# Using Taclets for the SMT-Integration of KeY

Benjamin Niedermann

2009/2010

## 1 Introduction

KeY's mightiness is among other things based on the large library of taclets which belong to the KeY-System. By means of these taclets KeY is able to show proof obligations containing JavaCard DL. Otherwise the KeY-System offers the possibility to hand proof obligations over to external provers like *Simplify*, *Z3* or *Yices*, to use their specialization in FOL. The external provers and KeY extend each other in a symbiotic way. To enhance this connection it suggests itself combining these two concepts more directly by using the taclet library also for the external provers. The idea is to translate the taclets into FOL and then to pass the translation over to the external provers as assumptions making concepts used in Key available for the external provers.
The process of passing the taclets over to externals provers is divided into two steps. First the taclets are translated into a KeY-formula to have a general format. Second the KeY-formula is translated into a special format used by the external prover. The second part is done by the SMT-Translation of KeY that is already implemented and applicable.
The main question to be answered is how the taclets could be translated into FOL. One possibility to achieve this goal is to translate a taclet into its meaning formula [1]. Meaning formulas contain the logical meaning of the taclet. They were introduced to give the possibility of showing the correctness of a given taclet.
For our purpose making the taclet library usable for external provers, we will take another direction. Instead of showing that the taclet is correct, we assume that its correctness has already been shown. We will only use the meaning formula to express the meaning of a taclet in FOL.
The presented translation mechanism of a taclet can in turn be divided into two main steps. First the taclet is translated into its meaning formula consisting only of a boolean skeleton. In doing so the formulas and terms of the taclet are assembled to one formula. The second step handles with the elimination of schema variables in the resulting meaning formula. Of course the success of the translation depends on the kind of schema variables occurring in the given taclet. It is not possible to translate every taclet into FOL, e.g. taclets that contain substitutions cannot be translated. But it transpired that especially taclets making statements about basic rules can be translated. Until now over 130 taclets are supported successfully.
In section 2 a short introduction to the translation of taclets into first order logic is given. Section 3 describes how to use the translation in KeY, while section 4 describes the implementation of the taclet translation. Finally, some examples are given for the convenient of the reader.

## 2 Translation of Taclets into Assumptions

This section describes how taclets can generally be translated into formulas of first order logic.

### 2.1 Preliminaries

Since this document is addressed to readers who are familiar with the KeY-System and its theoretical basics, only necessary definitions for comprehension are adduced. First we should say how taclets can be described formally. A taclet that is considered in this document has the following

shape:

$$\texttt{t1}\{\quad \texttt{\textbackslash assumes}(ifSeq)\ \texttt{\textbackslash find}(findSeq)\ \texttt{\textbackslash replacewith}(rw_1)\ \texttt{\textbackslash add}(add_1);\ \ldots\ \texttt{\textbackslash replacewith}(rw_k)\ \texttt{\textbackslash add}(add_k)\ \};$$

and can accordingly be defined as a tuple $(findSeq, ifSeq, rw_1, \ldots, rw_k, add_1, \ldots, add_k)$. The considered taclets does not contain any patterns of the type \addrule. Theoretically it would be possible to support \addrule by the translation but for the beginning this pattern is omitted to reduce the complexity.
To make a connection between taclets and validity we use the known concepts of rules and sequents [1]. A sequent $\Gamma \vdash \Delta$ is derivable in a sound and complete calculus if $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid.
Likewise in [1] we write $(\Gamma \vdash \Delta)^* := \bigwedge \Gamma \rightarrow \bigvee \Delta$. The union of two sequents is defined by $(\Gamma_1 \Rightarrow \Delta_1) \cup (\Gamma_1 \Rightarrow \Delta_1) := \Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2$.
A rule is defined as it is proposed in [2, Definition 3]:

**Definition 1.** *A rule $R$ is a binary relation between the set of all tuples of sequents and the set of all sequents. If $R(\langle P_1, \ldots, P_k \rangle, C)$ $(k \geq 0)$, then the conclusion $C$ is derivable from the premises $P_1, \ldots, P_k$.*

