KeY Quicktour for JML

Work in progress

The KeY Team

This article is a variant of [BHS] by
Thomas Baar, Reiner Hähnle, and Steffen Schlager.

April 18, 2013

1 Introduction

When we started writing this document, we aimed at providing a short tutorial accompanying the reader at her/his first steps with the KeY system. The KeY-Tool is designed as an integrated environment for creating, analysing, and verifying software models and their implementation. The reader shall learn how to install and use the basic functionality of the KeY-Tool. Besides practical advises how to install and get KeY started, we show along a small project how to use the KeY-Tool to verify programs.

Verification means to prove that a program complies with its specification in a mathematical rigorous way. In order to fulfil this task, the specification needs to be given in a formal language with a precise defined meaning. In the current version of the document we focus on the popular *Java Modeling Language* (JML) [LPC$^+$11, LBR04] as specification language.

In the next sections we show how to verify a JML annotated (specified) JavaCard program. Therefore features KeY a calculus for the complete JavaCard language including advanced features like transactions.

Besides JML, the KeY-Tool supports JavaCardDL as specification languages.

For a longer discussion on the architecture, design philosophy, and theoretical underpinnings of the KeY-Tool please refer to [BHS07, ABB$^+$05]. The formalization of the heap has changed significantly with KeY 2.0. For the new heap model we refer to [Wei11].

In case of questions or comments don't hesitate to contact the KeY-support team at *support@key-project.org*.

1.1 Version Information

This tutorial was tested for KeY version 2.0.

1.2 Installation

You can choose between different methods to install and use KeY. We recommend for this tutorial the Java Web Start variant described in Sect. 1.2.1.

1.2.1 The KeY-Prover by Java Web Start

Java Web Start is a Java Technology which allows to start applications directly from a website. No installation is needed. You can visit our homepage

`http://www.key-project.org/download`

which contains a link to Java Web Start the KeY-Prover.

Please note that you have to have installed the Java Web Start facility (which should come along with your Java distribution).

1.2.2 Bytecode and Sourcecode Installation

The download site offers also the binary and source code version of KeY. If you intend to choose one of them, please note that you need to download several third party libraries. The required libraries are packaged in a single tar-gzipped archive that is also linked from our main download site `http://www.key-project.org/download`. Please follow the instructions given in the README files.

2 Tutorial Example

2.1 Scenario

The tutorial example is a small paycard application consisting of two packages `paycard` and `gui`. Package `paycard` contains all classes implementing the program logic and has no dependencies to the `gui` package.

Package `paycard` consists of the classes: `PayCard`, `LogFile` and `LogRecord`. The `gui` package contains `ChargeUI`, `IssueCardUI`, and the main class `Start`.

In order to compile the project change to the jml directory and execute the following command:

```
javac -sourcepath . gui/*.java
```
(use `gui\` under Windows)

Executing from the same directory

```
java -classpath . gui.Start
```
starts the application. Try this now[1].

The first dialog when executing the main method in `Start` asks the customer (i.e. the user of the application) to obtain a paycard. A paycard can be charged by the customer with a certain amount of money and thereafter used for cashless payment until the pre-loaded money is eaten up.

To prevent the risk for the customer when loosing the paycard, there is a limit up-to-which money can be loaded/charged on the paycard. Depending on the limit there are three paycard variants offered by the bank: a standard paycard with a limit of 1000, a junior paycard with a limit of 100, or a paycard with a user-defined limit. The initial balance of a newly issued paycard is zero.

In the second dialog coming up after obtaining a paycard, the customer may charge her/his paycard with a certain amount of money. But the charge operation is only successful if the current balance of the paycard plus the amount to charge is less than the limit of the paycard. Otherwise, i.e., if the current balance plus the amount to charge is greater or equal the limit of the paycard, the charge operation does not change the balance on the paycard and an attribute counting unsuccessful operations is increased.

The KeY-Tool aims to *formally prove* that the implementation actually satisfies such requirements. For example, one can formally verify the invariant that the balance on the paycard is always less than the limit of the paycard.

The intended semantics of some classes is specified with the help of invariants denoted in the Java Modeling Language (JML) [LPC$^+$11, LBR04]. Likewise, the behavior of most methods is described in form of pre-/postconditions in the JML. We do not go into details on *how* JML specifications for Java classes are created. The tools downloadable from `http://jmlspecs.org/download.shtml` may be helpful here. In particular, **we require and assume that all JML specifications are complying to the JML standards** [LPC$^+$11]. KeY's JML front-end is no substitute for the JML parser / type checker.

[1] potentially arising warnings can be safely ignored here

2.2 A First Look on the JML Specification

Before we can verify that the program satisfies the property mentioned in the previous section, we need to express it in JML. The remaining section tries to give a short, intuitive impression on how such a specification looks like. In Sect. 3 JML specifications are explained in a bit more depth.

Load the file `paycard/PayCard.java` in an editor of your choice and search for method `isValid`. You should see something like

```
/*@
  @ public normal_behavior
  @ requires true;
  @ ensures \result == (unsuccessfulOperations<=3);
  @ assignable \nothing;
  @*/
public /*@pure@*/ boolean isValid() {
   if (unsuccessfulOperations<=3) {
      return true;
   } else {
      return false;
   }
}
```

JML specifications are annotated as special marked comments[2] in Java files. Comment containing JML annotations have an 'at' sign directly after the comment sign as start marker and multi-line comments also as end-marker.

The JML annotation in the above listing represents a JML method contract. A contract states that when the caller of a method ensures that certain conditions (precondition + certain invariants (see Sect. 4)) then the method ensures that after the execution the postcondition holds[3].

The precondition is `true`. This means the precondition does not place additional[4] conditions the caller has to fulfill in order to be guaranteed that after the execution of the method its postcondition holds.

The `ensures` clause specifies the method's postcondition and states simply that the return value of the method is `true` if and only if there have not been more than 3 unsuccessful operations. Otherwise `false` is returned.

For the other parts of the method specification see Sect. 4.

[2]It is also possible to have them in a separate file (not yet supported by KeY).
[3]The complete semantics is more complex; see Sect. 4 and [LPC$^+$11].
[4]There might be conditions stemming from invariants.

3 How to Verify JML Specifications with the KeY-Tool

JML specifications, in particular pre- and postconditions, can be seen as abstractions of an implementation. In this context, an implementation is called *correct* if it actually implies properties expressed in its specification. The KeY-Tool includes functionality to *verify* the correctness of an implementation with respect to its specification.

In this section we describe how to start (Sect. 3.1) the KeY-Prover and load the tutorial example (Sect. 3.2) as well as a short overview about the graphical user interface and its options (Sect. 3.3). Last but not least, we explain how to configure the KeY-Prover to follow the tutorial example (Sect. 3.4).

3.1 Starting the KeY-Prover

In order to verify a program, you first need to start the KeY prover. This is done either by using the webstart mechanism (see Sect. 1.2.1) or by calling the `runProver` script of your KeY distribution[5], e.g., by running

```
bin/runProver
```

3.2 Loading the Tutorial Example

After downloading and unpacking this quicktour you should find a directory `jml` containing the two subdirectories `paycard` and `gui`. We refer to the directory `jml` as top-level directory.

