KeY Quicktour for JML

Work in progress

Christian Engel, Andreas Roth, Abian Blome,
Richard Bubel, Simon Greiner

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

March 25, 2009

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 its specification in 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+08, 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 UML/OCL and JavaCardDL as specification languages. Later versions of this quicktour will cover them – for the moment we can refer only to an outdated quicktour for OCL [BHS] from which this document evolved.

For a longer discussion on the architecture, design philosophy, and theoretical underpinnings of the KeY-Tool please refer to [BHS07, ABB+05].

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 1.4.

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.

Attention: Support for Borland Together has been discontinued with KeY 1.4. Nevertheless the code is still present and the basic functionality should work with the stand-alone (non-eclipse) versions of Together Solo or TogetherCC, but that is without guarantee. The bytecode resp. sourcecode version is required for the Borland Together plug-in.

1.2.3 The KeY-plugin for Eclipse

In this section we will describe how to setup the KeY-plugin for Eclipse as well as the use of some of its core features. We assume that Eclipse has already been installed on the target computer. Start it and in the menu *Help* select *Software Updates* and then *Find and Install*. In the new window activate *Search for new features to install* and click on *Next*. Now add the *New Remote Site* `http://www.key-project.org/KeYDists/KeY_Feature/` with a name of your choice. Click on *Finish*. Now a new window should appear with a list of installable features. Mark KeY and continue. After accepting the license agreement and selecting a usable location you will be asked to verify the installation. Do so by clicking on *Install all* and restart Eclipse when asked to. This completes the installation. It is possible to update the KeY-plugin by selecting *Search for updates of the currently installed features* in the *Software updates* menu. You can now start the standalone version of the prover by either clicking on the KeY logo in the menu bar or from the *Verification* menu.

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 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+08, 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+08]. 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 ia 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+08]
[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` or `startProver` script of your KeY distribution[5], e.g. by running

`bin/runProver` or `bin/startProver`

If you use the Eclipse intall, start the KeY-Prover as stand-alone tool via the menu **Verification**.

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.

Before you continue, please check in the menu bar that under **Options** | **Specification Languages** the option **JML** is activated (if not, please select it).

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`

     or

     `bin/startProver <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`.*

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

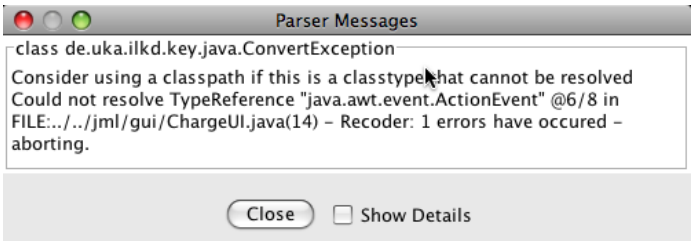


Figure 1: Error dialog complaining about an unknown type

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. B.

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

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

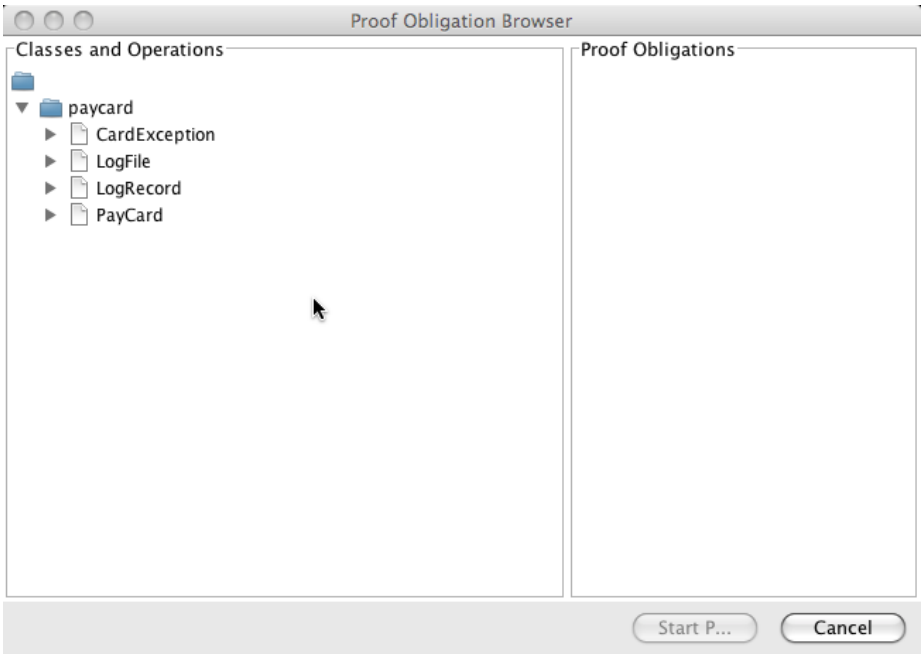
   The browser allows you to select the proof obligation (kind of property), you want to verify. Selecting method `charge` of class `PayCard` offers a number of proof obligations (Fig. 2(b)) such as **PreservesInv**, **EnsuresPost**, **RespectModifies** and several more.