Normally we will write a rule as the rule schemata

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_k}{C}$$

and we will use the notation to present a taclet $t = (findSeq, ifSeq, rw_1, \ldots, rw_k, add_1, \ldots, add_k)$ as:

$$R_t = \frac{rw_1 \cup add_1 \cup ifSeq \ldots rw_k \cup add_k \cup ifSeq}{findSeq \cup ifSeq}$$

Consider the following example: The taclet

$$\texttt{boolean\_equal\_2}\{\quad \texttt{\textbackslash find}((b_1 = TRUE \leftrightarrow b_2 = TRUE))\ \texttt{\textbackslash inSequentState}\ \texttt{\textbackslash replacewith}(b_1 = b_2)\ \};$$

becomes the rule schemata

$$R_t = \frac{b_1 \doteq b_2}{b_1 \doteq TRUE \leftrightarrow b_2 \doteq TRUE}$$

To be able to talk about the soundness of a rule $R_t$ and its corresponding taclet $t$ we define:

**Definition 2.** *A rule $R$ is sound, if for each tuple $(P_1, \ldots, P_k, Q) \in R$ the following implication holds:*

$$\textit{if } P_1, \ldots, P_k \textit{ are valid then } Q \textit{ is valid.}$$

Taclets consist among other things of schema variables that must be instantiated while applying a taclet. Formally we introduce a map $\iota$ from term schema variables to concrete terms to describe such an instantiation. The instantiation $\iota$ is continued to arbitrary schema terms and formulas such the following equation holds:

$$\iota(op(t1, \ldots, t_n)) = \begin{cases} \iota(op) & \text{if } n = 0 \text{ and } op \text{ is schema variable} \\ op(\iota(t1), ..., \iota(tn)) & \text{otherwise} \end{cases}$$

For the formal definition see [2, Definition 13]. Obviously an instantiation $\iota$ is a homomorphism.

## 2.2 Translation of Taclets - Basic Structure

This section deals with the general translation of a taclet to a formula of first order logic. Although there are many different concepts within taclets like generic sorts, program schema variables or substitution, all of these taclets can be reduced to the same logical structure.

**Theorem 1.** *Every taclet* $t = (\mathit{findSeq}, \mathit{ifSeq}, rw_1, \ldots, rw_k, add_1, \ldots, add_k)$ *can be translated into a formula*

$$M(t) = \iota(\bigwedge_{i=1}^{k}(rw_i^* \vee add_i^* \vee \mathit{ifSeq}^*) \rightarrow (\mathit{findSeq}^* \vee \mathit{ifSeq}^*)),$$

*so that the taclet* $t$ *is sound if the the formula* $M(t)$ *is valid for all instantiations* $\iota$*.*

*Proof.* A taclet $t = (\mathit{findSeq}, \mathit{ifSeq}, rw_1, \ldots, rw_k, add_1, \ldots, add_k)$ is sound by definition if and only if its corresponding rule $R_t$ is sound. Let $(P_1, \ldots, P_k, Q)$ be a arbitrary tuple in $R_t$ where $P_i = \iota(rw_i \cup add_i \cup \mathit{ifSeq})$ and $Q = \iota(\mathit{findSeq} \cup \mathit{ifSeq})$, then $R_t$ is sound if and only if the following implication holds:

$$P_1, \ldots, P_k \text{ are valid then } Q \text{ is valid.} \tag{1}$$

A single sequent $P_i$ $(1 \leq i \leq k)$ is valid if $(P_i)^*$ is valid. This means since the deduction theorem the implication holds if

$$P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^* \tag{2}$$

is valid.

$$P_1^* \wedge \ldots \wedge P_k^* \rightarrow Q^* = \bigwedge_{i=1}^{k} \iota(rw_i \cup add_i \cup \mathit{ifSeq})^* \rightarrow \iota(\mathit{findSeq} \cup \mathit{ifSeq})^* \tag{3}$$

Now we have to pull out the $\iota$ over the propositional operators. The fact that $\iota$ is a homomorphism allows us to proceed this step in both directions and leads us therefore to the equivalent formula:

$$\iota(\bigwedge_{i=1}^{k}(rw_i \cup add_i \cup \mathit{ifSeq})^* \rightarrow (\mathit{findSeq} \cup \mathit{ifSeq})^*) \tag{4}$$