1. You have to choose the Java source files you want to verify. They contain both the source code and the JML annotations. You can do this by either

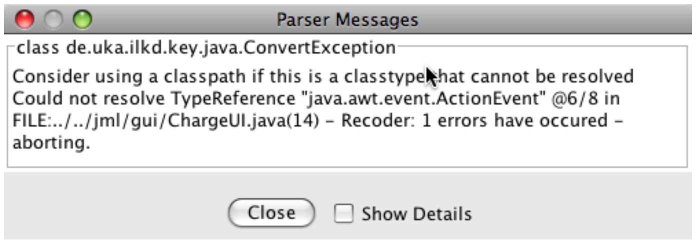
   - adding on the command line the path to the `paycard` directory:

     ```
     bin/runProver <path_to_quicktour>/jml/paycard
     ```

     (Windows: replace '/' by '\')
   - opening **File** | **Load** and selecting the `paycard` package directory after having started `runProver` without any arguments.

   KeY will then load the tutorial example and parse the JML annotations. *If you get an error dialog similar to the one in Fig. 1 than you have selected the `jml` directory instead of its subdirectory `paycard`.*

   If you have your own projects you want to verify you can proceed similarly. Please note, that KeY supports by default only a very limited selection of the standard library classes (the complete list can be found in [Red]), how to extend them and how to configure more complex projects that use 3rd party libraries is described in brief in App. D.


Parser Messages

class de.uka.ilkd.key.java.ConvertException

```
Consider using a classpath if this is a classtype that cannot be resolved
Could not resolve TypeReference "java.awt.event.ActionEvent" @6/8 in
FILE:../../jml/gui/ChargeUI.java(14) – Recoder: 1 errors have occured –
aborting.
```

Figure 1: Error dialog complaining about an unknown type

2. Now the Proof Management window should appear as shown in Fig. 2(a).

   In the left part of the window title **Contract Targets**, the Proof Management lists all packages, classes/interfaces and methods of the project to be verified in a tree structure similar to standard file managers.

   The browser allows you to select the proof obligation (kind of property), you want to verify. Selecting method `charge` of class `PayCard` offers three contracts (Fig. 2(b))

   one **exceptional_behavior** and two **normal_behavior** contracts. Select that **normal_behavior** contract which in its **post**condition talks about `balance`, and confirm by pressing the button **OK**. More details about the contract configurator will be given in Sect. 4.

3. You should now see the KeY-Prover window with the loaded proof obligation as in Fig. 3. The prover is able to handle predicate logic as well as Dynamic Logic. The KeY-Prover was developed as a part of the KeY-Project and is implemented in JAVA. It features interactive application of proof rules as well as automatic application controlled by strategies. In the near future more powerful strategies will be available.

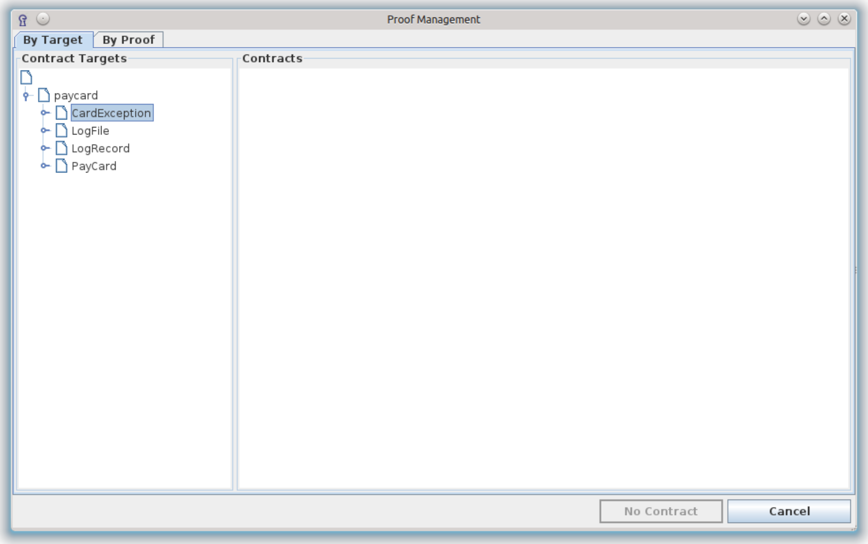
   In Sect. 4.3, we show how to prove some of the proof obligations generated for the tutorial example.

3.3 The KeY-Prover

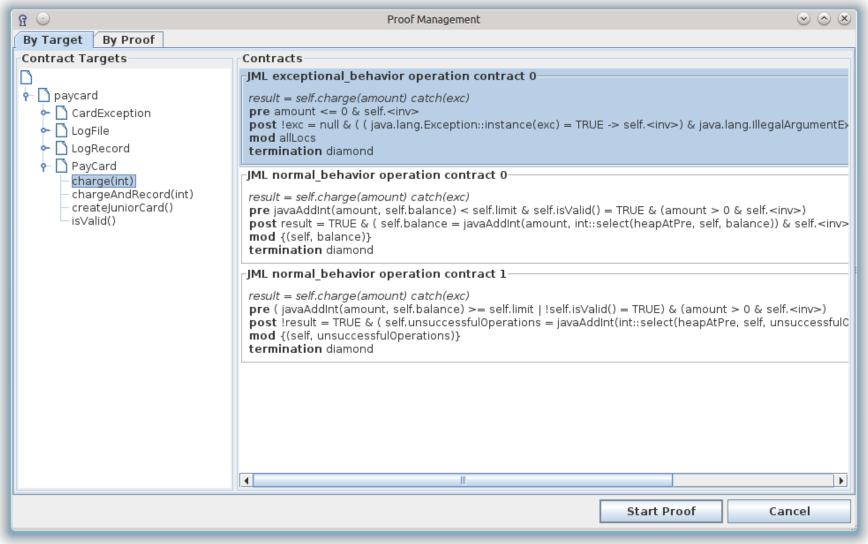
We assume that you have performed the steps described in the previous section and that you see now something similar to Fig. 3. In this section we describe the GUI of the KeY-Tool and its different components. Most of the components in the GUI are also labelled with a tooltip.

The KeY-Prover window consists of a menubar, a toolbar (all buttons explained in C) and three panes where the lower left pane is additionally tabbed. Each pane is described below.

[5]In this case we assume that you have installed the KeY-Tool as described in Sect. 1.2.2.



(a) Proof Obligation Browser after startup with expanded `paycard` package



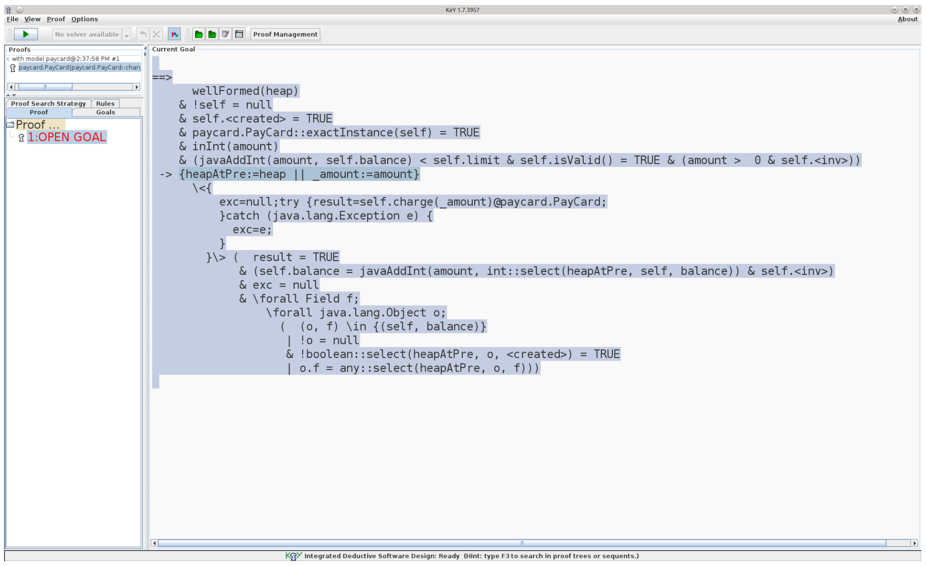
```
JML exceptional_behavior operation contract 0
result = self.charge(amount) catch(exc)
pre amount <= 0 & self.<inv>
post !exc = null & ( ( java.lang.Exception::instance(exc) = TRUE -> self.<inv>) & java.lang.IllegalArgumentE
mod allLocs
termination diamond

JML normal_behavior operation contract 0
result = self.charge(amount) catch(exc)
pre javaAddInt(amount, self.balance) < self.limit & self.isValid() = TRUE & (amount > 0 & self.<inv>)
post result = TRUE & ( self.balance = javaAddInt(amount, int::select(heapAtPre, self, balance)) & self.<inv>
mod {(self, balance)}
termination diamond

JML normal_behavior operation contract 1
result = self.charge(amount) catch(exc)
pre ( javaAddInt(amount, self.balance) >= self.limit | !self.isValid() = TRUE) & (amount > 0 & self.<inv>)
post !result = TRUE & ( self.unsuccessfulOperations = javaAddInt(int::select(heapAtPre, self, unsuccessfulC
mod {(self, unsuccessfulOperations)}
termination diamond
```

