Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic

Richard Bubel[1] Andreas Roth[2] Philipp Rümmer[3]

Institut für Logik, Komplexität und Deduktionssysteme
Universität Karlsruhe, Germany

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.

Key words: taclets, lightweight tactics, dynamic logic, theorem proving

1 Introduction

Background

Taclets are a new approach for constructing powerful interactive theorem provers [?]. First introduced as *Schematic Theory Specific Rules* [?], 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 compared to an approach based on meta languages like ML, and their seamless integration with graphical user interfaces of theorem provers which they can be efficiently compiled [?] 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

[1] Email:bubel@ira.uka.de
[2] Email:aroth@ira.uka.de
[3] Email:ruemmer1@ira.uka.de

syntax and intuitive 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 KeYproject [**?**] is based on taclets implementing a sequent calculus for JavaCard Dynamic Logic (*JavaCardDL*) [**?**]. *JavaCard* is a subset of *Java* lacking multi-threading, garbage collection 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 is proven in the calculus itself and without escaping to higher order logics. By our work, we extend an approach already described in [**?**] for classical first-order logic to *JavaCardDL*.

Related Work

Related to our approach are other projects for program verification like Bali [**?**,**?**], where consistence and correctness of rules that cover the Java semantics are ensured using Isabelle, or the LOOP project [**?**] 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 for *derived* taclets—further work has been carried out in the KeY project in order to cross-validate a few selected axiomatic rules against the *Java* axiomatisation of Bali [**?**].

Structure of this Paper

In Sect. 1.1 we repeat the most important concepts of classical dynamic logics and *JavaCardDL*. A formal description of taclets and a definition of the basic vocabulary used throughout the paper is given in Sect. 2. The different steps to be performed in order to prove the correctness of derived taclets are described in Sect. 3–5. In Sect. 6 we give a justification of the complete procedure as main theorem. Finally, in Sect. 7 we discuss the current and future work to be done.

1.1 Dynamic Logic

Classical Dynamic Logics

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

- $\langle \mathrm{p} \rangle \phi$ ("diamond p ϕ"): p terminates and after the execution of p formula ϕ holds
- $[\mathrm{p}]\phi$ ("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 $\Sigma = (At_0, Prg_0, Op)$, where At_0, Prg_0 are enumerable sets of propositional variables and atomic programs respectively. Besides the classical propositional operators $\neg, \rightarrow$ the operator set Op contains box $[\mathrm{p}]$ and diamond $\langle \mathrm{p} \rangle$ 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 $\phi \rightarrow \psi$ and $\neg\phi$ are DPDL formulas
- if $\mathrm{p} \in Prg$ is a program and ϕ a formula in PDL then $\langle \mathrm{p} \rangle \phi$ and $[\mathrm{p}]\phi$ are DPDL formulas
- the set Prg of programs is the smallest set satisfying

(i) $Prg_0 \subseteq Prg$

(ii) if $\mathrm{p}, \mathrm{q} \in Prg$ and $\psi \in$ 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 $(S, (\rho_\mathrm{p})_{\mathrm{p} \in Prg})$ with a set S of states, and transition relations $\rho_\mathrm{p} : S \rightarrow S$ which define the semantics of each program $\mathrm{p} \in Prg$. The relations ρ_p have to adhere certain conditions w.r.t. the program constructors (;, **if**−**else**, etc.) from the definition above, for example, program composition $\rho_{\mathrm{p};\mathrm{q}} = \rho_\mathrm{q} \circ \rho_\mathrm{p}$.

An excerpt from an axiom system for DPDL in terms of sequent calculus rules is given in Table 1.

DPDL is useful to reason about program properties induced by the program constructors. However, as a consequence of constructing programs from atomic (anonymous) programs without any fixed semantics, they lack possibilities to talk about individual programs and, thus, about functional properties.

Like the step from propositional to first-order logic, one extends DPDL to deterministic quantified dynamic logics (DQDL). DQDL extends the propositional part to full first-order logic (with equality and universe D), and on the program side it replaces the anonymous atomic programs with assignments of the form v=t, where v is a variable and t an arbitrary term. In general, each program state $s \in S$ is assigned a first order structure (D, I) and a variable valuation $\sigma : Var \rightarrow D$ respecting $\rho_{x=t}(s) = s'$ with $\sigma' = \sigma_x^{t^{(D,I),\sigma}}$.

Again a relatively[4] complete calculus can be given, the corresponding

[4] Usually DQDL is interpreted in an arithmetic structure.

$$\frac{\Gamma \vdash \langle \textbf{if}\ (\psi)\ \{\mathrm{p};\ \textbf{while}\ (\psi)\ \{\mathrm{p}\}\ \}\ \textbf{else}\ \{\}\rangle\phi, \Delta}{\Gamma \vdash \langle \textbf{while}\ (\psi)\ \{\mathrm{p}\}\rangle\phi, \Delta} \quad (1)$$

$$\frac{\Gamma, \psi \vdash \langle \mathrm{p}\rangle\phi,\ \Delta \quad \Gamma, \neg\psi \vdash \langle \mathrm{q}\rangle\phi,\ \Delta}{\Gamma \vdash \langle \textbf{if}\ (\psi)\ \{\mathrm{p}\}\ \textbf{else}\ \{\mathrm{q}\}\rangle\phi,\ \Delta} \quad (2)$$

$$\frac{\Gamma \vdash \langle \mathrm{p}\rangle\langle \mathrm{q}\rangle\phi,\ \Delta}{\Gamma \vdash \langle \mathrm{p;q}\rangle\phi,\ \Delta} \quad (3) \qquad \frac{\Gamma^{\{\mathrm{x}\leftarrow\mathrm{z}\}}, \mathrm{x} \doteq \mathrm{t}^{\{\mathrm{x}\leftarrow\mathrm{z}\}} \vdash \phi, \Delta^{\{\mathrm{x}\leftarrow\mathrm{z}\}}}{\Gamma \vdash \langle \mathrm{x{=}t}\rangle\phi, \Delta} \quad (4)$$

Table 1
DPDL/DQDL Axiomatisation (excerpt). z is a new variable.

assignment rule is shown in Table 1.