It is easy to see that $(P \cup Q)^* \equiv (P^* \vee Q^*)$. Thus we get the following equivalent formula:

$$\iota(\bigwedge_{i=1}^{k}((rw_i^* \vee add_i^* \vee \mathit{ifSeq}^*) \rightarrow (\mathit{findSeq}^* \vee \mathit{ifSeq}^*)) \tag{5}$$

□

After deriving the basic formula we must consider on the one hand how to translate sequents and on the other hand how to deal with missing expressions. A sequent $\Gamma \vdash \Delta$ is translated into $\bigwedge \Gamma \rightarrow \bigvee \Delta$. If a sequent is missing the corresponding translation is *false*. The only exception is a missing rewrite sequent $rw_i$. In this case we use the translation of the find sequent $findSeq$ instead of the translation of the rewrite sequent $rw_i$.
Most of the taclets have schema variables, therefore this problem is discussed in general at this point. For more detailed information see 4.2. Let $t$ be a taclet and $M(t)$ the meaning formula of $t$, containing the schema variable $SV$. The KeY System distinguishes, amongst other things, between 3 different types as follows:

1. *term schema variables*: The schema variable can be translated into a universal quantified logical variable of the same sort as $SV$ has.

2. $formula\ schema\ variables$ are not supported yet.

3. *program schema variables*: Program schema variables could be translated in the same way as term schema variables, but in many taclets the program schema variables are attended by attribute terms, that are instantiated with concrete values by the translation (see 4.2.4). To reduce the complexity of the resulting formula the same is done to program schema variables. Consequently the translation can be generally described as an instantiation of these schema variables with concrete values. At first sight this approach does not seem feasible because the number of the high number of possible instantiations, but it turned out that the external provers can handle this in many cases. For detailed information and a discussion see 4.2.4 and 4.2.6.

## 2.3 Rewriting Taclets

*Rewriting taclets* have the same shape as the taclets introduced in 2.1 with the difference that the $find$-Pattern can be only a formula or term:

t2{ \assumes($assum$) \find($find$)
\replacewith($rw_1$) \add($add_1$);
...
\replacewith($rw_k$) \add($add_k$) };

The idea for the translation is to re-use the meaning formula (1) by reducing the taclet to a non-rewriting taclet. The detailed reduction can be found in [1, page 234, section 4.5.3]. The result are two meaning formulas:

| **Type of** $find$ | **translation** |
|---|---|
| term | $\bigwedge_{i=1}^{k}(find \doteq rw_i \rightarrow add_i^*) \rightarrow assum^*$ |
| formula | $\bigwedge_{i=1}^{k}((find \leftrightarrow rw_i) \rightarrow add_i^*) \rightarrow assum^*$ |

## 2.4 Correctness

The derivation of the meaning formula shows that if the meaning formula is valid, then the corresponding taclet is sound. The opposite direction that would be necessary for our purposes is not true in general. The reason is that we have to use the deduction theorem to obtain the formula (2) from the sentence (1). Consider this example: Assume that $P$ is the sequent $\vdash p(c)$ and $Q$ is the sequent $\vdash \forall x.p(x)$ where $c$ is a constant and $p$ a predicate, then we obtain the following statements by applying $P$ and $Q$ to (1) respectively (2):

$$\text{If } p(c) \text{ is valid then } \forall x.p(x) \text{ is valid.} \tag{6}$$

$$p(c) \rightarrow \forall x.p(x) \tag{7}$$

Obvious (6) is a valid statement because $p(c)$ is not a true expression in every FOL-model. On the other hand (7) is not valid. This arises the problem that we cannot use the meaning formulas for every taclet. To motivate why we can use nevertheless the meaning formulas to achieve our goals we have to consider the reason why they were introduced: The idea was to design a formula that on the one hand encodes the meaning of a taclet and on the other hand is valid for as many taclets as possible. Therefore the probability that we can show the validness of a meaning formula of a concrete taclet should not be underestimated. Consequently the idea is to show for every supported taclet that its corresponding meaning formula is valid. Because the taclet is translated into a KeY formula first it is obvious using KeY to show the validness. The KeY System offers the user the possibility to save the resulting assumptions to a KeY problem file that can be proven afterwards. Remark: Until now it has never occurred that the resulting KeY problem file cannot be proved, i.d. there have not been any wrong translations of taclets.