   Some of the proof obligations are explained in Sect. 4 for a complete and detailed overview see Chap. 5 and 8 in [BHS07] and [Rot06].

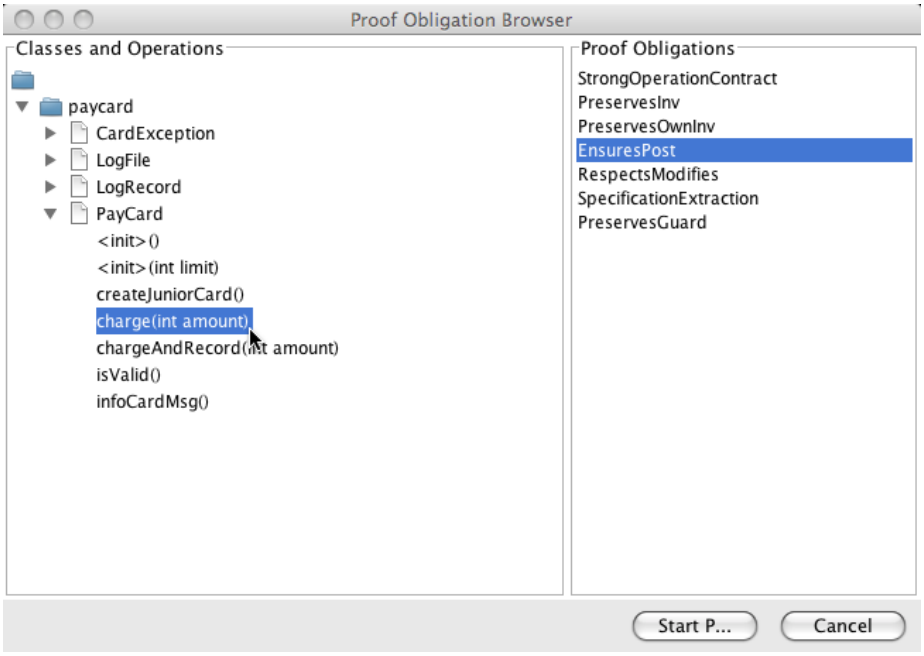
   For the moment please select **EnsuresPost** and press the button **Start Proof**. In the then upcoming Contract Configurator window three contracts should be displayed, one **exceptional_behavior** and two **normal_behavior** contracts. Select that **normal_behavior** contract which in its **post**condition 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.