Example 1.1 *For the universe $D = \mathbb{N}$ of natural numbers, the DQDL formula* $\langle \mathrm{x{=}3;;y{=}x;}\rangle \mathrm{y} \doteq \mathrm{x}$ *can be proven valid with the rules of Table 1 as the proof on the right shows.*

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{\mathrm{x} \doteq 3, \mathrm{y} \doteq x \vdash \mathrm{y} \doteq \mathrm{x}}\ (close)}{\mathrm{x} \doteq 3 \vdash \langle \mathrm{y{=}x;}\rangle \mathrm{y} \doteq \mathrm{x}}\ (4)}{\vdash \langle \mathrm{x{=}3;}\rangle\langle \mathrm{y{=}x;}\rangle \mathrm{y} \doteq \mathrm{x}}\ (4)}{\vdash \langle \mathrm{x{=}3;y{=}x;}\rangle \mathrm{y} \doteq \mathrm{x}}\ (3)$$

JavaCardDL

The step from academic languages as described in the previous paragraphs to real world programming languages like *JavaCard* [?,?] leads to several complications. In the next few paragraphs, we introduce some features of *JavaCardDL* [?]. First some preliminaries:

- Formulas must not occur in *JavaCardDL* programs, instead *Java* expressions of type **boolean** are used as guards.
- The set of variables $Var = PVar \uplus LVar$ is the disjoint union of program variables *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 $x \in LVar$ and $\mathrm{o}, \mathrm{u} \in PVar$, then $\forall x.\langle \mathrm{o{=}u;}\rangle x \doteq \mathrm{u}$ is a well-formed *JavaCardDL* formula, whereas $\forall x.\langle \mathrm{o{=}}x\mathrm{;}\rangle x \doteq \mathrm{u}$ is not.
- All states have the same universe D, and predicates and logical variables are assumed to be rigid.

A sequent calculus covering *JavaCard* has to cope with aliasing, side-effects, abrupt termination as result of thrown exceptions, **break**s, **continue**s or **return**s and more. The KeY approach is led by 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) is not available in JavaCardDL due to abrupt termination. For example*

$$\vdash \langle \mathrm{l}: \{\ \textbf{if}\ (\mathrm{v} == 0)\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \textbf{else}\ \{\ \mathrm{v} = 0;\ \}\ \mathrm{v} = 3;\}\ \rangle \mathrm{v} \doteq 3$$

cannot be decomposed to

$$\vdash \langle \mathrm{l}: \{\ \textbf{if}\ (\mathrm{v} == 0)\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \textbf{else}\ \{\ \mathrm{v} = 0;\ \}\ \}\rangle\langle \mathrm{v} = 3\rangle \mathrm{v} \doteq 3,$$

as this is obviously not equivalent for $\mathrm{v} = 0$*.*

Decomposition was essential for DPDL and DQDL in order to reduce the complexity of programs stepwise to atomic programs or assignments, which can be handled by rules of the calculus without having 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 '$..\ \circ_1\ ...$' whose inactive prefix '$..$' matches on all preceding labels, opening braces or **try** blocks. Consider the following rule:

$$\frac{\#\mathsf{b} \doteq \textbf{true} \vdash \langle ..\ \{\#\mathsf{sta1}\}\ ...\rangle\phi \qquad \#\mathsf{b} \doteq \textbf{false} \vdash \langle ..\ \{\#\mathsf{sta2}\}\ ...\rangle\phi}{\vdash \langle ..\ \textbf{if}\ (\#\mathsf{b})\ \{\#\mathsf{sta1}\}\ \textbf{else}\ \{\#\mathsf{sta2}\}\ ...\rangle\phi} \tag{5}$$

where $\#\mathsf{b}$ is a side-effect free boolean expression and $\#\mathsf{sta1}$, $\#\mathsf{sta2}$ are arbitrary *JavaCard* statements.

Example 1.3 (Example 1.2 continued) *Applying rule (5) to*

$$\vdash \langle \mathrm{l}: \{\ \textbf{if}\ (\mathrm{v} == 0)\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \textbf{else}\ \{\ \mathrm{v} = 0;\ \}\ \mathrm{v} = 3;\}\rangle \mathrm{v} \doteq 3$$

where $\circ_1$ *corresponds to the program between* 'l:{' *(inactive program prefix) and* 'v = 3;}' *(suffix of the program context) now yields the two sequents*

(i) $(\mathrm{v}{==}0) \doteq \textbf{true} \vdash \langle \mathrm{l}: \{\ \{\ \textbf{break}\ \mathrm{l};\ \}\ \mathrm{v} = 3;\ \}\rangle \mathrm{v} \doteq 3$ *and*

(ii) $(\mathrm{v}{==}0) \doteq \textbf{false} \vdash \langle \mathrm{l}: \{\ \{\ \mathrm{v} = 0;\ \}\ \mathrm{v} = 3;\ \}\rangle \mathrm{v} \doteq 3$

2 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 [?].

Taclets by Example

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

$$\texttt{find}(1 + 1)\ \texttt{replacewith}(2)$$

In a taclet—in addition to the logical content of the described rule—an operational 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] in taclets may contain elements from a set SV of *schema variables*. An *instantiation* $\iota(\mathsf{v})$ of a schema variable $\mathsf{v} \in SV$ 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 $\iota(\mathsf{i})$ 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 be canonically continued on schematic expressions:

$$\iota(op(e_1, \ldots, e_n)) = \begin{cases} \iota(op) & \text{if } n = 0 \text{ and } op \in SV \\ op(\iota(e_1), \ldots, \iota(e_n)) & \text{otherwise} \end{cases} \tag{6}$$

Thus e describes a set $\{\iota(e) \mid \iota \text{ is an instantiation map for every } \mathsf{v} \in SV\}$ of concrete expressions. For instance, a taclet $\texttt{find}(\mathsf{i} + \mathsf{i})\ \texttt{replacewith}(2 * \mathsf{i})$ contains schematic terms over $\{\mathsf{i}\}$. Applied on a sequent containing the term $3 + 3$, i is instantiated with $\iota(\mathsf{i}) = 3$ and the taclet replaces $\iota(\mathsf{i} + \mathsf{i}) = 3 + 3$ with $\iota(2 * \mathsf{i}) = 2 * 3$ in the new goal.

[5] By *expression* we denote syntactical elements like terms or formulas, but in the context of *JavaCardDL* also *Java* programs.

Taclet Syntax

`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:

- 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(` $\vdash \phi \rightarrow \psi$`)` `replacewith(`$\phi \vdash \psi$`)` (over the schema variables $\{\phi, \psi\}$) 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 described behind the key word `addrules`. Though a theoretical treatment in accordance with the concepts of this paper is possible [**?**] we omit this feature in the sequel.