# 3 Using the Taclet Translation in KeY

This sections describes briefly how to use the taclet translation within the KeY system.

## 3.1 Options

The settings dialog for the taclet translation ("Options|Decision Procedures") offers several options to change the behavior of the taclet translation:

1. 'Taclets|Selection': In this menu the user can select the taclets that should be translated into assumptions. This selection influences the behavior of the used external provers significantly. If the user chooses too many taclets the external prover can be overwhelmed, therefore the user should nearly know which taclets are necessary for closing the proof. The recommended approach is to start the external prover with different taclet selections.

2. 'Taclets': In this tab the user can determine the number of different generic sorts that are allowed within a taclet. Most of the taclet contain two or less different generic sorts. You should use a minimum of different generic sorts! The actual number you have to choose depends on the taclets that you want to translate.
   Furthermore you can store the assumptions made of the selected taclets to a problem file of KeY. This options offer the user the possibility to prove the assumptions with KeY.

# 4 Implementation

This section describes the structure of the implementation.

## 4.1 Structure

The process of passing the taclets over to external provers is divided into two steps. First the taclets will be translated into KeY-formula to have a general format. Second the KeY-formula will be translated into a special format of a external prover that is used. The second part is done by the SMT-Translation that is yet implemented and applicable. To achieve this the class structure in figure 4.1 is implemented. Only important classes are shown.

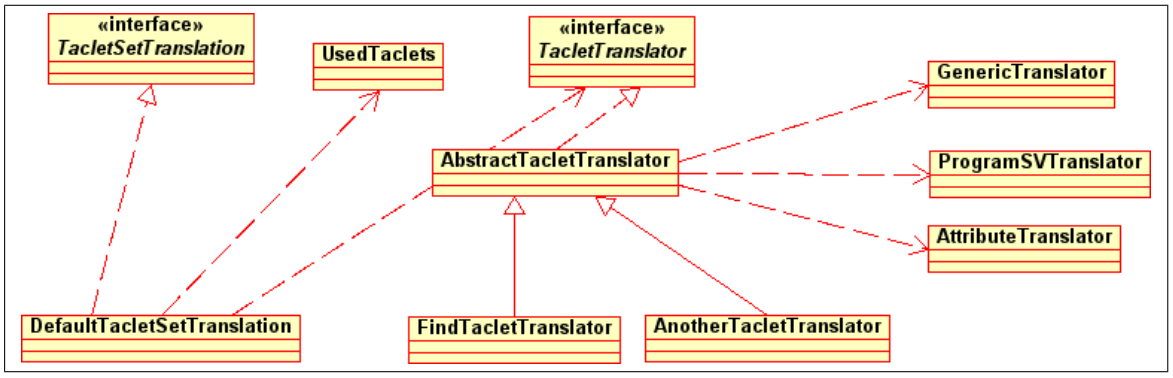


Abbildung 1: class diagram

- Class *UsedTaclets*: Within this class the taclets are encoded that are supported by the taclet translation. If you want to add further taclets change this class. Beware that you only add taclets that can be translated correctly by the translation mechanism. The user is not

allowed to change the list of supported taclets while runtime to ensure the correctness of KeY. The correctness of KeY should not depend on the user.

- Interface *TacletSetTranslation*: Interface to the SMT-Translation. Classes that implement this interface are used to translate a set of taclets.
- Class *DefaultTacletSetTranslation*: Implements the interface *TacletSetTranslation*. This class translates sequentially the given taclets by means of *TacletTranslator*.
- Interface *TacletTranslator*: Classes that implement this interface are used to translate a single taclet into a FOL-formula. The classes should be stateless.
- Class *AbstractTacletTranslator*: This class implements the interface *TacletTranslator* and provides the *translation pipeline* (4.2). Subclasses can override methods of this class to change the single functionality of the pipeline. *AbstractTacletTranslator* make use of the classes *AttributeTranslator*, *ProgramSVTranslator* and *GenericTranslator* to source functionality out. Furthermore *AbstractTacletTranslator* contains static methods that are used by different classes.
- Class *FindTacletTranslator*: Extends *AbstractTacletTranslation* and translates the basic structure of a given taclet.
- Class *GenericTranslator*: Translates the generic sorts within a taclet, by instantiating them with concrete sorts that are collected from the current sequent.
- Class *AttributeTranslator*: Translates program schema variables that refer to attribute terms.