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



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

Figure 2: The Proof Obligation Browser window

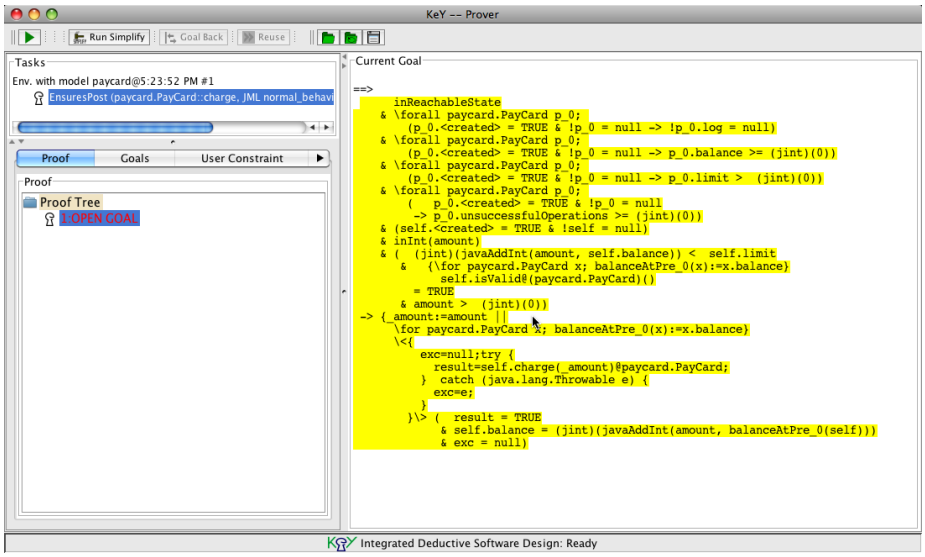


Figure 3: The KeY-Prover with loaded proof obligation **EnsuresPost** for method `charge` of class `PayCard`

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.

The KeY-Prover window consists of three panes where the lower left pane is additionally tabbed. Each pane is described below.

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.[6]

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

[6]During this quicktour you should always load a problem in a new proof environment. So if you are asked whether you want to re-use a proof, please select *Cancel*.

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 the node you had clicked on belongs to.

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.

User Constraint: To explain this functionality would go beyond the scope of this quicktour. It won't be required in the sequel.

Rules: In this pane (Fig. 4(c)) 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, a window comes up where the corresponding rule is explained.

Proof Search Strategy: This tab (see Fig. 4(b)) 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:

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. If the checkbox *Autoresume strategy* is selected, the prover automatically resumes applying the strategy after an interactive rule application.

FOL vs. JavaDL If you want to prove some properties of a JAVA-program you should use the strategy *Java DL*, as in the sequel of this quicktour. For purely first order logic problems use the strategy *FOL* (which stands for First Order Logic).

Goal Chooser Choose how strategies are exploring branches. Usually `Default` is to be preferred except for large proofs where `Depth First` shows a significantly lower memory footprint.

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 `Non-closeable Goal`: strategy stops in all situations when `Default` stops but also already when a goal is encountered on which no further rule is (automatically) applicable.

Logical splitting Influences usage of rules branching a proof tree. Logical means 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 `Normal`, `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 Method 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*).

Query treatment Queries used as terms in formulas are evaluated either by symbolical execution (`Expand`), or are moved to the succedent (`Prog2Succ`) so that contracts can be used, or are not evaluated at all (`None`).

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

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 ▷ - 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.

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.

Tools menu for executing available tools like the Proof Obligation Browser.

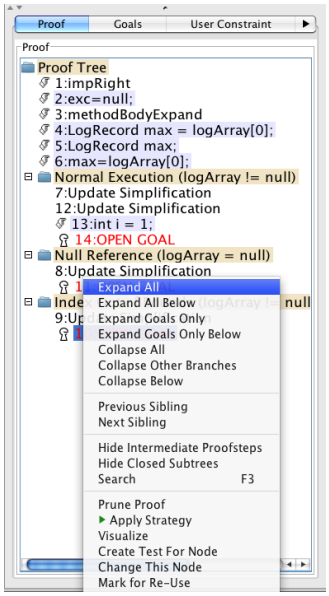
About menu (as the name says).

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.

