exist in DPDL, tuples of *Java* jump statements are attached to occurrences of symbols for statements.
* Schema variables for program contexts can be replaced with a surrogate *Java*

block containing atomic programs.

From now on, we only consider the replacement of schema variables for logical variables, terms, formulas and statements, and we also assume that the concerned taclets only contain schema variables of these kinds. Other kinds of schema variables are treated in a similar way in [[12](#_bookmark56)].

* 1. *Skolem symbols*

We define two syntactic domains that consist of symbols for the skolemisation of schema variables:

* + - Symbols that are placeholders for terms and formulas, and which are similar to ordinary function and predicate symbols
    - Symbols that are placeholders for *Java* statements, similar to the atomic programs of DPDL.

As usual, the elements of both domains are assigned signatures that determine syntactically well-formed expressions. Their shape is described in more detail as follows.

Skolem Symbols for Terms and Formulas

The sets of symbols for terms and formulas are denoted with *Func*Sk and *Pred*Sk

(resp.). The signature

*α*(*s*Sk

)= (*S, S*1*,... , Sn, T*1*,... , Tk*) for *s*Sk ∈ *Func*Sk

(*S*1*,... , Sn, T*1*,... , Tk*) for *s*Sk ∈ *Pred*Sk

of a symbol *s*Sk ∈ *Func*Sk ∪ *Pred*Sk consists of

* + - a result sort *S*, if *s*Sk ∈ *Func*Sk,
    - a finite sequence *S*1*,... , Sn* of sorts that determines the number and kinds of term arguments; this sequence corresponds to the signature of ordinary predicate symbols,
    - a finite sequence *T*1*,... , Tk* of *Java* types, which are the component types of a tuple of program variables that *s*Sk is applied to.

Accordingly, the inductive definition of well-formed terms and formulas is extended by:

If *s*Sk ∈ *Func*Sk ∪ *Pred*Sk is a symbol with the signature *α*(*s*Sk) as above, *t*1*,... , tn* are terms of the sorts *S*1*,... , Sn* and pv1*,... ,* pv*k* ∈ *PVar* are program variables of the types *T*1*,... , Tk*, then

*s*Sk(*t*1*,... , tn*; pv1*,... ,* pv*k* ) is a term of sort *S* or a formula (resp.).

Skolem Symbols for Statements

The set of skolem symbols used for statements is denoted with *Statement*Sk. The signature *α*(*st*Sk)= (*T*1*,... , Tk, m*) of a symbol *st*Sk ∈ *Statement*Sk consists of

* + - a finite sequence *T*1*,... , Tk* of *Java* types (analogously to the symbols for terms or formulas),
      * a natural number *m* that gives the size of the *jump table*; this is a tuple of *Java*

statements that are arguments of occurrences of *st*Sk within programs.

The symbols *Statement*Sk extend the definition of well-formed *Java* programs, i.e. the following (informal) rule is added to the *Java* grammar [[8](#_bookmark52)]:

Given *st*Sk ∈ *Statement*Sk of signature *α*(*st*Sk) = (*T*1*,... , Tk, m*), program vari- ables pv1*,... ,* pv*k* of the types *T*1*,... , Tk* and let *jst* 1*,... , jst m* be *Java* statements of the following kinds [10](#_bookmark28)

* + - * + return-statements, with or without an argument (a plain program variable).
        + break- and continue-statements, with or without a label.
        + throw-statements whose argument is a program variable. Then

is a statement.