(b) Proof Obligation Browser listing proof obligations for method `charge` of class `PayCard`

Figure 2: The Proof Obligation Browser window


KeY 1.7.3957
File View Proof Options
No solver available
Proof Management
Proofs
with model paycard@2:37:58 PM #1
paycard.PayCard[paycard.PayCard::charg
Proof Search Strategy
Rules
Proof
Goals
Proof ...
1:OPEN GOAL
Current Goal
About
Integrated Deductive Software Design: Ready (Hint: type F3 to search in proof trees or sequents.)

```

==>
     wellFormed(heap)
   & !self = null
   & self.<created> = TRUE
   & paycard.PayCard::exactInstance(self) = TRUE
   & inInt(amount)
   & (javaAddInt(amount, self.balance) < self.limit & self.isValid() = TRUE & (amount >  0 & self.<inv>))
 -> {heapAtPre:=heap || _amount:=amount}
     \<{
         exc=null;try {result=self.charge(_amount)@paycard.PayCard;
         }catch (java.lang.Exception e) {
           exc=e;
         }
       }\> (  result = TRUE
            & (self.balance = javaAddInt(amount, int::select(heapAtPre, self, balance)) & self.<inv>)
            & exc = null
            & \forall Field f;
                \forall java.lang.Object o;
                  (  (o, f) \in {(self, balance)}
                   | !o = null
                   & !boolean::select(heapAtPre, o, <created>) = TRUE
                   | o.f = any::select(heapAtPre, o, f)))
```

Figure 3: The KeY-Prover with loaded contract for method `charge` of class `PayCard`

Upper left pane: Every problem you want to prove with the KeY-Prover is loaded in a proof environment. In this pane all currently loaded problems respectively their proof environments are listed.

Lower left pane: This pane contains the following five tabs.

> **Proof:** This pane (see Fig. 4(a)) contains the whole proof tree which represents the current proof. The nodes of the tree correspond to sequents (goals) at different proof stages. Click on a node to see the corresponding sequent and the rule that was applied on it in the following proof step (except the node is a leaf). Leaf nodes of an open proof branch are colored red whereas leaves of closed branches are colored green.
>
> Pushing the right mouse button on a node of the proof tree will open a pop-up context menu. If you choose now *Prune Proof*, the proof tree will be cut off at this node, so all nodes lying below will be deleted. Choosing *Apply Strategy* will start an automatic proof search (see later *Automatic Proving*), but only on that branch which the node you had clicked on belongs to. Reverting a proof step is possible by choosing *Goal Back* in the pop-up context menu.
>
> The context menu also contains commands that allow to hide closed subtrees, to blind out inner nodes, to collapse, or expand the tree. The commands help to keep track of a proof.

Goals: In this pane the open goals of a certain proof (corresponding to one entry in the upper left pane) are listed. To work on a certain goal just click on it and the selected sequent will be shown in the right pane.

Rules: In this pane (Fig. 4(b)), all the rules available in the system are indicated. KeY distinguishes between *axiomatic taclets* (rules that are always true in the given logic), *lemmas* (that are derived from and thus provable by axiomatic taclets) and *built-in rules* (for example how certain expressions can be simplified).

By clicking on a rule of the list, the description of that rule is shown in the box below the rule list.

Proof Search Strategy: This tab (see Fig. 4(c)) allows you to define the active strategy from a set of available strategies. There are several parameters and only the most important ones will be covered here, a complete list can be found in B:

Max. Rule Applications You can set the number N_{aut} of automatic rule applications using the slider. Even if the automatic strategy can still apply rules after N_{aut} applications, automatic proving stops.

Stop At Choose when strategy execution shall stop. Possible values are `Default`: strategy stops when no rules are applicable or the maximal number of steps is reached and `Unclosable`: strategy stops in all situations when `Default` stops but also already when the first goal is encountered on which no further rule is (automatically) applicable.

Proof splitting Influences usage of rules branching a proof tree. Only rules working on formulas not on programs fall under the chosen policy, i.e., program rules causing splits are still applied even if splitting is switched off. The values are `free` (withour restrictions), `Delayed` (allows still splitting but prefers other rules) and `Off` (no splitting).

Loop treatment This allows to set up how while-loops are treated. They can be left untouched (`None`), handled using stated invariant contracts (`Invariant`), or repeatedly unrolled (`Expand`).

Method treatment Methods can also be left untouched (`None`), have their method contracts applied (`Contracts`), or be inlined, i.e. have the method body expanded in place (`Expand`).

Dependency contract For the simplification of heap terms setting this option to `On` the information in JML's `accessible` clause is used.

Arithmetic treatment The KeY-Prover has several options for the treatment of arithmetic expressions:

Basic: Using this option, polynomial expressions are simplified. In the antecedent Gröbner Bases are computed polynomials. Linear inequations are handled using (partial) Omega procedures.

DefOps: Using the option DefOps, mathematical symbols such as:
`/`, `%`, `jdiv`, `jmod`, range predicates, such as `int_RANGE`, `short_MIN` and symbols for mathematical operations on integers with a certain semantic such as `addJint` or `mulJshort`, are expanded. This means for example constants, such as `short_MIN`, are replaced by their concrete values (in this case -32768) and range predicates, such as `inInt` are replaced by their ranges (in this case $i \leq int\_MAX \wedge int\_MIN \leq i$).

Model Search: Setting the model search option, the KeY-Prover supports non-linear equations and model search. Additionally multiplication of inequations with each other and systematic case distinctions (cuts) can be performed. This method is guaranteed to find counterexamples for invalid goals that only contain polynomial (in)equations. Such counterexamples turn up as trivially unprovable goals. It is also able to prove many more valid goals involving (in)equations, but will in general not terminate on such goals.

Quantifier treatment Sometimes quantifiers within the sequent have to be instantiated. This can be either done manually (`None`) or automatically with different alternatives:

No Splits Instantiate a quantifier only if this will not cause the proof to split.

Unrestricted Instantiates a quantifier even when causing splits. However the startegy tries to predict the number of caused open branches and will prefer those with no or only few splits.

No Splits with Progs Chooses between the `No Splits` and `Unrestricted` behaviour depending on programs present in the sequent. If a program is still present the `No splits` behaviour is used. Otherwise quantifiers are instantiated unrestricted

Right pane: In this pane you can either inspect inner, already processed, nodes of the proof tree or you can continue the proof by applying rules to the open goals, whichever you choose in the left pane.