Often more information on the sequent a taclet is applicable to is needed. Such side conditions are described by the following optional taclet constituents:

- 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 $\iota(\mathtt{v})$ to occur unbound in $\iota(\mathtt{s})$.
  - `v new depending on s`, which introduces a new skolem symbol $\iota(\mathtt{v})$ (possibly depending on free "meta variables" occurring in $\iota(\mathtt{s})$).

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

$$\begin{array}{c} [\mathtt{if}\ (\mathit{context})]\ [\mathtt{find}\ (f)]\ [\mathtt{varcond}\ (c_1,\ldots,c_k)] \\ [\mathtt{replacewith}\ (rw_1)]\ [\mathtt{add}\ (add_1)]; \\ \vdots \qquad\qquad \vdots \\ [\mathtt{replacewith}\ (rw_n)]\ [\mathtt{add}\ (add_n)] \end{array} \tag{7}$$

For $i = 1 \dots n$, *context* and add_i stand for a schematic sequent, f and rw_i for a schematic term, formula, or sequent but all of the same kind. $c_1, \dots, c_k$ 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 rw_i are declared to be never empty: we assume that skipping `replacewith` is a shorthand for $rw_i = f$, a skipped `find` means $f = \ \vdash \mathit{false}$, and *false* always occurs in succedents of sequents. A skipped `if`- or `add`-part means $\mathit{context} = \ \vdash$ or $add_i = \ \vdash$ (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 SV_{tac} denote the schema variables contained in a taclet *tac*. Schema variables $\mathsf{v} \in SV_{tac}$ are assigned to one out of a predefined list of types, each having special properties concerning admissible instantiations $\iota(\mathsf{v})$. An instrument to define these properties is to introduce *prefix sets* (denoted by $\Pi_l(\mathsf{v})$, $\Pi_{pv}(\mathsf{v})$, and $\Pi_{jmp}(\mathsf{v})$) 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*, $\iota(\mathsf{v})$ must be of that sort. Moreover, $\iota(\mathsf{v})$ must be a logical variable. For $\mathsf{v} \neq \mathsf{v}' \in SV$: if v' is a Variable schema variable then $\iota(\mathsf{v}) \neq \iota(\mathsf{v}')$. $\iota(\mathsf{v})$ must not occur bound in $\iota(\mathsf{v}'')$ for all $\mathsf{v}'' \in SV$.
- Term, then v is assigned a *sort*, $\iota(\mathsf{v})$ must be of that sort.

  v is assigned a set $\Pi_l(\mathsf{v}) \subseteq SV$ of schema variables. $\Pi_l(\mathsf{v})$ is defined to be the smallest set with, for all constituents of *tac*, if v occurs in the scope of a Variable schema variable $\mathsf{v}' \in SV$ then $\mathsf{v}' \in \Pi_l(\mathsf{v})$ except there is a variable condition v' `not free in` v declared in t. v is assigned a set $\Pi_{pv}(\mathsf{v})$ which is the smallest set of program variables that occur but are not declared in *tac* or are declared above[6] every occurrence of v.

  We require from instantiations ι: If, for some $\mathsf{v}' \in SV_{tac}$, $\iota(\mathsf{v}')$ is a logical variable that occurs unbound in $\iota(\mathsf{v})$ then $\mathsf{v}' \in \Pi_l(\mathsf{v})$; if $\iota(\mathsf{v}')$ is a program variable that occurs undeclared in $\iota(\mathsf{v})$ then $\mathsf{v}' \in \Pi_{pv}(\mathsf{v})$.
- Formula, then as for Term, v is assigned $\Pi_l(\mathsf{v})$ and $\Pi_{pv}(\mathsf{v})$. v must fulfil the same conditions concerning these sets.
- Statement, then $\iota(\mathsf{v})$ is a *JavaCard* statement. Again, v is assigned $\Pi_{pv}(\mathsf{v})$ and it must satisfy the same conditions as above concerning this set.

  v is assigned a set $\Pi_{jmp}(\mathsf{v})$ 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 $\iota(\mathsf{v})$ with a target

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

not in $\iota(\mathsf{v})$ then $jst \in \Pi_{jmp}(\mathsf{v})$.[7] Usually Statement schema variables have names starting with # to distinguish them from regular *Java* elements.

- ProgramVariable, then $\iota(\mathsf{v})$ is a local program variable or class attribute of *Java*. v is assigned a *Java* type and $\iota(\mathsf{v})$ must be of that type. Again, names of this kind of variable start with #.
- ProgramContext, then $\iota(\mathsf{v})$ is a program transformation[8] pt that takes a *Java* program element α and delivers a new program element $pt(\alpha)$, such that $pt(\alpha)$ is a sequence of statements of which the first one contains α and has only opening braces, opening **try** blocks, etc., in front. For this case, the continuation of the instantiation map (6) is then modified to

$$\iota(op(e_1, \ldots, e_n)) := pt(\iota(e_1))$$

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.

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

$$\begin{aligned} &\texttt{find}(\mathsf{t}) \quad \texttt{replacewith}(0) \quad \texttt{add}(\mathsf{t} \doteq 0 \ \vdash\ ); \\ &\phantom{\texttt{find}(\mathsf{t})} \quad \texttt{replacewith}(\mathsf{t}) \quad \texttt{add}(\ \vdash\ \mathsf{t} \doteq 0) \end{aligned} \tag{8}$$

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

$$\texttt{find}(\langle \#\mathsf{sta}\ \mathrm{x{+}{+};}\rangle\phi) \quad \texttt{replacewith}(\langle \#\mathsf{sta}\ \mathrm{x{=}x{+}1;}\rangle\phi) \tag{9}$$

Another 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.*

$$\begin{aligned} &\texttt{find}(\langle \mathrm{l}:\ \textbf{if}\ \ (\mathrm{x{=}{=}0})\ \#\mathsf{sta1}\ \textbf{else}\ \#\mathsf{sta2}\rangle\phi) \\ &\quad \texttt{replacewith}(\langle \mathrm{l}:\ \#\mathsf{sta1}\ \rangle\phi) \quad \texttt{add}(\mathrm{x} \doteq 0\ \vdash\ ); \\ &\quad \texttt{replacewith}(\langle \mathrm{l}:\ \#\mathsf{sta2}\ \rangle\phi) \quad \texttt{add}(\ \vdash\ \mathrm{x} \doteq 0) \end{aligned} \tag{10}$$

[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 $\iota(\mathsf{v})$ must be an expression.