*st*Sk(pv1*,... ,* pv*k* ; *jst* 1; *...* ; *jst m*)

* 1. *From Meaning Formula to Proof Obligation*

From now on we suppose that a taclet *tac* with meaning formula *M* (*tac* ) is fixed. Let *SVtac* be the set of schema variables that *M* (*tac* ) contains. We define an instan- tiation *ι*Sk over *SVtac* that replaces each schema variable either with a *JavaCardDL*

variable or with an appropriate skolem expression. The definition refers to the

properties of schema variables as introduced in Sect. [2](#_bookmark8):

* + - If x ∈ *SVtac* is of type Variable, then *ι*Sk(x) ∈ *LVar* is a new logical variable that has the same sort as x.
    - If sv ∈ *SVtac* is of type Term, Formula or Statement, then let {pv1*,... ,* pv*k* } = Π*pv*(sv) be the program variables that can occur undeclared in instantiations of sv. Let *T*1*,... , Tk* be the *Java* types of pv1*,... ,* pv*k* .
    - If sv ∈ *SVtac* is of type Term, then

*ι*Sk(sv)= *f*Sk(*v*1*,... , vl*; pv1*,... ,* pv*k* )

is a term, where

* *v*1*,... , vl* with *vi* = *ι*Sk(x*i*) are the instantiations of x1*,... ,* x*l* ∈ *SVtac*, which are distinct Variable schema variables determined by the prefix Π*l*(sv)= {x1*,... ,* x*l*} of sv in *tac*
* and *f*Sk ∈ *Func*Sk denotes a new skolem symbol with signature

*α*(*f*Sk)= (*S, S*1*,... , Sl, T*1*,... , Tk*)

where *S* is the sort of sv and *S*1*,... , Sl* are the sorts of *v*1*,... , vl*.

* + - Analogously, if sv ∈ *SVtac* is a schema variable of type Formula, then

*ι*Sk(sv)= *p*Sk(*v*1*,... , vl*; pv1*,... ,* pv*k*)

10 Which are exactly the reasons that can lead to an abrupt termination of a statement, see [[8](#_bookmark52)].

is a formula containing a new skolem symbol *p*Sk ∈ *Pred*Sk for formulas.

* + - If sv ∈ *SVtac* is a schema variable of type Statement, then two additional (and new) program variables are needed: tsv of *Java* type Throwable, and dsv of *Java* type int (the latter variable is used in Sect. [5.3](#_bookmark29)). Let {*jst* 1*,... , jst m*} = Π*jmp*(sv) be jump statements that can occur uncaught in instantiations of sv.

The instantiation *ι*Sk(sv) of sv is the statement [11](#_bookmark30)

*ι*Sk(sv)= *st*Sk(pv1*,... ,* pv*k,* tsv*,* dsv; *jst* 1; *...* ; *jst m*; throw tsv)

where *st*Sk denotes a new skolem symbol for statements with signature

*α*(*st*Sk)= (*T*1*,... , Tk,m* + 1).

Finally, the *taclet proof obligation* of *tac* is defined to be the formula

*M*Sk(*tac* ) := *ι*Sk(*M* (*tac* ))

Example 5.1 (Example [4.3](#_bookmark25) continued) *The proof obligations listed below can be constructed from the meaning formulas of taclets tac*1*, tac*2 *and tac*3*:*

*M*Sk(*tac*1)= *t*Sk *.* Sk *.*

=0 ∧ *t*

=0 ∨

Sk *.*

Sk ∧ ¬(*t*Sk *.*

(18)

*M*Sk(*tac*2) = (19)

*t*

= *t*

= 0)

⟨*sta*Sk(v, t#sta, d#sta; throw t#sta); v++;⟩*p*Sk(v) ↔

(20)

⟨*sta*Sk(v, t#sta, d#sta; throw t#sta); v=v+1;⟩*p*Sk(v)

*M*Sk(*tac*3) = (21)

(⟨l: if (x==0) *β*1

* 1. Sk

*.*

1. Sk

(⟨l: if (x==0) *β*1

else *β* ⟩*p*

1. Sk

else *β* ⟩*p*

(x) ↔ ⟨l: *β* ⟩*p*

1. Sk

(x) ↔ ⟨l: *β* ⟩*p*

(x)) ∧ x =0 ∨

*.*

(x)) ∧ ¬(x = 0)

(22)

*where we use the abbreviations*

*β*1 = *sta*1Sk(x, t#sta1, d#sta1; break l; throw t#sta1);

*β*2 = *sta*2Sk(x, t#sta2, d#sta2; break l; throw t#sta2);

* 1. *Decomposition Rules*

Calculus rules for *JavaCardDL* programs always modify the leading statements within a program block (see Sect. [1](#_bookmark1)). Unfortunately, the addition of skolem symbols for statements would destroy the (relative) completeness of a set of rules: If a skolem symbol turns up as the first active statement of a program block, no *JavaCardDL*

rule will be applicable to that block.