Rules can be applied either interactively or non-interactively using strategies:

Interactive Proving: By moving the mouse over the current goal you will notice that a subterm of the goal is highlighted (henceforth called the *focus term*). Pressing the left mouse button displays a list of all proof rules currently applicable to the focus term.

A proof rule is applied to the focus term simply by selecting one of the applicable rules and pressing the left mouse button. The effect is that a new goal is generated. By pushing the button *Goal Back* in the main window of the KeY-Prover it is possible to undo one or several rule applications. Note, that it is currently not possible to backtrack from an already closed goal.

Automatic Proving: Automatic proof search is performed applying so-called strategies which can be seen as a collection of rules suited for a certain task. To determine which strategy should be used select the tab item *Proof Search Strategy* in the left pane as described above.

To start (respectively continue) the proof push the *run strategy*-button on the toolbar labelled with the $\triangleright$ - symbol.

3.4 Configure the KeY-Prover

In this section we explain how to configure the KeY-Prover to follow the tutorial and give a few explanations about the implications of the chosen options. Most of the options are accessible via the KeY-Prover menu. An exhaustive list is available as part of Appendix A. In order to verify or change some of the necessary options it is necessary to have a proof obligation loaded into the KeY-Prover as described in Sect. 3.2.

The menu bar consists of different pull down menus:

File menu for file related actions like loading and saving of problems resp. proofs, or opening the **Proof Management** window

View menu for changing the look of the KeY-Prover

Proof menu for changing and viewing proof specific options

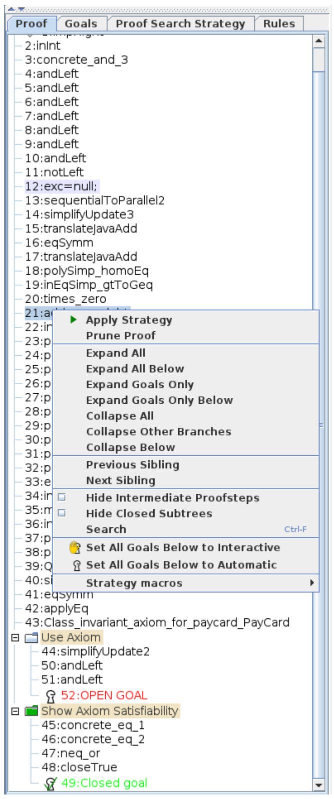
Options menu for configuring general options affecting any proof

About menu (as the name says)

KeY provides a complete calculus for the Java Card 2.2.x version including additional features like transactions. Further it provides some more concepts of real Java like class and object initialisation. This quicktour is meant to help with the first steps in the system.

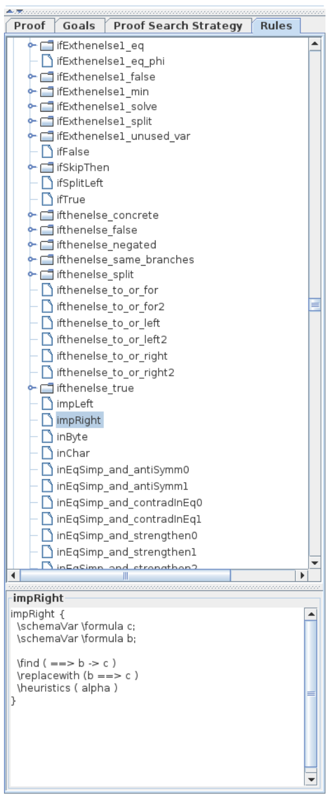
For simplicity, we deactivate some advanced concepts and configure the KeY-Prover to use the normal arithmetic integers to model Java integer types, which will avoid to deal with modulo arithmetics. *Important:* Please note that this configuration is unsound with respect to the Java semantics.

In order to configure the KeY-Prover in the mentioned way select **Options | Taclet Options**. The dialog shows a list of available options. Clicking on an option in the list also shows a short explanation for the option. The list



```
2:inInt
3:concrete_and_3
4:andLeft
5:andLeft
6:andLeft
7:andLeft
8:andLeft
9:andLeft
10:andLeft
11:notLeft
12:exc=null;
13:sequentialToParallel2
14:simplifyUpdate3
15:translateJavaAdd
16:eqSymm
17:translateJavaAdd
18:polySimp_homoEq
19:inEqSimp_gtToGeq
20:times_zero
21:a
22:ir
23:p
24:p
25:p
26:p
27:p
28:p
29:p
30:p
31:p
32:p
33:e
34:ir
35:m
36:ir
37:p
38:p
39:Q
40:si
41:eqSymm
42:applyEq
43:Class_invariant_axiom_for_paycard_PayCard
Use Axiom
    44:simplifyUpdate2
    50:andLeft
    51:andLeft
    52:OPEN GOAL
Show Axiom Satisfiability
    45:concrete_eq_1
    46:concrete_eq_2
    47:neq_or
    48:closeTrue
    49:Closed goal
```

(a) The Proof tree tab



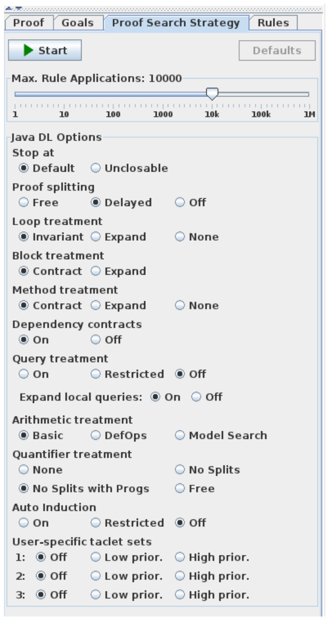
```
ifExthenelse1_eq
ifExthenelse1_eq_phi
ifExthenelse1_false
ifExthenelse1_min
ifExthenelse1_solve
ifExthenelse1_split
ifExthenelse1_unused_var
ifFalse
ifSkipThen
ifSplitLeft
ifTrue
ifthenelse_concrete
ifthenelse_false
ifthenelse_negated
ifthenelse_same_branches
ifthenelse_split
ifthenelse_to_or_for
ifthenelse_to_or_for2
ifthenelse_to_or_left
ifthenelse_to_or_left2
ifthenelse_to_or_right
ifthenelse_to_or_right2
ifthenelse_true
impLeft
impRight
inByte
inChar
inEqSimp_and_antiSymm0
inEqSimp_and_antiSymm1
inEqSimp_and_contradInEq0
inEqSimp_and_contradInEq1
inEqSimp_and_strengthen0
inEqSimp_and_strengthen1
inEqSimp_and_strengthen2
```

impRight

```
impRight {
 \schemaVar \formula c;
 \schemaVar \formula b;

 \find ( ==> b -> c )
 \replacewith (b ==> c )
 \heuristics ( alpha )
}
```

(b) The Rules tab



Max. Rule Applications: 10000

Java DL Options

Stop at
[✓ Default] Unclosable

Proof splitting
Free [✓ Delayed] Off

Loop treatment
[✓ Invariant] Expand None

Block treatment
[✓ Contract] Expand

Method treatment
[✓ Contract] Expand None

Dependency contracts
[✓ On] Off

Query treatment
On Restricted [✓ Off]

Expand local queries: [✓ On] Off

Arithmetic treatment
[✓ Basic] DefOps Model Search

Quantifier treatment
None No Splits
[✓ No Splits with Progs] Free

Auto Induction
On Restricted [✓ Off]

User-specific taclet sets
1: [✓ Off] Low prior. High prior.
2: [✓ Off] Low prior. High prior.
3: [✓ Off] Low prior. High prior.