## 4.2 The Translation Pipeline

The translation of a taclet is fragmented into several steps. The description of this steps is geared to the current implementation of the taclet translation and explains how the special concepts like generic sorts, schema variables and attribute operators are translated.

### 4.2.1 Check the Taclet.

First of all the taclet is checked, this means the procedure decomposes the given taclet and analysis its basic structure and conditions:

| Supported variable conditions: | Supported operators: |
|---|---|
| *TypeComparisionCondition*, *TypeCondition*, *AbstractOrInterfaceType*, *ArrayComponentTypeCondition* | *Junctor*, *Equality*, *Quantifier*, *RigidFunction*, *IfThenElse*, *TermSV*, *FormulaSV*, *VariableSV*, *MetaNextToCreate*, *NonRigidHeapDependentFunction*, *AttributeOp*, *MetaCreated*, *ProgramSV*, *ArrayOp* |

### 4.2.2 Generate the Basic Structure of the Taclet.

For generating the basic structure see 2.2.

### 4.2.3 Exchange Schema Variables for Logic Variables.

The resulting term is decomposed recursively. Every *term schema variable* is replaced by a free and unique logical variable. In this case unique means that schema variables that are equal are nevertheless replaced by the same variable but different schema variables not.

#### 4.2.4 Replace Attribute Terms.

Some taclets make a statement about the relationship between an object and its attributes where the object and its attributes are only named by schema variables.
Example:

only_object_are_referenced {
\assumes($obj.\langle created \rangle \doteq TRUE,\ inReachableState \vdash$)
\find($obj.attribute$)
\varcond(\isReference(\typeof($attribute$)))
\add($obj.attribute.\langle created \rangle \doteq TRUE \lor obj.attribute \doteq null \vdash$)}

If you want to translate such a taclet $t$ into a first order logic formula you cannot do so without knowing the context that belongs to the current sequent that should be proofed. At least you need to know which Java classes are involved in the current proof obligation. The simplest but not very efficient way would be the following: For every Java class $C$ that belongs to the current proof obligation instantiate $t$ with all possibilities that can be derived from $C$ in respect of $t$. Assume for example that we want to proof the sequent:

$$S : inReachableState \land o@C.\langle created \rangle \doteq TRUE \vdash o.a.\langle created \rangle \doteq TRUE$$

Further the only Java class that belongs to this proof obligation is $C$ with the attributes $a$ and $b$ that are of reference types. The translation would be:

1. $inReachableState \land created(obj) \doteq TRUE) \rightarrow$
   $\forall\ T\ obj.((created(a(obj)) \doteq TRUE\ \lor a(obj) \doteq null)$
2. $inReachableState \land created(obj) \doteq TRUE) \rightarrow$
   $\forall\ T\ obj.((created(b(obj)) \doteq TRUE\ \lor b(obj) \doteq null)$

Since we cannot quantify over attributes we must instantiate every taclet with all possible combinations of attributes that belong to the used classes. Accordingly we get two instantiations in this example. Obviously to show the validity of sequent $S$ we only need the first instantiation while the second instantiation is unnecessary. Consequently the idea is that we do not need every attribute of $C$ for the instantiation, but that the attributes that occur in $S$ could suffice. There is no guarantee that every sequent can be proved by only considering attributes that occur in $S$, but this is a good trade off between efficiency and power of the taclet translation. Therefore the implementation collects first all attribute terms of $S$ and then uses this set for the instantiation of $t$.
Especially this part of the translation can easily become a bottle neck: Not only the number of attributes that belongs to a single class and the number of classes increase the number of possible instantiation of a taclet $t$, but also the kind of statement that $t$ makes can have a huge influence on the possible number of instantiations. There are taclets whose translation depend not only on one but on two or more Java classes, e.g. taclets that make statements about the inheritance relationship between Java classes. For this case no optimization was implemented so far.
To reduce the complexity of the formula the quantifiers are omitted by the same idea: Instead of quantifying over all possible objects, concrete objects are collected from the sequent $S$ for the instantiation (See the third example in 5).
As it can been seen in the example above, the implementation must respect the *variable conditions* that belong to the taclet $t$.