As we have stated in Sect. [1.1](#_bookmark2) that a “naive” decomposition rule for *JavaCardDL* cannot be posed due to abrupt termination, we define a family of decomposition

11 We always add a throw-statement, as instantiations of sv may always terminate abruptly through an exception regardless of Π*jmp*(sv).

rules specifically for statement skolem symbols. These rules cope with abrupt ter- mination by applying a transformation to the statement *α* = *st*Sk(*.. .*). This trans-

formation splits *α* in two parts *α*1 = *st*'

Sk

(*.. .*) and *α*2, such that the concatenation

*α*1; *α*2 is equivalent to the original statement *α*. Furthermore, the first program fragment *α*1 is constructed in a way that prevents abrupt termination, and thus, the equivalence

⟨.. *st*Sk(*.. .*); *β* ...⟩*φ* ↔ ⟨*st*'

Sk

(*.. .*)⟩⟨.. *α*2; *β* ...⟩*φ* (23)

holds. The remaining statement *α*2 does no longer contain any skolem symbols,

i.e. it is a pure *JavaCard* program, and hence it is possible to handle *α*2 by the application of regular *JavaCardDL* rules.

We assume that for each statement skolem symbol *st*Sk ∈ *Statement*Sk that occurs within *ι*Sk a second new skolem symbol *Dec*(*st*Sk) is introduced, which has the same signature as *st*Sk except for the jump table:

*α*(*st*Sk)= (*T*1*,... , Tk, m*) =⇒ *α*(*Dec*(*st*Sk)) = (*T*1*,... , Tk,* 0)*.*

Following equivalence ([23](#_bookmark31)), two decomposition taclets *D*o

*st*Sk

and *D*▢

Sk

*st*

for dia-

mond and box modalities (resp.) are introduced for each statement skolem sym-

bol *st*Sk that occurs in *ι*Sk. We only give the definition of *D*o

*st*Sk

, as the taclet for

boxes is obtained analogously:

o

*D*

*st*Sk

: { find ( ⟨*.. st*Sk(p1*,... ,* p*k*; #jst1; *...* ; #jst*m*); *...*⟩*φ* )

replacewith ( ⟨*Dec*(*st*Sk)(p1*,... ,* p*k*);⟩⟨*.. ic ...*⟩*φ* ) }

where p1*,... ,* p*k* are schema variables for program variables, #jst1*,... ,* #jst*m* are variables for statements corresponding to the signature *α*(*st*Sk) and *φ* is a schema variable for formulas. Furthermore the taclet contains an if-cascade *ic*, which is denoted by *α*2 in equivalence ([23](#_bookmark31)):

{ if ( p*k* == 1 ) #jst1

else i f ( p*k* == 2 ) *...*

else i f ( p*k* == *m* ) #jst*m* }

In this statement at most one of the jump statements represented by the schema variables #jst1*,... ,* #jst*m* is selected and executed, depending on the value of the last program variable argument p*k* (note that the type of p*k* is int by the definitions of the last section).