Semantics

Taclets have a precise operational semantics, which is described in detail in [?], 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 sequents as follows:

$$\begin{aligned}(\Gamma_1 \vdash \Delta_1) \cup (\Gamma_2 \vdash \Delta_2) &:= \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2\\ (\Gamma_1 \vdash \Delta_1) \subseteq (\Gamma_2 \vdash \Delta_2) &\text{ iff } \Gamma_1 \subseteq \Gamma_2 \text{ and } \Delta_1 \subseteq \Delta_2\end{aligned}$$

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

$$\frac{\mathit{rw}_1 \cup \mathit{add}_1 \cup (\Gamma \vdash \Delta) \qquad \ldots \qquad \mathit{rw}_k \cup \mathit{add}_k \cup (\Gamma \vdash \Delta)}{f \cup (\Gamma \vdash \Delta)} \tag{11}$$

where $\Gamma \vdash \Delta$ is an arbitrary sequent with $\mathit{context} \subseteq f \cup (\Gamma \vdash \Delta)$.

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

$$\frac{\mathit{seq}[\mathit{rw}_1] \cup \mathit{add}_1 \cup (\Gamma \vdash \Delta) \qquad \ldots \qquad \mathit{seq}[\mathit{rw}_k] \cup \mathit{add}_k \cup (\Gamma \vdash \Delta)}{\mathit{seq}[f] \cup (\Gamma \vdash \Delta)}$$

where $\Gamma \vdash \Delta$ is an arbitrary sequent with $\mathit{context} \subseteq \mathit{seq}[f] \cup (\Gamma \vdash \Delta)$.

In Sect. 4, the notion of *meaning formulas* is derived that makes the meaning of these rule schemas induced by taclets more precise.

Because of 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 reduces drastically the cognitive burden on the user. For an extensive account on user interaction see [?].

3 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 [?] which has already provided this kind of bootstrapping for classical first-order logic.

Given a taclet *tac*, we first derive a *meaning formula* $M(\mathit{tac})$ (see Sect. 4), 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:

$$\texttt{find}(\mathit{true} \wedge\ \phi\ \vdash\ )\quad \texttt{replacewith}(\phi \vdash)$$

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

$$M(tac_0) = \neg\phi \rightarrow \neg(\mathit{true} \wedge \phi) \text{ or equivalently } (\mathit{true} \wedge \phi) \rightarrow \phi$$

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) 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

$$M_{\text{Sk}}(tac_0) = (\mathit{true} \wedge \phi_{\text{Sk}}) \rightarrow \phi_{\text{Sk}},$$

where ϕ_{Sk} is a new nullary predicate. We call these formulas $M_{\text{Sk}}(tac)$ *taclet proof obligations*. $M_{\text{Sk}}(tac)$ is a *JavaCardDL* formula (with a slightly extended vocabulary) 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 [?] and sketched in Sect. 6.

On a semantic level, this theorem can be justified by arguing that if an application 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 [?], 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 $M_{\text{Sk}}(tac)$.

4 Meaning Formulas of Taclets

The basis for our reasoning about the correctness of taclets is a *meaning formula* [?] 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] 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 $(\Gamma \vdash \Delta)^* := \bigwedge \Gamma \rightarrow \bigvee \Delta$, in particular $(\vdash \phi)^* = \phi$ and $(\phi \vdash)^* = \neg\phi$. Furthermore, in this section by the validity of a sequent we mean the validity of $(\Gamma \vdash \Delta)^*$. We define a (sequent) calculus C to be *sound* if only valid sequents are derivable in C. We conceive rules

$$\frac{P_1 \quad \dots \quad P_n}{Q}$$

as relations between tuples of sequents (the premisses) and single sequents (the conclusion) and define that a rule $R \in C$ is sound if for all tuples $(\langle P_1, \dots, P_k\rangle, Q) \in R$:

$$\text{if } P_1, \dots, P_k \text{ are valid, then } Q \text{ is valid.} \tag{12}$$

For the calculus C we can state:

Lemma 4.1 *C is sound if all rules $R \in C$ are sound.*

The rules R_{tac} we are interested in are defined through taclets *tac* over a set SV of schema variables in the form as defined in (7). Assuming first that the `find`-part is a sequent, taclets induce the rule schema (11). To apply Lem. 4.1, for each instantiation ι of SV, (12) must be shown for $k = n$, $P_i = \iota(rw_i \cup add_i \cup \Gamma \vdash \Delta)$ $(i = 1 \dots n)$, and $Q = \iota(f \cup \Gamma \vdash \Delta)$. Since the formulas of $\Gamma \vdash \Delta$ which are not in *context* are arbitrary and not influenced by the rule application we can simply omit them and show the lemma for $P_i = \iota(rw_i \cup add_i \cup context)$ $(i = 1 \dots n)$ and $Q = \iota(f \cup context)$. 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) can be strengthened to the local implication, namely that $P_1^* \wedge \dots \wedge P_n^* \rightarrow Q^*$ must be valid.

Since ι, as defined by (6), treats propositional junctors as a homomorphism and the operator $(\cdot)^*$ is a homomorphism regarding the union of sequents up to propositional transformations, this formula can now be simplified as follows:

$$P_1^* \wedge \dots \wedge P_n^* \rightarrow Q^* = \bigwedge_{i=1}^{n} \iota(rw_i \cup add_i \cup context)^* \rightarrow \iota(f \cup context)^* \tag{13}$$

$$= \iota\Big(\bigwedge_{i=1}^{n}(rw_i^* \vee add_i^*) \rightarrow (f^* \vee context^*)\Big). \tag{14}$$

If (14) is proven for all instantiations ι, then the rule R_{tac} represented by *tac* is sound.

In the next definition our previously made assumptions are revoked: the variable condition sv_i `new depending on`... introduces new skolem functions.

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

If $P_1, \ldots, P_n$ contain skolem symbols that do not occur in Q, the interpretation of the symbols can be regarded as universally quantified in (12) by the usual definition of 'valid'. Because of their negation in (13), they are existentially bound in the meaning formula. 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 $f \leftrightarrow rw_i$ or equation $f = rw_i$ to the antecedent.