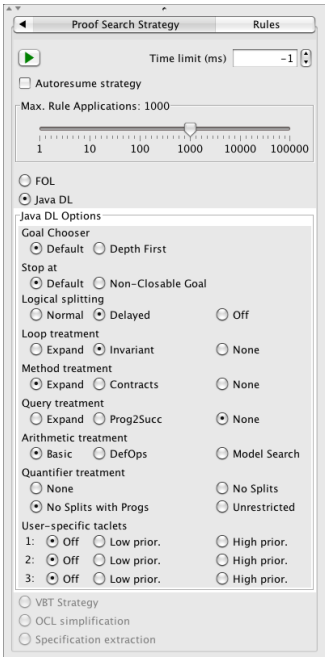
For simplicity, we deactivate some advanced concepts and configure the KeY-Prover use the normal arithmetic integers to model Java integer types, which will avoid to deal with modulo arithmetics. *Important:* Please note that this configuration is unsound with respect to the Java semantics.

In order to configure the KeY-Prover in the mentioned way select **Options** | **Default Taclet Options**. The dialog shows a list of available options. The list below explains the options necessary for this tutorial[7]. 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.

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



(a) The Proof tree tab



(b) The Proof Search Strategy tab

(c) The Rules tab

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

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.

arithmeticSemanticIgnoringOF (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.

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

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 and choose the following setting:

- Autoresume strategy should be unchecked (otherwise the prover will switch to automatic mode after each interactive rule application).
- Max. Rule Applications should be set to a value greater or equal 5000. 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:
  - Goal Chooser: Default
  - Stop at: Default
  - Logical splitting: Delayed (Normal should also work)
  - Loop treatment: Invariant
  - Method treatment: Expand

– Query treatment: Expand
– 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]. 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 does not support the full verification of a program. Instead, the KeY-Tool generates lightweight proof obligations that enable developers to prove selected properties of their program. These properties are of two kinds:

- *properties for method specifications:* we show that a method *fulfils* its method contract
- *properties for class specifications:* we show that a method *preserves* invariants of a class[8].

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 first argument takes a piece of JavaCard code and the second argument an arbitrary formula. Let p be a program and *post* 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.

[8] Earlier versions supported history constraints. KeY 1.4 underwent a complete rewrite concerning proof-obligations and JML. We are currently working on bringing all features back and more.

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

- The formula
$$i \doteq 0 \texttt{ -> } \langle i = i + 1; \rangle i > 0$$
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
$$i \doteq 0 \texttt{ -> } [i = i + 1;] i > 0$$
changes the termination aspect and does not require that the program terminate, i.e., this formula is already satisfied in a 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.

In the following sections we sketch the most important proof obligations generated to prove that methods and classes respect certain parts of their specification.

4.2.1 Selected Proof Obligations for Methods

The Proof Obligation Browser provides a selection of proof-obligations to verify different aspects of a method specification (also called method or operation contract). In this section we list several of them and explain them in brief. For a full coverage see [BHS07, Rot06].

A method contract for a method m of a class C consists in general of a

precondition *pre* describing the method specific[9] conditions 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. That 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 class D has a possibly empty set of invariants inv_D 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].

Let us have a closer look into some of the proof-obligations offered by the Proof Obligation Browser for a method m of class C:

EnsuresPost this proof-obligation generates a formula that is valid if a method fulfils its specification. Roughly spoken, if the precondition and a given set of invariants is satisfied then the post conditions holds and – optionally – the method terminates. The set of additional invariants is user customisable and can be selected in the Contract Configurator.

PreservesInv allows to verify that method m preserves validity of a given set of invariants INV. The generated proof-obligation states that when m is called in a state where its precondition and all invariants in INV hold then all of the invariants are also valid in the method's final state.

[9] additional conditions stem from invariants

PreservesOwnInv a special case of PreservesInv where set INV contains exactly all invariants of class C. This implements a proof-obligation for an often used lightweight specification.

RespectsModifies generates a formula used to verify the assignable/modifies clause.

The Contract Configurator allows to choose between different specifications available for a method, i.e. for different pre- and postcondition pairs. For most proof obligations it offers also to choose sets of invariants that are assumed to hold in the prestate (tab Assumes) and/or that must be ensured to hold in the poststate.