(c) The Proof Search Strategy tab

Figure 4: Selected components of the KeY-Tool graphical user interface

below explains the options necessary for this tutorial[6]. Please ensure that for each option the value as given in parentheses directly after the option name is selected. In case you have to change one or more values, you will have to reload the tutorial example in order to activate them.

javacard: (jcOff) There are two values for this option jcOn and jcOff. Switching on or off all taclets axiomatising JavaCard specific features like transaction.

initialisation: (disableStaticInitialisation) Specifies whether static initialisation should be considered.

intRules: (arithmeticSemanticsIgnoringOF) Here you can choose between different semantics for Java integer arithmetic (for details see [Sch02, Sch07, BHS07]). Three choices are offered:

- **javaSemantics** (Java semantics): Corresponds exactly to the semantics defined in the Java language specification. In particular this means, that arithmetical operations may cause over-/underflow. This setting provides correctness but allows over-/underflows causing unwanted side-effects.
- **arithmeticSemanticsIgnoringOF** (Arithmetic semantics ignoring overflow): Treats the primitive finite Java types as if they had the same semantics as mathematical integers with infinite range. Thus this setting does not fulfil the correctness criteria.
- **arithmeticSemanticsCheckingOF** (Arithmetic semantics prohibiting overflow): Same as above but the result of arithmetical operations is not allowed to exceed the range of the Java type as defined in the language specification. This setting not only enforces the java semantics but also ascertains that no overflow occur.

Please activate *Minimize Interaction* in the `Options` menu in order reduce interaction with the system.

As a last preparation step change to the Proof Search Strategy tab in the lower left pane and choose the following setting:

- Max. Rule Applications should be set to a value greater or equal 500. A too low value will cause the prover to leave automatic mode too early. In this case you might have to press the run strategy button more often than described in the tutorial.
- Java DL must be selected with the following sub options:
  - Stop at: Default
  - Proof splitting: Delayed (Normal should also work)

[6]App. A contains a list of all available options.

- Loop treatment: Invariant
- Method treatment: Expand
- Query treatment: Expand
- Expand Local Queries: On
- Arithmetic treatment: Basic is sufficient for this tutorial (when using division, modulo or similar you will need at least DefOps)
- Quantifier treatment: No Splits with Progs is a reasonable choice for most of the time
- User-specific taclets: all Off

4 Provable properties

In the following the ideas behind the various options for verification are described informally. A formal description of the generated proof obligations is contained in [BHS07, Wei11]. For further details on the mapping between JML specifications and the formulae of the JavaDL logic used in KeY please consult [Eng05].

Examples of usage within the context of the case study in this tutorial are described in Sect. 4.3.

4.1 Informal Description of Proof Obligations

The current implementation of the KeY-Prover generates two kinds of proof obligations – functional contracts and dependency contracts[7]. Method contracts describe the behaviour of a method and properties covered by method contracts include:

- *properties for method specifications:* we show that a method *fulfils* its method contract,
- *properties for class/object specifications:* we show that a method *preserves* invariants of the object on which the call is invoked.

For the verification of programs, *dependency contracts* are used to be able to rely on the properties of, e.g. the object invariant, in parts of the proof where method calls are invoked or other instructions are performed which change the memory locations on the heap. If these method calls only change memory locations on the heap, which do not affect those location on which the invariant at most depends on, it is still possible to use the stated properties of the invariant after such a method call. However, if the method calls also change location on the heap on which the invariant depends on, it is not possible to rely on those properties anymore and it has to be proven that the invaraint still holds. For a more detailed description see [Wei11].

4.1.1 The Logic in Use

In this section we make a short excursion to the formalism underlying the KeY-Tool. As we follow a deduction based approach towards software verification, logics are the basic formalism used. More precise a typed first-order dynamic logic called JavaCardDL.

We do not intend here to give a formal introduction into the used logic, but we explain the intended meaning of the formulas. Further we assume that the reader has some basic knowledge if classical first-order logic.

In addition to classical first-order logic, dynamic logic knows two additional operators called modalities, namely the diamond $\langle\cdot\rangle\cdot$ and box $[\cdot]\cdot$ modality. Their

[7]For a detailed description of both contracts see [Wei11].

first argument takes a piece of JavaCard code and the second argument an arbitrary formula. Let p be a program and ϕ an arbitrary formula in JavaCardDL then

- $\langle p \rangle \phi$ is a formula in JavaCardDL, meaning, program p terminates **and** in its final state formula ϕ holds.
- $[p]\phi$ is a formula in JavaCardDL, meaning, **if** program p terminates **then** in its final state formula ϕ holds.

The notion *state* is a central one. Simplified a state can be seen as current snapshot of the memory when running a program. It describes the values of each variable or field. A formula in JavaCardDL is evaluated in such a state.

Let i, j denote program variables. Some formulas in JavaCardDL:

- The formula
  $$i \doteq 0 \text{ -> } \langle i = i + 1; \rangle i > 0$$
  is a formula in JavaCardDL. The formual states:

  > If the value of i is 0 then the program $i = i + 1;$ terminates *and* in the final state (the state reached after executing the program) the program variable i is greater than 0.

  The diamond operator states implicitly that the program must terminate normally, i.e., no infinite loop/recursion and no uncaught exception).

  Replacing the diamond in the formula above by a box
  $$i \doteq 0 \text{ -> } [i = i + 1;] i > 0$$
  changes the termination aspect and does not require that the program terminates, i.e., this formula is already satisfied if in each state where the value of i is 0 and *if* the program $i = i + 1;$ terminates *then* in its final state i is greater than 0.

- A typical kind formula you will encounter is one with an update in front like
  $$\{i := a \,||\, j := b\} \, \langle tmp = i; i = j; j = tmp; \rangle i \doteq b \,\&\, j \doteq a$$
  Intuitively, an update can be seen as an assignment, the two vertical strokes indicate that the two assignments a to i and b to j are performed in parallel (simultaneously). The formula behind the update is then valid if in the state reached executing the two ‘assignments’, the program terminates (diamond!) and in the final state the content of the variables i and j have been swapped.

4.1.2 Sequents

Deduction with the KeY-Prover is based on a sequent calculus for a Dynamic Logic for JavaCard (JavaDL) [BHS07, Bec01].

A sequent has the form $\phi_1, \ldots, \phi_m \;\vdash\; \psi_1, \ldots, \psi_n$ $(m, n \geq 0)$, where the ϕ_i and ψ_j are JavaDL-formulas. The formulas on the left-hand side of the sequent symbol $\vdash$ are called *antecedent* and the formulas on the right-hand side are called *succedent*. The semantics of a sequent is the same as that of the formula $(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$ $(m, n \geq 0)$.

4.2 Proof-Obligations

In general a proof obligation is a formula that has to be proved valid. When we refer to a proof obligation, we mean usually the designated formula occurring in the root sequent of the proof. A method contract for a method m of a class C consists in general of a

precondition *pre* describing the method specific[8] conditions which a caller of the method has to fulfil before calling the method in order to be guaranteed that the

postcondition *post* holds after executing the method and that the

assignable/modifies clause *mod* is respected. This means that at most the locations described by *mod* are modified in the final state.

termination marker indicating if termination of the method is required. Termination required (total correctness) has termination marker `diamond`, i.e. the method must terminate when the called in a state where the precondition is fulfilled. The marker `box` does not require termination (partial correctness), i.e., the contract must only be fulfilled if the method terminates.

In addition each object O has a possibly empty set of invariants inv_O assigned to them.

For the general description we refer to this general kind of contract. Mapping of JML specification to this general contract notion is slightly indicated in Sect. 4.3. More details can be found in [BHS07, Eng05, Wei11].