Definition 4.2 (Meaning Formula) *Each taclet tac, as declared in (7), is assigned an* unquantified meaning formula tac^**, which is defined by:*

$$tac^* := \begin{cases} \bigwedge_{i=1}^n (rw_i^* \lor add_i^*) \to (f^* \lor context^*) & \textit{if } f \textit{ is a sequent} \\ \bigwedge_{i=1}^n \left(f \doteq rw_i \to add_i^*\right) \to context^* & \textit{if } f \textit{ is a term} \\ \bigwedge_{i=1}^n \left((f \leftrightarrow rw_i) \to add_i^*\right) \to context^* & \textit{if } f \textit{ is a formula} \end{cases}$$

Suppose $\mathsf{sv}_1, \ldots, \mathsf{sv}_k \in SV_{tac}$ *are all schema variables, which tac contains a variable condition* sv_i `new depending on`... *for.* $M(tac) := \exists \mathsf{x}_1 \ldots \exists \mathsf{x}_k . \phi$ *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 continued) *The taclets tac_1, tac_2, and tac_3 defined through (8), (9), and (10), resp., have (after applying some propositional equivalence transformations) the following meaning formulas:*

$$M(tac_1) = \big(\mathsf{t} \doteq 0 \land \mathsf{t} \doteq 0\big) \lor \big(\mathsf{t} \doteq \mathsf{t} \land \neg(\mathsf{t} \doteq 0)\big) \tag{15}$$

$$M(tac_2) = \langle \#\mathsf{sta}\ \mathrm{x}{+}{+};\rangle\phi \leftrightarrow \langle \#\mathsf{sta}\ \mathrm{x}{=}\mathrm{x}{+}1;\rangle\phi \tag{16}$$

$$\begin{aligned} M(tac_3) = \quad & \big((\langle \mathrm{l}:\ \textbf{if}\ (\mathrm{x}{=}{=}0)\ \#\mathsf{sta1}\ \textbf{else}\ \#\mathsf{sta2}\rangle\phi \leftrightarrow \langle \mathrm{l}:\ \#\mathsf{sta1}\ \rangle\phi) \\ & \land\ \mathrm{x} \doteq 0\big) \\ \lor & \big((\langle \mathrm{l}:\ \textbf{if}\ (\mathrm{x}{=}{=}0)\ \#\mathsf{sta1}\ \textbf{else}\ \#\mathsf{sta2}\rangle\phi \leftrightarrow \langle \mathrm{l}:\ \#\mathsf{sta2}\ \rangle\phi) \\ & \land\ \neg(\mathrm{x} \doteq 0)\big) \end{aligned} \tag{17}$$

5 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 do however not occur bound within the formula (resp. when considering 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 variables of type ProgramVariable are instantiated with the same concrete variable (which is not possible for Variable schema variables by the definitions of Sect. 2). 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 [?].

5.1 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 $\mathit{Func}_{\mathrm{Sk}}$ and $\mathit{Pred}_{\mathrm{Sk}}$ (resp.). The signature

$$\alpha(s_{\mathrm{Sk}}) = \begin{cases} (S, S_1, \ldots, S_n, T_1, \ldots, T_k) & \text{for } s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \\ (S_1, \ldots, S_n, T_1, \ldots, T_k) & \text{for } s_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}} \end{cases}$$

of a symbol $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$ consists of

- a result sort S, if $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}}$,
- a finite sequence $S_1, \ldots, S_n$ of sorts that determines the number and kinds of term arguments; this sequence corresponds to the signature of ordinary predicate symbols,

- a finite sequence $T_1, \dots, T_k$ of *Java* types, which are the component types of a tuple of program variables with which occurrences of s_{Sk} are equipped.

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

If $s_{\mathrm{Sk}} \in \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}}$ is a symbol with the signature $\alpha(s_{\mathrm{Sk}})$ as above, $t_1, \dots, t_n$ are terms of the sorts $S_1, \dots, S_n$ and $\mathrm{pv}_1, \dots, \mathrm{pv}_k \in \mathit{PVar}$ are program variables of the types $T_1, \dots, T_k$, then

$$s_{\mathrm{Sk}}(t_1, \dots, t_n; \mathrm{pv}_1, \dots, \mathrm{pv}_k)$$

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 $\mathit{Statement}_{\mathrm{Sk}}$. The signature $\alpha(\mathit{st}_{\mathrm{Sk}}) = (T_1, \dots, T_k, m)$ of a symbol $\mathit{st}_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ consists of

- a finite sequence $T_1, \dots, T_k$ 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 $\mathit{st}_{\mathrm{Sk}}$ within programs.

The symbols $\mathit{Statement}_{\mathrm{Sk}}$ extend the definition of well-formed *Java* programs, i.e. the following (informal) rule is added to the *Java* grammar [**?**]:

If $\mathit{st}_{\mathrm{Sk}} \in \mathit{Statement}_{\mathrm{Sk}}$ is a symbol with $\alpha(\mathit{st}_{\mathrm{Sk}}) = (T_1, \dots, T_k, m)$, $\mathrm{pv}_1, \dots, \mathrm{pv}_k$ are program variables of the types $T_1, \dots, T_k$ and $\mathit{jst}_1, \dots, \mathit{jst}_m$ are *Java* statements of the following kinds [10]

- **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

$$\mathit{st}_{\mathrm{Sk}}(\mathrm{pv}_1, \dots, \mathrm{pv}_k; \mathit{jst}_1; \dots; \mathit{jst}_m)$$

is a statement.

5.2 From Meaning Formula to Proof Obligation

From now on we suppose that a taclet *tac* with meaning formula $M(\mathit{tac})$ is fixed. Let $\mathit{SV}_{\mathit{tac}}$ be the set of schema variables $M(\mathit{tac})$ contains. We define an instantiation ι_{Sk} over $\mathit{SV}_{\mathit{tac}}$ 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:

[10] Which are exactly the reasons that can lead to an abrupt termination of a statement, see [**?**].