Example 5.2 *An application of the decomposition rule for diamond modalities could look as follows:*

▶ ⟨*st*' (t*,* d)⟩⟨try { if (d == 1) throw t; } catch (Exception e) {...}⟩*φ*

Sk

▶ ⟨try { *st*Sk(t, d; throw t); } catch (Exception e) {...}⟩*φ*

# Main Result

To show that *tac* is derivable, which is by Sect. [4](#_bookmark21) equivalent to the derivability of all instances of *M* (*tac* ), we assume that there is a closed proof *H* of *M*Sk(*tac* ) using the sequent calculus for *JavaCardDL* (extended by the skolem symbols and

the decomposition taclets of Sect. [5](#_bookmark26)). It is possible to transform *H* into a proof *Hφ*

for each instance *φ* of *M* (*tac* ):

Theorem 6.1 (Main Result) *Suppose that a proof H of M*Sk(*tac* ) *exists. Then for each instance φ* = *κ*(*M* (*tac* )) *of the meaning formula M* (*tac* ) *there is a proof Hφ.*

In the following we will sketch a proof of Theorem [6.1](#_bookmark33). Due to lack of space we skip most of the details of the proof; a more detailed account can be found in [[12](#_bookmark56)].

The proof obligation *M*Sk(*tac* ) = *ι*Sk(*M* (*tac* )) differs from other instances *φ* = *κ*(*M* (*tac* )) of the meaning formula in the instantiation of schema variables for terms, formulas and statements: In *M*Sk(*tac* ) such variables are replaced with skolem symbols as introduced in Sect. [5.1](#_bookmark27). [12](#_bookmark35) Hence it is possible to obtain a “proof” *H*' of *φ* by replacing each occurrence of a skolem symbol *s*Sk(*.. .*)= *ι*Sk(*sv*) in *H* with the instantiation *κ*(*sv*) from *φ*. In general, the tree *H*' cannot be expected to be a proof, as it is possible that the replacement of skolem symbols leads to in- valid rule applications. But by a slightly more complex transformation, as sketched below, it is possible to obtain a legal proof:

For the replacement of skolem symbols we define an appropriate kind of substi- tutions: We assume that a mapping *σ* of the skolem symbols

*Sym*Sk := *Func*Sk ∪ *Pred*Sk ∪ *Statement*Sk

to terms, formulas and *Java* statements “with holes” is given. Namely, we al- low that for a symbol *s*Sk with signature *α*(*s*Sk) = ([*S,* ]*S*1*,... , Sn, T*1*,... , Tk*) (or *α*(*s*Sk)= (*T*1*,... , Tk, m*) for statement symbols), the value *σ*(*s*Sk) containsa number

of holes ◦*i* labelled with natural numbers *i* ∈ {1*,... ,n* + *k*} (or *i* ∈ {1*,... ,k* + *m*}, resp.).

Example 6.2 *For a predicate skolem symbol p*Sk ∈ *Pred*Sk*, an example of a substi- tution is given by the following mapping:*

*σ*(*p*Sk)= *r*(◦2*, a*) ∧ *q*(◦1) ∧ ⟨◦2=1;⟩*φ for p*Sk ∈ *Pred*Sk*, α*(*p*Sk)= (*S,* int)*.*

The mapping *σ* is extended to terms, formulas, *Java* programs, sequents, proof trees and taclets as a morphism, and by the replacement of skolem symbols. Holes are replaced with the arguments of occurrences of skolem symbols: [13](#_bookmark36)

*σ*(*s*Sk(*r*1*,... , rl*)) := {◦1*/r*1*,... ,* ◦*l/rl*} *σ*(*s*Sk)

12 Schema variables for logical variables are in both cases simply instantiated with logical variables.

13 Extensive considerations about possible collisions are omitted in this document; see [[12](#_bookmark56)] for details.