#### 4.2.5 Quantify over all Logical Variables

To get general statements all logical variables must be universally quantified. This is done in this step.

### 4.2.6 Instantiate Generic Sorts

Many taclets make use of generic sorts. Likewise attribute terms generic sorts cannot be handled by SMT-provers directly. The approach to solve this problem is the same as described in 4.2.4. First all concrete sorts contained in the given sequent $S$ are collected. Then the terms given by 4.2.5 are instantiated with these sorts. While doing so the translation procedure must handle with variable conditions that restrict the instantiation of generic sorts.
Example:

`disjoint_repositories` {
$\texttt{\find}(G ::< get > (idx0) = H ::< get > (idx1))$
$\texttt{\varcond}(\texttt{\not\same}(G, H))$
$\texttt{\replacewith}(false)$}

The example make use of the two generic sorts $G$ and $H$. As described by `\varcond` the two generic sorts must not be instantiated by the same concrete sort.
Beware of the fact that the number of resulting terms increases exponentially according to the number of different generic sorts. Consequently there are $n^2$ ($n$: number of concrete sorts) possible instantiations for the taclet $disjoint\_repositories$. The good news is that there is only a fistful of taclets that contain more than two different generic sorts. None of the supported taclets has more than four different generic sorts.

### 4.2.7 Last check

The last check is done to guarantee that there is not a taclet that contains schema variables that have not been instantiated yet. This happens for example when the considered proofs state does not contain enough attribute terms or concrete sorts that match.

# 5 Examples

**1. example:**

schema variables:
$\texttt{\term}\ boolean\ b_1$
$\texttt{\term}\ boolean\ b_2$

taclet:
`boolean_equal_2`{ $\texttt{\find}((b_1 \doteq TRUE \leftrightarrow b_2 = TRUE))$ `\inSequentState`
$\texttt{\replacewith}(b_1 \doteq b_2)$ };

translation:
$boolean\_equal\_2^* \equiv$ $\forall\ boolean\ b_1'.\ \forall\ boolean\ b_2'.(((b_1' \doteq TRUE \leftrightarrow b_2' \doteq TRUE) \leftrightarrow b_1' \doteq b_2') \rightarrow false \rightarrow false)$

**2. example:**

schema variables:
$\texttt{\term}$ *boolean* $b$

taclet:

| | |
|---|---|
| `apply_eq_boolean_rigid` { | $\texttt{\assumes}(\vdash b \doteq TRUE)\texttt{\find}(b)\ \texttt{\inSequentState}$ |
| | $\texttt{\replacewith}(FALSE)$ }; |

translation:

| | |
|---|---|
| $apply\_eq\_boolean\_rigid^* \equiv$ | $\forall\ boolean\ b'.(((b' \doteq FALSE) \rightarrow false) \rightarrow (\vdash b \doteq TRUE)^*)$ |
| $(\vdash b \doteq TRUE)^* =$ | $b' \doteq true$ |

**3. example**
Assume that we want to show the following sequent $S$:

$$inReachableState \wedge o@C.\langle created\rangle \doteq TRUE \vdash o.a.\langle created\rangle \doteq TRUE$$

To prove this sequent the KeY-System would apply the following taclet:

`only_object_are_referenced` {
$\texttt{\assumes}(obj.\langle created\rangle \doteq TRUE,\ inReachableState \vdash)$
$\texttt{\find}(obj.attribute)$
$\texttt{\varcond}(\texttt{\isReference}(\texttt{\typeof}(attribute)))$
$\texttt{\add}(obj.attribute.\langle created\rangle \doteq TRUE \vee obj.attribute \doteq null \vdash)$}

where *obj* and *attribute* are program schema variables and $C$ is the following Java class:

```
class C{
        Object a;
        Object b;
}
```

The translation of this taclet is:

$$(inReachableState \wedge created(o) \doteq TRUE) \rightarrow (created(a(o)) \doteq TRUE \vee a(o) \doteq null)$$

# Literatur


[1] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[2] Bernhard Beckert and Martin Giese and Elmar Habermalz and Reiner Hähnle and Andreas Roth and Philipp Rümmer and Steffen Schlager. *Taclets: A New Paradigm for Constructing Interactive Theorem Provers.* Number 9–2004. 2004.