- If $\mathsf{x} \in SV_{tac}$ is of type $\mathsf{Variable}$, then $\iota_{\mathrm{Sk}}(\mathsf{x}) \in LVar$ is a new logical variable that has the same sort as x.
- If $\mathsf{sv} \in SV_{tac}$ is of type Term, $\mathsf{Formula}$ or $\mathsf{Statement}$, then let $\{\mathrm{pv}_1, \ldots, \mathrm{pv}_k\} = \Pi_{pv}(\mathsf{sv})$ be the program variables that can occur undeclared in instantiations of sv. Let $T_1, \ldots, T_k$ be the *Java* types of $\mathrm{pv}_1, \ldots, \mathrm{pv}_k$.
- If $\mathsf{sv} \in SV_{tac}$ is of type Term, then

$$\iota_{\mathrm{Sk}}(\mathsf{sv}) = f_{\mathrm{Sk}}(v_1, \ldots, v_l; \mathrm{pv}_1, \ldots, \mathrm{pv}_k)$$

  is a term, where
  - $v_1, \ldots, v_l$ with $v_i = \iota_{\mathrm{Sk}}(\mathsf{x}_i)$ are the instantiations of $\mathsf{x}_1, \ldots, \mathsf{x}_l \in SV_{tac}$, which are distinct $\mathsf{Variable}$ schema variables determined by the prefix $\Pi_l(\mathsf{sv}) = \{\mathsf{x}_1, \ldots, \mathsf{x}_l\}$ of sv in *tac*
  - and $f_{\mathrm{Sk}} \in Func_{\mathrm{Sk}}$ denotes a new skolem symbol with signature

$$\alpha(f_{\mathrm{Sk}}) = (S, S_1, \ldots, S_l, T_1, \ldots, T_k)$$

    where S is the sort of sv and $S_1, \ldots, S_l$ are the sorts of $v_1, \ldots, v_l$.
- Analogously, if $\mathsf{sv} \in SV_{tac}$ is a schema variable of type $\mathsf{Formula}$, then

$$\iota_{\mathrm{Sk}}(\mathsf{sv}) = p_{\mathrm{Sk}}(v_1, \ldots, v_l; \mathrm{pv}_1, \ldots, \mathrm{pv}_k)$$

  is a formula containing a new skolem symbol $p_{\mathrm{Sk}} \in Pred_{\mathrm{Sk}}$ for formulas.
- If $\mathsf{sv} \in SV_{tac}$ is a schema variable of type $\mathsf{Statement}$, then two additional (and new) program variables are needed: t_{sv} of *Java* type **Throwable**, and d_{sv} of *Java* type **int** (the latter variable is used in Sect. 5.3). Let $\{jst_1, \ldots, jst_m\} = \Pi_{jmp}(\mathsf{sv})$ be jump statements that can occur uncaught in instantiations of sv. The instantiation $\iota_{\mathrm{Sk}}(\mathsf{sv})$ of sv is the statement [11]

$$\iota_{\mathrm{Sk}}(\mathsf{sv}) = st_{\mathrm{Sk}}(\mathrm{pv}_1, \ldots, \mathrm{pv}_k, \mathrm{t}_{\mathsf{sv}}, \mathrm{d}_{\mathsf{sv}}; jst_1; \ldots; jst_m; \textbf{throw}\ \mathrm{t}_{\mathsf{sv}})$$

  where st_{Sk} denotes a new skolem symbol for statements with signature $\alpha(st_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m+1)$.

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

$$M_{\mathrm{Sk}}(tac) := \iota_{\mathrm{Sk}}(M(tac))$$

Example 5.1 (Example 4.3 continued) *From the meaning formulas of the taclets tac_1, tac_2 and tac_3 the following proof obligations are derived:*

[11] We always add a **throw**-statement, as instantiations of sv may always terminate abruptly through an exception regardless of $\Pi_{jmp}(\mathsf{sv})$.

$$M_{\text{Sk}}(tac_1) = \big(t_{\text{Sk}} \doteq 0 \land t_{\text{Sk}} \doteq 0\big) \lor \big(t_{\text{Sk}} \doteq t_{\text{Sk}} \land \neg(t_{\text{Sk}} \doteq 0)\big) \tag{18}$$

$$M_{\text{Sk}}(tac_2) = \tag{19}$$

$$\begin{aligned}&\langle sta_{\text{Sk}}(\mathrm{v},\ \mathrm{t}_{\#\mathsf{sta}},\ \mathrm{d}_{\#\mathsf{sta}};\ \mathbf{throw}\ \mathrm{t}_{\#\mathsf{sta}});\ \mathrm{v{+}{+}};\rangle p_{\text{Sk}}(\mathrm{v}) \leftrightarrow \\ &\langle sta_{\text{Sk}}(\mathrm{v},\ \mathrm{t}_{\#\mathsf{sta}},\ \mathrm{d}_{\#\mathsf{sta}};\ \mathbf{throw}\ \mathrm{t}_{\#\mathsf{sta}});\ \mathrm{v{=}v{+}1};\rangle p_{\text{Sk}}(\mathrm{v})\end{aligned} \tag{20}$$

$$M_{\text{Sk}}(tac_3) = \tag{21}$$

$$\begin{aligned}&\big((\langle \mathrm{l}:\ \mathbf{if}\ (\mathrm{x{=}{=}0})\ \beta_1\ \mathbf{else}\ \beta_2\rangle p_{\text{Sk}}(\mathrm{x}) \leftrightarrow \langle \mathrm{l}:\ \beta_1\rangle p_{\text{Sk}}(\mathrm{x})) \land \mathrm{x} \doteq 0\big) \lor \\ &\big((\langle \mathrm{l}:\ \mathbf{if}\ (\mathrm{x{=}{=}0})\ \beta_1\ \mathbf{else}\ \beta_2\rangle p_{\text{Sk}}(\mathrm{x}) \leftrightarrow \langle \mathrm{l}:\ \beta_2\rangle p_{\text{Sk}}(\mathrm{x})) \land \neg(\mathrm{x} \doteq 0)\big)\end{aligned} \tag{22}$$

where we use the abbreviations

$$\begin{aligned}\beta_1 &= sta1_{\text{Sk}}(\mathrm{x},\ \mathrm{t}_{\#\mathsf{sta1}},\ \mathrm{d}_{\#\mathsf{sta1}};\ \mathbf{break}\ \mathrm{l};\ \mathbf{throw}\ \mathrm{t}_{\#\mathsf{sta1}});\\ \beta_2 &= sta2_{\text{Sk}}(\mathrm{x},\ \mathrm{t}_{\#\mathsf{sta2}},\ \mathrm{d}_{\#\mathsf{sta2}};\ \mathbf{break}\ \mathrm{l};\ \mathbf{throw}\ \mathrm{t}_{\#\mathsf{sta2}});\end{aligned}$$