Example 6.3 (Example [6.2](#_bookmark34) continued) *The mapping σ is applied in the fol- lowing way to a formula containing the symbol p*Sk*:*

*σ* ∀*x.p*Sk(*x*; i) = ∀*x.* *r*(i*, a*) ∧ *q*(*x*) ∧ ⟨i=1;⟩*σ*(*φ*)

* 1. *Treatment of Taclets*

The most important observation to prove Theorem [6.1](#_bookmark33) is the following lemma:

Lemma 6.4 (Lifting of Taclet Applications) *Suppose that Rtac*' *is a rule schema that is described by a taclet tac*'*, and that tac*' *does not contain skolem symbols (as introduced in Sect.* [*5.1*](#_bookmark27)*). If an instance of Rtac*' *is given by*

*P*1 ·· · *Pn*

*Q*

*and σ is a substitution of skolem symbols, then there is a proof tree with root se- quent σ*(*Q*)*, whose open goals are exactly the sequents σ*(*P*1)*,... , σ*(*Pn*)*.*

Proof. First suppose that the considered rule application is not the application of a rewrite taclet within an argument of a skolem symbol occurrence. Then it can be shown that

*σ*(*P*1) ··· *σ*(*Pn*)

*σ*(*Q*)

is an instance of *Rtac*' .

Otherwise, if a rewrite taclet is applied to a term *t* within an argument of a skolem symbol occurrence, it is possible that a single occurrence of *t* in *Q* produces more than one occurrence of *σ*(*t*) in *σ*(*Q*) (like in example [6.3](#_bookmark37), where a single occur- rence of the program variable i in the original formula yields multiple occurrences after the application of *σ*). Provided that the cut-rule and rules treating equations

*.*

are available, it is then possible to perform a cut with the equation *σ*(*t*) = *σ*(*t*) and

apply *tac*'

*.*

to one side of the equation. Afterwards the equations *σ*(*t*) = *σ*(*ti*) can

be used to replace all occurrences of *σ*(*t*) successively. This is illustrated by the following proof tree fragment, in which we use the notation (Γ ▶Δ) = *σ*(*Q*):

*σ*(*P*1)

.

*σ*(*Pn* )

.

*.* . *.* .

Γ1*, σ*(*t*) = *σ*(*t*1) ▶ Δ1 ·· ·

*.*

Γ*n, σ*(*t*) = *σ*(*tn* ) ▶ Δ*n*

*tac*' ∗ *.*

Γ*, σ*(*t*) = *σ*(*t*) ▶ Δ

Γ ▶ Δ

Γ ▶ Δ*, σ*(*t*) = *σ*(*t*)

Corollary 6.5 *Suppose that the proof H of M*Sk(*tac* ) = *ι*Sk(*M* (*tac* )) *only con- sists of applications of taclets tac*'*, and that the concerned taclets tac*' *do not con- tain skolem symbols. Then for each instance φ* = *κ*(*M* (*tac* )) *of the meaning for- mula M* (*tac* ) *there is a proof Hφ.*

Proof. W.l.o.g. we may assume that *ι*Sk and *κ* are equal w.r.t. the instantiations of schema variables of type Variable. Each taclet application within *H* can then be replaced with the proof tree fragment that is obtained from Lem. [6.4](#_bookmark38), for a *σ* that substitutes skolem expressions *s*Sk(*.. .*) = *ι*Sk(*sv*) with the concrete instantia- tion *κ*(*sv*), i.e. in a way such that *σ*(*M*Sk(*tac* )) = *φ*.

* 1. *Treatment of Decomposition Rules*

Lem. [6.4](#_bookmark38) of the last section is not directly applicable to applications of the

, *D*

taclets *D*o

*s*Sk

▢

*s*Sk

(Sect. [5.3](#_bookmark29)), as these taclets contain statement skolem symbols *s*Sk

and *Dec*(*s*Sk). If these symbols are replaced with arbitrary *Java* statements by the

application of a substitution *σ* (as introduced in the previous section), then the

obtained taclet will furthermore be unsound in general.

We circumvent these problems by constructing particular substitutions *σ* of the

*s*Sk

symbols *s*Sk and *Dec*(*s*Sk) with the property that *σ*(*D*o

*s*Sk

), *σ*(*D*▢

) are sound taclets,

so that subsequently Lem. [6.4](#_bookmark38) can be applied for obtaining a proof tree.