4.3 Application to the Tutorial Example

Now we apply the described proof obligations to the tutorial example. First we demonstrate the generation of proof obligations, then we show how these can be handled by the KeY-Prover. Please make sure that the default settings of the KeY-Prover are selected (see Chapter 3.3) and the maximum number of automatic rule applications is 5000. Be warned that the names of the proof rules and the structure of the proof obligations may be subject to changes in the future.

[8] Additional conditions stem from invariants.

4.3.1 Method Specifications

Normal Behavior Specification Case. In the left part of the Proof Obligation Browser, expand the directory `paycard`. From the now available classes select `PayCard` and then the method `isValid`. This method is specified by the JML annotation

```
public normal_behavior
  requires true;
  ensures result == (unsuccessfulOperations<=3);
  assignable \nothing;
```

This JML method specification treats the `normal_behavior` case, i.e., a method satisfying the precondition (JML boolean expression following the `requires` keyword) *must not* terminate abruptly throwing an exception. Further each method satisfying the precondition must

- terminate (missing diverges clause),
- satisfy the postcondition – the JML boolean expression after the `ensures` keyword, and
- only change the locations expressed in the `assignable` clause; here: must not change any location. The assignable clause is actually redundant in this concrete example, as the method is already marked as `pure` which implies `assignable \nothing`.

Within KeY you can now prove that the implementation satisfies the different aspects of the specification, i.e., that if the precondition is satisfied then the method actually terminates normally and satisfies the postconditon and that the assignable clause is respected. In addition it is also proven that the object invariants of the object on which the methodcall is invoked are preserved.

The contracts pane in the Proof Management window offers only one proof obligation: `JML normal_behavior opertaion contract`. It summarizes all parts of the specification which will be considered in the proof.

The selected contract says that a call to this method, if the object invariant holds (**pre** `self.<inv >`), always terminates normally and that the `return` value is true iff the parameter `unsuccessfulOperations` is ≤ 3, Additionally the object invariant holds after the method call, therefore it is preserved. The sequent displayed in the large prover window after loading the proof obligation exactly reflects this property. We confirm the selection by pressing the *Start proof* button.

Start the proof by pushing the *Start*-button (the one with the green "play" symbol). The proof is closed automatically by the strategies. It might be necessary that you have to push the button more than once if there are more rule applications needed than you have specified with the "Max. Rule Applications" slider.

Exceptional Behavior Specification Case. An example of an exceptional behavior specification case can be found in the JML specification of method `charge(int amount)` in class `PayCard`. The exceptional case reads

```
public exceptional_behavior
    requires amount <= 0;
```

This JML specification is for the exceptional case. In contrast to the normal_behavior case, the precondition here states under which circumstances the method is expected to terminate abruptly by throwing an exception.

Use the Proof Management (either by clicking onto the button Proof Management or **File** | **Proof Management**). Continue as before, but select this time method `charge(int amount)` of class `PayCard`. In contrast to the previous example, the contracts pane offers you three contracts: two for the normal behavior case and one for the exceptional case. As we want to prove the contract for the exceptional case select the contract named: *JML exceptional_behavior operation contract*.

The KeY proof obligation for this specification requires that if the parameter `amount` is negative or equal to 0, then the method throws a `IllegalArgument-Exception`.

Start the proof again by pushing the *run strategy*-button. The proof is closed automatically by the strategies.

Generic Behavior Specification Case. The method specification for method `createJuniorCard` in `PayCard` is:

```
ensures \result.limit==100;
```

This is a lightweight specification, for which KeY provides a proof obligation that requires the method to terminate (maybe abruptly) and to ensure that, if it terminates normally, the `limit` attribute of the result equals 100 in the post-state. By selecting the `createJuniorCard` method, choosing *JML operation contract* named contract in the Contract Configurator, an appropriate JavaDL formula is loaded in the prover. The proof can be closed automatically by the automatic strategy.

4.3.2 Type Specifications

The instance invariant of type `PayCard` is

```
public invariant log.\inv
&& balance >= 0 && limit > 0
&& unsuccessfulOperations >= 0 && log != null;
```

The invariant specifies that the `balance` of a paycard must be non-negativ, that it must be possible to charge it with some money (`limit > 0`) and that the number of `unsuccessfulOperations` cannot be negative. Further the invariant expresses that the log file, which keeps track of the transactions must always exist (`log != null`) and that the instance referred to by `log` has to satisfy the instance invariant of its own class type (`log.\inv`).

The method `charge` of `PayCard` must preserve this invariant unless it does not terminate. This proof obligation is covered by proving that `self.<inv>` must be satisfied in the precondition and must hold in the postcondition. This has to be covered by all specification cases and is included in the specification in the Contract Configurator(`self.<inv>` in the pre- and in the postcondition).

There is one exception: if a method is annotated with the keyword `helper`, the proof obligation will not include the invariants. An example for such a method is `isValid()` of class `PayCard`. If a method is declared as helper method, the invariant of the object on which the method is called does not have to hold for its pre- and postcondition, for more details see [Wei11]. The advantage is a simpler proof obligation, however the disadvantage is that in order to show that the method fulfills its contract, it is not possible to rely on the invariants.

In addition invariants can also be annotated with an `accessible` clause. In this clause all fields and locations have to be included, on which the invariant at most depends on. For example, the invariant of class `LogRecord` is

```
!empty ==> (balance >= 0 && transactionId >= 0);
```

and it is annotated with the accessible clause

```
accessible \inv: this.empty, this.balance,
transactionCounter, this.transactionId;
```

This means the invariant of class `LogRecord` depends at most on the fields or locations and therefore given in the `accessible` clause. If one of these fields is changed, it has to be proven, whether the invariant still holds. The accessible clause can simplify proofs, because in cases where the heap is changed but not the mentioned fields and the validity of the invariant is proven before, we know that the invariant still holds.

4.3.3 Proof-Supporting JML Annotations

In KeY, JML annotations are not only input to generate proof obligations but also support proof search. An example are loop invariants. In our scenario there is a class `LogFile` which keeps track of a number of recent transactions by storing the balances at the end of the transactions. Consider the constructor `LogFile()` of that class. To prove the normal_behavior specification proof obligation of the method, one needs to reason about the incorporated `while` loop. Basically there are two possibilities do this in KeY: use induction or use loop invariants. In general, both methods require interaction with the user during proof construction. For loop invariants, however, *no interaction* is needed if the JML `loop_invariant` annotation is used. In the example the loop invariant, written in the JML notation, indicates that the array `logArray` contains newly created entries (`fresh(logArray[x])`) and these entries are not `null` in the array from the beginning up to position `i`:

```
/*@ loop_invariant
  @ 0 <= i && i <= logArray.length &&
  @ (\forall int x; 0 <= x && x < i; logArray[x] != null && \fresh(logArray[x]))
  @ && LogRecord.transactionCounter >= 0;
  @ assignable logArray[*], LogRecord.transactionCounter;
  @ decreases logArray.length - i;
  @*/
while(i<logArray.length){
   logArray[i++] = new LogRecord();
}
```

If the annotation had been skipped, we would have been asked during the proof to enter an invariant or an induction hypothesis. With the annotation no further interaction is required to resolve the loop.

Select the contract *JML normal behavior operation contract* of `LogFile`'s method `LogFile()`, and press *OK*.

Choose *Loop treatment None* and start the prover. When no further rules can be applied automatically, select the while loop including the leading updates, press the mouse button, select the rule *loopInvariant* and then *Apply Rule*. This rule makes use of the invariants and assignables specified in the source code. Restart the strategies and run them until all goals are closed.