5.3 Decomposition Rules

Calculus rules for *JavaCardDL* programs always modify the leading statements within a program block (see Sect. 1). 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 that a "naive" decomposition rule for *JavaCardDL* cannot be posed due to abrupt termination, we define a family of decomposition rules specifically for statement skolem symbols. These rules cope with abrupt termination by applying a transformation to the statement $\alpha = st_{\text{Sk}}(\ldots)$. This transformation splits α in two parts $\alpha_1 = st'_{\text{Sk}}(\ldots)$ and α_2, such that the concatenation $\alpha_1; \alpha_2$ 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

$$\langle ..\ st_{\text{Sk}}(\ldots); \beta\ ...\rangle\phi \leftrightarrow \langle st'_{\text{Sk}}(\ldots)\rangle\langle ..\ \alpha_2; \beta\ ...\rangle\phi \tag{23}$$

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 $st_{\text{Sk}} \in \mathit{Statement}_{\text{Sk}}$ that occurs within ι_{Sk} a second new skolem symbol $\mathit{Dec}(st_{\text{Sk}})$ is introduced, which has the same signature as st_{Sk} except for the jump table:

$$\alpha(st_{\text{Sk}}) = (T_1, \ldots, T_k, m) \qquad \Longrightarrow \qquad \alpha(\mathit{Dec}(st_{\text{Sk}})) = (T_1, \ldots, T_k, 0).$$

Following equivalence (23), two decomposition taclets $D^{\diamond}_{st_{\text{Sk}}}$ and $D^{\square}_{st_{\text{Sk}}}$ for diamond and box modalities (resp.) are introduced for each statement skolem symbol st_{Sk} that occurs in ι_{Sk}. We only give the definition of $D^{\diamond}_{st_{\mathrm{Sk}}}$, as the taclet for boxes is obtained analogously:

$$D^{\diamond}_{st_{\mathrm{Sk}}}: \quad \{ \quad \texttt{find (} \langle ..\ st_{\mathrm{Sk}}(\mathsf{p}_1, \ldots, \mathsf{p}_k; \#\mathsf{jst}_1; \ldots; \#\mathsf{jst}_m);\ \ ...\rangle\phi \texttt{)}$$
$$\texttt{replacewith (} \langle Dec(st_{\mathrm{Sk}})(\mathsf{p}_1, \ldots, \mathsf{p}_k);\rangle\langle ..\ ic\ ...\rangle\phi \texttt{) } \}$$

where $\mathsf{p}_1, \ldots, \mathsf{p}_k$ are schema variables for program variables, $\#\mathsf{jst}_1, \ldots, \#\mathsf{jst}_m$ are variables for statements corresponding to the signature $\alpha(st_{\mathrm{Sk}})$ 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):

```
{         if  (  p_k == 1  )  #jst_1
     else if  (  p_k == 2  )    ...
     else if  (  p_k == m  )  #jst_m  }
```

In this statement at most one of the jump statements represented by the schema variables $\#\mathsf{jst}_1, \ldots, \#\mathsf{jst}_m$ 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:*

$$\frac{\vdash\ \langle st'_{\mathrm{Sk}}(\mathrm{t}, \mathrm{d})\rangle\langle \textbf{try}\ \{\ \textbf{if}\ (\mathrm{d} == 1)\ \textbf{throw}\ \mathrm{t};\ \}\ \textbf{catch}\ (\textbf{Exception}\ \ \mathrm{e})\ \{...\}\rangle\phi}{\vdash\ \langle \textbf{try}\ \{\ st_{\mathrm{Sk}}(\mathrm{t},\ \mathrm{d};\ \textbf{throw}\ \mathrm{t});\ \}\ \textbf{catch}\ (\textbf{Exception}\ \ \mathrm{e})\ \{...\}\rangle\phi}$$

6 Main Result

To show that tac is derivable, which is by Sect. 4 equivalent to the derivability of all instances of $M(tac)$, we assume that there is a closed proof H of $M_{\mathrm{Sk}}(tac)$ using the sequent calculus for *JavaCardDL* (extended by the skolem symbols and the decomposition taclets of Sect. 5). It is possible to transform H into a proof H_ϕ for each instance ϕ of $M(tac)$:

Theorem 6.1 (Main Result) *Suppose that a proof H of $M_{\mathrm{Sk}}(tac)$ exists. Then for each instance $\phi = \kappa(M(tac))$ of the meaning formula $M(tac)$ there is a proof H_ϕ.*

In the following we will sketch a proof of Theorem 6.1. Due to lack of space we skip most of the details of the proof; a more detailed account can be found in [**?**].

The proof obligation $M_{\mathrm{Sk}}(tac) = \iota_{\mathrm{Sk}}(M(tac))$ differs from other instances $\phi = \kappa(M(tac))$ of the meaning formula in the instantiation of schema variables for terms, formulas and statements: In $M_{\mathrm{Sk}}(tac)$ such variables are replaced with skolem symbols as introduced in Sect. 5.1.[12] Hence it is possible

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

to obtain a "proof" H' of ϕ by replacing each occurrence of a skolem symbol $s_{\mathrm{Sk}}(\ldots) = \iota_{\mathrm{Sk}}(sv)$ in H with the instantiation $\kappa(sv)$ 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 invalid 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 substitutions: We assume that a mapping σ of the skolem symbols

$$\mathit{Sym}_{\mathrm{Sk}} := \mathit{Func}_{\mathrm{Sk}} \cup \mathit{Pred}_{\mathrm{Sk}} \cup \mathit{Statement}_{\mathrm{Sk}}$$

to terms, formulas and *Java* statements "with holes" is given. Namely, we allow that for a symbol s_{Sk} with signature $\alpha(s_{\mathrm{Sk}}) = ([S,]S_1, \ldots, S_n, T_1, \ldots, T_k)$ (or $\alpha(s_{\mathrm{Sk}}) = (T_1, \ldots, T_k, m)$ for statement symbols), the value $\sigma(s_{\mathrm{Sk}})$ contains a number of holes $\circ_i$ labelled with natural numbers $i \in \{1, \ldots, n+k\}$ (or $i \in \{1, \ldots, k+m\}$, resp.).