Lemma 6.6 *Suppose that σ is a substitution that replaces all skolem symbols of a* *formula ψ, and s*Sk *is a skolem symbol for statements. Then there is a substitution σ*' *that differs from σ only in the symbols s*Sk*, Dec*(*s*Sk)*, such that*

* + 1. *σ*'(*D*o

*s*Sk

)*, σ*'(*D*▢

) *are sound taclets*

* + 1. *There is a proof tree (fragment) whose root is* ▶ *σ*(*ψ*)*, such that the only goal left is* ▶ *σ*'(*ψ*)*.*

*s*Sk

Referring to this lemma it is possible to formulate an analogue of Lem. [6.4](#_bookmark38) for decomposition taclets:

Lemma 6.7 (Lifting of Decompositions) *Suppose that RD is a rule that is de-*

*s*Sk

*scribed by a decomposition taclet D (D* = *D*o

*s*Sk

*or D* = *D*▢

*). If an instance of RD*

*is given by*

*P Q*

*and σ*' *is a substitution of skolem symbols as in Lem.* [*6.6*](#_bookmark40) *w.r.t. D, then there is a proof tree of σ*'(*Q*)*, whose only goal left is the sequent σ*'(*P* )*.*

Proof. First the application of *D* is replaced with an application of the taclet *σ*'(*D*), which is sound by Lem. [6.6](#_bookmark40), ([i](#_bookmark41)) (this substitutes certain occurrences of *s*Sk, *Dec*(*s*Sk) within *P* and *Q*). Subsequently Lem. [6.4](#_bookmark38) can be applied to the resulting rule application w.r.t. *σ*'.

Corollary 6.8 *Suppose that the proof H of M*Sk(*tac* )= *ι*Sk(*M* (*tac* )) *only consists of applications of taclets tac*' *that do not contain skolem symbols, and of applications of decomposition taclets. Then for each instance φ* = *κ*(*M* (*tac* )) *of the meaning formula M* (*tac* ) *there is a proof Hφ.*

Proof. *σ* is chosen as in the proof of Cor. [6.5](#_bookmark39). By repeated application of Lem. [6.6](#_bookmark40), ([ii](#_bookmark42)) it is possible to construct a proof tree with root sequent ▶ *φ* and

a single goal ▶ *σ*'(*M*Sk(*tac* )), with a substitution *σ*' that is chosen according to Lem. [6.6](#_bookmark40) for each skolem symbol *s*Sk for statements.

It is then possible to construct a closed proof tree of ▶ *σ*'(*M*Sk(*tac* )) by trans- forming *H*: Each taclet application within *H* is replaced with the proof tree frag- ment that is obtained from Lem. [6.4](#_bookmark38) or Lem. [6.7](#_bookmark43) (according to the kind of the taclet).

# Conclusions

In this paper, we have outlined how to ensure correctness of derived taclets. Because of limited space, we have only sketched the basic idea and covered only some few kinds of schema variables. The presented concept is completely integrated in the

taclet-based KeY prover, which also supports a bigger class of possible *JavaCardDL*

taclets.

As future work, it remains

* to generalise the concept of skolemisation of meaning formulas,
* to study quantified first-order logics with skolemised statements as ‘atomic’ pro- grams, and
* to explore further areas of application, as for example, proofs of program trans- formation properties.

Taclets are a simple but powerful concept. By their syntactic and semantic simplicity, users are enabled to write new rules and add them to the system easily.

We have shown that, despite this fact, the correctness of the rule base can be efficiently ensured—even for a special purpose logic like *JavaCardDL*.

# Acknowledgement

We would like to thank Martin Giese and Steffen Schlager for useful comments on earlier versions of this paper, as well as Bernhard Beckert and P.H. Schmitt for fruitful discussions. Also we want to thank the anonymous referees and workshop organisers.