As can be seen, KeY makes use of an extension to JML, which is that *assignable* clauses can be attached to loop bodies, in order to indicate the locations that can at most be changed by the body. Doing this makes formalizing the loop invariant considerably simpler as the specifier needs not to add information to the invariant specifying all those program variables and fields that are not changed by the loop. Of course one has to check that the given assignable clause is correct, this is done by the invariant rule. We refer to [BHS07] for further discussion and pointers on this topic.

5 Notes

The KeY-Tool is still very much work in progress so that parts of this tutorial may be outdated as you read it. Moreover, the JML semantics are still subject to discussions, and there is no formal semantics specification for JML. Differences between the JML semantics of other tools and the (implicitly given) semantics in KeY are therefore possible. The JML dialect of KeY even extends JML in some points (as we have seen above for *assignable* clauses in *loop_invariants*).

- Supported platforms:
  - Linux and MacOS X are tested, Solaris should work as well
  - Windows NT, 2000 and XP should work when using the KeY byte code version.
- Restrictions of the KeY-Prover:
  - manual not yet available
- Restrictions on JDK:
  - Problem: Sometimes windows show up "rolled up" and only the title bar is visible. This happens only if you use JRE 1.5.
    Solution: Use JDK 1.6
  - Problem: tool tips are flickering occasionally
    Workaround: reduce the number of tool tip lines in the menu **View**

A List of Menu Options

In the following we describe some menu items available in the main menu of the KeY-Prover. In this quicktour we will restrict ourselves to the most important ones.

File File related actions

- | **Load Example:** Opens a file browser with included example files.
- | **Load:** Loads a problem or proof file; selecting a directory opens the proof managment window with the generated proof obligation for the chosen specification language
- | **Reload:** Reloads last file loaded.
- | **Edit last opened file:** If a default external editor is set, this action opens the last opened file in the default external editor.
- | **Save:** Saves the current selected proof. Note, that if there are several proofs loaded (see the upper left pane) only the one currently worked on is saved.
- | **Proof Management:** Allows browsing through the available proof obligations respectively contracts.
- | **Prove:** | **User-Defined Taclets:** Loads user-defined taclets and generates the corresponding proof obligation.
  - | **KeY's Taclets:** Creates proof obligations for some selected taclets.
  - | **Taclets using the batch mode:**
- | **Load User-defined Taclets:** Allows to activate and deactivate theories given as taclet collection in a `.key` file.
- | **Recent Files:** List the last five loaded files (if they are still present).
- | **Exit:** Quits the KeY-Prover (be warned: the current proof is lost!).

View Settings influencing the look of the user interface

- | **Use Pretty Syntax:** If set, infix notation for functions and predicates are used.
- | **Use Unicode Symbols:** If set, unicode symbols instead of ASCII symbols are used for logical symbols.
- | **Font size** Changes the font size of the right prover pane
  - | **Smaller:** Decreases the font size.
  - | **Larger:** Increases the font size.
- | **ToolTip options:** Configures the tooltip shown when hovering over a taclet in the list of applicable taclets.
- | **Visual Node Diff:** Opens a new window which allows to view the difference between two chosen proof nodes.

Proof Proof specific options

| **Start:** Run the proof (semi-)automatically w.r.t. to current strategy options.

| **Goal Back:** Undo one proof step.

| **Abandon Task:** Quits the currently active proof. All other loaded problems will stay in the KeY-Prover.

| **Search in proof tree:** Opens a textfield in the proof pane, which allows to search for a string in the proof tree.

| **Search in sequent view:** Opens a window with a textfield in the lower-left corner of the current goal pane, which allows to search for a string in the sequent view.

| **Show used contracts:**

| **Show Active Taclet Options:** Shows the taclet options chosen for the current proof.

| **Show All Active Settings:** Opens a window displaying all settings used in the current proof.

| **Show Proof Statistics:** Shows some general statistics about the proof size and interactive steps.

| **Show Known Types:** Lists all types present in the current proof environment.

Options General options

| **Taclet Options:** In the following, each taclet option is described briefly. The respective default settings are given in parenthesis. The meaning of all settings is beyond the scope of this quicktour. Please use the default settings unless you know what you are doing. Note that this list is not complete.

javacard: (jcOff) There are two values for this option jcOn and jcOff. Switching on or off all taclets axiomatising JavaCard specific features like transaction. If switched off, the taclet options transactions and transactionsAbort have no effect.

assertions: (on) There exists are different values for this option

on evaluates assert statements and raises an `AssertionException` if the condition evaluates to false. This behaviour models the behaviour of the Java virtual machine with assertions enabled globally.

off skips evaluation of assert statement. In particular, the arguments of the assert statements are not evaluated at all. This behaviour models the behaviour of the Java virtual machine with assertions disabled globally.

safe using this option ensures that the shown property is valid no matter if assertions are globally enabled or disabled. Proofs with this option are typically harder.

Please note: There is no support other than option safe for enabling or disabling assertions package or class wise.

initialisation: (disableStaticInitialisation) Specifies whether static initialisation should be considered.

intRules: (arithmeticSemanticIgnoringOF) Here you can choose between different semantics for Java integer arithmetic (for details see [Sch02, Sch07, BHS07]). Three choices are offered:

javaSemantics (Java semantics): Corresponds exactly to the semantics defined in the Java language specification. In particular this means, that arithmetical operations may cause over-/underflow. This setting provides correctness but allows over-/underflows causing unwanted side-effects. This corresponds to the `code_java_math` macro in JML.

arithmeticSemanticsIgnoringOF (Arithmetic semantics ignoring overflow, default): Treats the primitive finite Java types as if they had the same semantics as mathematical integers with infinite range. Thus this setting does not fulfil the correctness criteria. This corresponds to the `code_bigint_math` macro in JML.

arithmeticSemanticsCheckingOF (Arithmetic semantics prohibiting overflow): Same as above but the result of arithmetical operations is not allowed to exceed the range of the Java type as defined in the language specification. This setting not only enforces the java semantics but also ascertains that no overflow occur. This corresponds to the `code_safe_math` macro in JML.

programRules: (Java) Changes between different program languages[9].

model fields The semantics of model fields is given by the `represents` clause in the JML specification. The setting of this option decides how the represents clauses are handeled. It has two possible values treatAsAxiom and showSatisfiability:

treatAsAxiom: Represents clauses are seen as axioms. If this option is set the satisfiability of the represents clauses is not shwon and therefore it may introduce inconsistent specifications, e.g., he following contradictory JML clause will not be rejected:

```
//@ represents modelField == modelField + 1;
```

showSatisfiability: For every expansion of the represents clause, the satisfiability of the definition has to be shown. Cross-definition inconsistencies can still be formulated, however:

[9]Ensure that *Java* is selected.

```
//@ represents modelField1 == modelField2;
//@ represents modelField2 == modelField1 + 1;
```

runtime exceptions There are two possible ways for the KeY-Prover in handling runtime exceptions – ban or allow.

ban: If runtime exceptions are banned, KeY treats the occurence of runtime exceptions as irrecoverable program failure. Setting this option results in smaller proofs and is complete for defensive programmed programs, i.e. programs which do not intentionally use corner cases.

allow: If runtime exceptions are allowed, KeY treats runtime exceptions as defined in the Java language specification, i.e. implicit runtime exceptions[10] are raised and therefore such exceptions have to be considered in the proof. Setting this option results in larger proofs.