Example 6.2 *For a predicate skolem symbol $p_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}}$, an example of a substitution is given by the following mapping:*

$$\sigma(p_{\mathrm{Sk}}) = r(\circ_2, a) \land q(\circ_1) \land \langle \circ_2{=}1; \rangle \phi \qquad \text{for } p_{\mathrm{Sk}} \in \mathit{Pred}_{\mathrm{Sk}},\ \ \alpha(p_{\mathrm{Sk}}) = (S, \mathbf{int}).$$

The mapping σ is continued 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]

$$\sigma(s_{\mathrm{Sk}}(r_1, \ldots, r_l)) := \{\circ_1/r_1, \ldots, \circ_l/r_l\}\big(\sigma(s_{\mathrm{Sk}})\big)$$

Example 6.3 (Example 6.2 continued) *The mapping σ is applied in the following way to a formula containing the symbol p_{Sk}:*

$$\sigma\big(\forall x. p_{\mathrm{Sk}}(x; \mathrm{i})\big) = \forall x. \big(r(\mathrm{i}, a) \land q(x) \land \langle \mathrm{i}{=}1; \rangle \sigma(\phi)\big)$$

6.1 Treatment of Taclets

The most important observation to prove Theorem 6.1 is the following lemma:

Lemma 6.4 (Lifting of Taclet Applications) *Suppose that $R_{tac'}$ 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). If an instance of $R_{tac'}$ is given by*

$$\frac{P_1 \quad \cdots \quad P_n}{Q}$$

[13] Extensive considerations about possible collisions are omitted in this document; see [?] for details.

and σ is a substitution of skolem symbols, then there is a proof tree with root sequent $\sigma(Q)$, whose open goals are exactly the sequents $\sigma(P_1), \dots, \sigma(P_n)$.

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

$$\frac{\sigma(P_1) \quad \cdots \quad \sigma(P_n)}{\sigma(Q)}$$

is an instance of $R_{tac'}$.

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 $\sigma(t)$ in $\sigma(Q)$ (like in example 6.3, where a single occurrence 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 $\sigma(t) \doteq \sigma(t)$ and apply tac' to one side of the equation. Afterwards the equations $\sigma(t) \doteq \sigma(t_i)$ can be used to replace all occurrences of $\sigma(t)$ successively. This is illustrated by the following proof tree fragment, in which we use the notation $(\Gamma \vdash \Delta) = \sigma(Q)$:

$$\dfrac{\dfrac{\begin{matrix}\sigma(P_1)\\ \vdots \\ \Gamma_1, \sigma(t) \doteq \sigma(t_1) \;\vdash\; \Delta_1\end{matrix} \quad \cdots \quad \begin{matrix}\sigma(P_n)\\ \vdots \\ \Gamma_n, \sigma(t) \doteq \sigma(t_n) \;\vdash\; \Delta_n\end{matrix}}{\Gamma, \sigma(t) \doteq \sigma(t) \;\vdash\; \Delta}\,tac' \qquad \dfrac{*}{\Gamma \;\vdash\; \Delta, \sigma(t) \doteq \sigma(t)}}{\Gamma \;\vdash\; \Delta}$$

□

Corollary 6.5 *Suppose that the proof H of $M_{\mathrm{Sk}}(tac) = \iota_{\mathrm{Sk}}(M(tac))$ only consists of applications of taclets tac', and that the concerned taclets tac' do not contain skolem symbols. Then for each instance $\phi = \kappa(M(tac))$ of the meaning formula $M(tac)$ there is a proof H_ϕ.*

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, for a σ that substitutes skolem expressions $s_{\mathrm{Sk}}(\dots) = \iota_{\mathrm{Sk}}(sv)$ with the concrete instantiation $\kappa(sv)$, i.e. in a way such that $\sigma(M_{\mathrm{Sk}}(tac)) = \phi$. □

6.2 Treatment of Decomposition Rules

Lem. 6.4 of the last section is not directly applicable to applications of the taclets $D^{\diamond}_{s_{\mathrm{Sk}}}$, $D^{\Box}_{s_{\mathrm{Sk}}}$ (Sect. 5.3), as these taclets contain statement skolem symbols s_{Sk} and $Dec(s_{\mathrm{Sk}})$. 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 symbols s_{Sk} and $\mathit{Dec}(s_{\mathrm{Sk}})$ with the property that $\sigma(D^{\diamond}_{s_{\mathrm{Sk}}})$, $\sigma(D^{\Box}_{s_{\mathrm{Sk}}})$ are sound taclets, so that subsequently Lem. 6.4 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}, $\mathit{Dec}(s_{\mathrm{Sk}})$, such that*

(i) *$\sigma'(D^{\diamond}_{s_{\mathrm{Sk}}})$, $\sigma'(D^{\Box}_{s_{\mathrm{Sk}}})$ are sound taclets*

(ii) *There is a proof tree (fragment) whose root is $\vdash \sigma(\psi)$, such that the only goal left is $\vdash \sigma'(\psi)$.*

Referring to this lemma it is possible to formulate an analogue of Lem. 6.4 for decomposition taclets:

Lemma 6.7 (Lifting of Decompositions) *Suppose that R_D is a rule that is described by a decomposition taclet D ($D = D^{\diamond}_{s_{\mathrm{Sk}}}$ or $D = D^{\Box}_{s_{\mathrm{Sk}}}$). If an instance of R_D is given by*

$$\frac{P}{Q}$$

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

Proof. First the application of D is replaced with an application of the taclet $\sigma'(D)$, which is sound by Lem. 6.6, (i) (this substitutes certain occurrences of s_{Sk}, $\mathit{Dec}(s_{\mathrm{Sk}})$ within P and Q). Subsequently Lem. 6.4 can be applied to the resulting rule application w.r.t. σ'. □

Corollary 6.8 *Suppose that the proof H of $M_{\mathrm{Sk}}(\mathit{tac}) = \iota_{\mathrm{Sk}}(M(\mathit{tac}))$ only consists of applications of taclets tac' that do not contain skolem symbols, and of applications of decomposition taclets. Then for each instance $\phi = \kappa(M(\mathit{tac}))$ of the meaning formula $M(\mathit{tac})$ there is a proof H_ϕ.*

Proof. σ is chosen as in the proof of Cor. 6.5. By repeated application of Lem. 6.6, (ii) it is possible to construct a proof tree with root sequent $\vdash \phi$ and a single goal $\vdash \sigma'(M_{\mathrm{Sk}}(\mathit{tac}))$, with a substitution σ' that is chosen according to Lem. 6.6 for each skolem symbol s_{Sk} for statements.

It is then possible to construct a closed proof tree of $\vdash \sigma'(M_{\mathrm{Sk}}(\mathit{tac}))$ by transforming H: Each taclet application within H is replaced with the proof tree fragment that is obtained from Lem. 6.4 or Lem. 6.7 (according to the kind of the taclet). □

7 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. Even a greater class of possible *JavaCardDL* taclets is supported.

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' programs, and
- to explore further areas of application, as for example, proofs of program transformation 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*.

Acknowledgements

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.