# References

1. Ahrendt, W., T. Baar, B. Beckert, R. Bubel, M. Giese, R. H¨ahnle, W. Menzel, W. Mostowski, A. Roth,

S. Schlager and P. H. Schmitt, *The KeY tool*, Software and System Modeling 4 (2005), pp. 32–54.

1. Ahrendt, W., A. Roth and R. Sasse, *Automatic validation of transformation rules for Java verification against a rewriting semantics*, in: G. Sutcliffe and A. Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, LNCS 3835 (2005), pp. 412–426.
2. Beckert, B., *A Dynamic Logic for the Formal Verification of Java Card Programs*, in: I. Attali and

T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041 (2001), pp. 6–24.

1. Beckert, B., M. Giese, E. Habermalz, R. H¨ahnle, A. Roth, P. Ru¨mmer and S. Schlager, *Taclets: a new paradigm for constructing interactive theorem provers*, Revista de la Real Academia de Ciencias

Exactas, F´ısicas y Naturales, Serie A: Matem´aticas 98 (2004), special Issue on Symbolic Computation in Logic and Artificial Intelligence.

1. Beckert, B., R. H¨ahnle and P. H. Schmitt, editors, “Verification of Object-Oriented Software: The KeY Approach,” LNCS 4334, Springer-Verlag, 2007.
2. Beckert, B., S. Schlager and P. H. Schmitt, *An improved rule for while loops in deductive program verification*, in: K.-K. Lau, editor, *Proceedings, Seventh International Conference on Formal Engineering Methods (ICFEM), Manchester, UK*, LNCS 3785 (2005), pp. 315–329.
3. Giese, M., *Taclets and the KeY prover*, in: C. Lu¨th and D. Aspinall, editors, *Intl., Workshop on User Interfaces for Theorem Provers, UITP 2003, Rome, Italy*, Electronic Notes in Theoretical Computer Science (2004), to appear.
4. Gosling, J., B. Joy, G. Steele and G. Bracha, “The Java Language Specification,” Addison Wesley, 2000, 2nd edition.
5. Habermalz, E., “Ein dynamisches automatisierbares interaktives Kalku¨l fu¨r schematische theoriespezifische Regeln,” Ph.D. thesis, Universit¨at Karlsruhe (2000).
6. Harel, D., *Dynamic Logic*, in: D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II*, Reidel, 1984 pp. 497–604.
7. Jacobs, B., J. van den Berg, M. Huisman, M. van Berkum, U. Hensel and H. Tews, *Reasoning about Java classes*, in: *Proceedings, Object-Oriented Programming Systems, Languages and Applications (OOPSLA’98)*, Vancouver, Canada, 1998, pp. 329–340.
8. Ru¨mmer, P., *Ensuring the soundness of taclets – Constructing proof obligations for Java Card DL taclets*, Studienarbeit, Fakult¨at fu¨r Informatik, Universita¨t Karlsruhe (2003).
9. Sasse, B., *Formal correctness of a program logic calculus for the deductive verification of Java programs*, Studienarbeit, Fakult¨at fu¨r Informatik, Universita¨t Karlsruhe (2002).
10. Sun Microsystems, Inc., Palo Alto/CA, “Java Card 2.0 Language Subset and Virtual Machine Specification,” (1997).
11. Trentelman, K., *Proving correctness of javacard dl taclets using bali.*, in: *SEFM*, 2005, pp. 160–169.
12. von Oheimb, D., *Axiomatic semantics for Javalight* , in: S. Drossopoulou, S. Eisenbach, B. Jacobs,

G. T. Leavens, P. Mu¨ller and A. Poetzsch-Heffter, editors, *Proceedings, Formal Techniques for Java Programs, Workshop at ECOOP’00, Cannes, France*, 2000.

1. von Oheimb, D., “Analyzing Java in Isabelle/HOL,” Ph.D. thesis, Institut fu¨r Informatik, Technische Universit¨at Mu¨nchen (2001).
2. Widmann, F., “Crossverification of While Loop Semantics,” Diplomarbeit, Universit¨at Karlsruhe, Fakult¨at fu¨r Informatik (2006).