The current setting of the taclet options can be viewed by choosing **Proof | Show Active Taclet Options**.

| **Minimize Interaction** If this option is used and the automtaic strategy is used, KeY tries to minimize user interaction. That means that for example, if the KeY-Prover is able to find instantiations by itself, the user is not asked to provide them.

| **Right Click for Macros**

| **One Step Simplification** In the KeY-Prover one step simplification is a mechanism to automatically apply several simplifying and normalization rule applications to the sequent. For the user these rule applications are aggregated into one visible rule application One Step Simplification in the proof tree. Setting this option often leads to simpler sequents and results in finding a proof faster, but the user lacks transparency of the proof, because the rule applications of the one step simplifier are not shown in very detail compared to all other rule applications in the KeY-Prover.

| **SMT Solvers:** This option allows you to choose one or more external decision procedures that can be invoked during proofs and to set options for each external solver seperately. There is a native interface to Simplify. A variety of other provers CVC3, Yices, and Z3 are directly supported via SMTLIB [BRST08]. In addition, translations of taclets to the SMTLIB language can be written to a text file (Taclet Translation) to be loaded by any SMT prover. There are further options on the set of taclets to translate.

[10]With implicit exceptions, we mean exceptions not explicitely programmed by the developer using `throw new java.lang.exception`.

B List of all Strategy Tab Settings

Max. Rule Applications You can set the number N_{aut} of automatic rule applications using the slider. Even if the automatic strategy can still apply rules after N_{aut} applications, automatic proving stops.

Stop At Choose when strategy execution shall stop. Possible values are `Default`: strategy stops when no rules are applicable or the maximal number of steps is reached and `Unclosable`: strategy stops in all situations when `Default` stops but also already when the first goal is encountered on which no further rule is (automatically) applicable.

Proof splitting Influences usage of rules branching a proof tree. Only rules working on formulas not on programs fall under the chosen policy, i.e., program rules causing splits are still applied even if splitting is switched off. The values are `free` (withour restrictions), `Delayed` (allows still splitting but prefers other rules) and `Off` (no splitting).

Loop treatment This allows to set up how while-loops are treated. They can be left untouched (`None`), handled using stated invariant contracts (`Invariant`), or repeatedly unrolled (`Expand`).

Block treatment It is possible to specify Java blocks with contracts. This option allows to set how KeY treats such contracts.

contract: If this option is set, Java blocks are replaced by their contract. Three properties have to be shown in the proof:

- the validity of the contract
- the pre condition if the contract is satisfied
- the use case of the contract

expand: If this option is set, Java blocks are expanded and block contracts are not used.

Method treatment Methods can also be left untouched (`None`), have their method contracts applied (`Contracts`), or be inlined, i.e. have the method body expanded in place (`Expand`).

Dependency contract For the simplification of heap terms setting this option to `On` the information in JML's `accessible` clause is used.

Query treatment A query is a method that is used as a function in the logic and stems from the specification. There are three options for query treatment in KeY:

On: Rewrite a query to a method call such that contracts or inlining (dependent on the method treatment setting) can be used.

Restricted: Same as `On` but with restrictions:

- Priority of expanding queries that occur earlier on a branch is higher than for queries introduced more recently. This approximates in a breath-first search with respect to query expansion.
- Reexpansion of identical query terms is suppressed.
- A query is not expanded if one of its arguments contains a literal greater than a computed limit[11], or smaller than a computed limit. This helps detecting loops in a proof
- Queries are expanded after the loop body in the `Preserves Invariant` branch of the loop invariant rule
- Queries are expanded in the `Base Case` and the conclusio of the `Step Case` branch when using Auto Induction

Off: The query statements are ignored and the proof has to be done without using them.

In addition the KeY-Prover offers a setting for the expansion of local queries in certain safe cases. Safe cases are:

- the return type of the expanded method is known
- the object on which the methodcall is invoked is self or a parent of self.

This setting is indepedent of the query treatment setting.

Arithmetic treatment The KeY-Prover has several options for the treatment of arithmetic expressions:

Basic: Using this option, polynomial expressions are simplified. In the antecedent Gröbner Bases are computed polynomials. Linear inequations are handled using (partial) Omega procedures.

DefOps: Using the option DefOps, symbols such as:
`/`, `%`, `jdiv`, `jmod`, ...
`int_RANGE`, `short_MIN`, ...
`inInt`, `inByte`, ...
`addJint`, `mulJshort`, ...
are expanded.

Model Search: Setting the model search option, the KeY-Prover supports non-linear equations and model search. Additionally multiplication of inequations with each other and systematic case distinctions (cuts) can be performed. This method is guaranteed to find counterexamples for invalid goals that only contain polynomial (in)equations. Such counterexamples turn up as trivially unprovable goals. It is also able to prove many more valid goals involving (in)equations, but will in general not terminate on such goals.

[11]The computation of this limit is done with sophisticated methods for loop detection and would go beyond the scope of this quicktour

Quantifier treatment Sometimes quantifiers within the sequent have to be instantiated. This can be either done manually (`None`) or automatically with different alternatives:

`No Splits` Instantiate a quantifier only if this will not cause the proof to split.

`Unrestricted` Instantiates a quantifier even when causing splits. However the startegy tries to predict the number of caused open branches and will prefer those with no or only few splits.

`No Splits with Progs` Chooses between the `No Splits` and `Unrestricted` behaviour depending on prgrams present in the sequent. If a program is still present the `No splits` behaviour is used. Otherwise quantifiers are instantiated unrestricted

C Handy Shortcuts and Buttons

In the following an overview of all shortcuts currently used in the KeY-Prover is given. Additional, if buttons in the toolbar exist, their actions are listed here as well.

Action/Command	Shortcut	Button in Toolbar
File related		
Load	Ctrl+O	📁
Reload	Ctrl+R	📂
Save	Ctrl+S	💾
Proof Management	Ctrl+M	Proof Management
Exit	Ctrl+Q	-
Edit last opened file	-	📝
Appearance		
Use pretty syntax	Ctrl+P	-
Font size: smaller	Ctrl+Up	-
Font size: larger	Ctrl+Down	-
Proof specific		
Start automatic strategy	Ctrl+S	▶
Abandon task	Ctrl+W	-
Undo last rule application	Ctrl+Z	↶
Search in proof tree	Ctrl+F	-
Search in sequent view	F3	-
Prune tree below selected node	-	✂
SMT Solver	-	Run Z3, CVC3 ▾
Options		
Taclet options	Ctrl+T	-
Toggle one Step Simplifier	Ctrl+Shift+S	1ss

D Setting Up Own Projects

D.1 API of Supported Standard Library Classes

If not specified otherwise via a classpath directive, KeY includes a restricted set of signatures of classes and methods from the default standard library. A complete listing of them is available as separate document [Red].

Sorry this chapter needs still to be written. If you run into a situation where you need information about the classpath directive, please

- look into the `README.classpath` file contained in the subdirectory `doc/` of the source code distribution.
- do not hesitate to ask for further support at `support@key-project.org`.

References

[ABB$^+$05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Bec01] Bernhard Beckert. 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, pages 6–24. Springer-Verlag, 2001.

[BHS] Thomas Baar, Reiner Hähnle, and Steffen Schlager. Key quicktour. See `http://www.key-project.org/download/`.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[BRST08] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[Eng05] Christian Engel. A translation from jml to java dynamic logic. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, January 2005.

[LBR04] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science, November 2004. See `http://www.jmlspecs.org`.

[LPC$^+$11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Mller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. Jml reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, July 2011.

[Red] JavaRedux - API. API documentation of a restricted subset of the Java Standard Library Classes.

[Sch02] Steffen Schlager. Handling of Integer Arithmetic in the Verification of Java Programs. Master's thesis, Universität Karlsruhe, 2002. Available at: `http://i12www.ira.uka.de/~key/doc/2002/DA-Schlager.ps.gz`.

[Sch07] Steffen Schlager. *Symbolic Execution as a Framework for Deductive Verification of Object-Oriented Programs.* PhD thesis, Fakultät für Informatik der Universität Karlsruhe, February 2007.

[Wei11] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.