4.2.2 Selected Proof-Obligations for Classes/Interfaces

KeY offers currently only one proof-obligation on the class resp. interface level namely BehaviouralSubtypingInv. This proof-obligation ensures that the invariants of the chosen class imply the invariant of their superclass(es).

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), especially that the current strategy is *Java DL* 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.

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 caluse),

- 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 or that the assignable clause is respected. We concentrate know on the first aspect.

Choose the proof obligation *EnsuresPost* in the right pane and press the button *Start Proof*. The next dialog that pops up is the Contract Configurator, it allows to select the contract you want to prove. We select the only **normal_behavior** contract which is offered. The configurator also offers you the possibility to customise the set of assumed invariants. By default exact the invariants of the declaring class (here: `PayCard`) are selected. We will simply keep the default selection and confirm by pressing the *OK* button.

The selected contract says that a call to this method always (**pre** `true`) terminates normally and that the `return` value is true iff the parameter `unsuccessfulOperations` is ≤ 3. The sequent displayed in the large prover window after loading the proof obligation exactly reflects this property.

First, select the checkbox *Autoresume strategy* in the tab *Proof Search Strategy* in the lower left pane and then start the proof by pushing the *run strategy*-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;
    assignable \nothing;
```

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. The assignable clause states that even in this case, no fields are allowed to be changed.

Use the Proof Obligation Browser (*Tools→Proof Obligation Browser...*). Continue as before, but select this time method `charge(int amount)` of class `PayCard`. In contrast to the previous example, the Contract Configurator 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*. For the assumed invariants we will keep the default selection and we confirm our selection by pressing the *OK* button.

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. We may assume the invariants of `PayCard`. By selecting the `createJuniorCard` method, choosing *EnsuresPost* again and then *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 strategy *Java DL*.

4.3.2 Type Specifications

The instance invariant of type `PayCard` is

```
this.balance >= 0
&& this.limit > 0
&& unsuccessfulOperations >=0;
```

The method `charge` of `PayCard` must preserve these invariants unless it does not terminate. The KeY proof obligation to check this property is called *PreservesOwnInv*. Please read carefully as there is also a proof obligation called *PreservesInv* which allows you to customise the set of invariants to be preserved.

Open the Proof Obligation Browser, once again and select class `PayCard` and method `charge(int amount)`. Then choose the proof obligation *PreservesOwn-Inv* and proceed as usual. The proof closes automatically.

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 method `getMaximumRecord()` in that class. It returns the stored log entry (`LogRecord`) with the greatest balance. 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 variable `max` contains the largest value of the traversed part of the array (up to position `j`):

```
/*@ loop_invariant 0<=i && i <= logArray.length
  @                && max!=null &&
  @   (\forall int j; 0 <= j && j<i;
  @    max.balance >= logArray[j].balance);
  @ assignable max, i;
  @*/
while(i<logArray.length){
  LogRecord lr = logArray[i++];
  if (lr.getBalance() > max.getBalance()){
    max = lr;
  }
}
```

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.

Load the *EnsuresPost* proof obligation of `LogFile`'s `getMaximumRecord()`, select the contract *JML normal behavior operation contract* and press *OK*.

Choose the strategy *Java DL* and the *Loop treatment None*. Make sure *Autoresume strategy* is selected and start the prover. When no further rules can be applied automatically, select the while loop including the leading updates, press the mouse button and select the rule *loopInvariant*. This rule makes use of the invariants and assignables specified in the source code. Several goals remain open after the strategies have resumed their work.

Restart the strategies and run them until only one goal is left open, pressing *Run Simplify* should then close the remaining goal.

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.

4.3.4 Using the KeY-plugin for Eclipse

This section is currently out-of-date as the eclipse plugins are undergoing restructuring. The principal approach is still valid.

This section will give a quick overview on the visualization features added by the KeY-plugin for Eclipse. We will assume that the plugin has already been installed as described above. Start Eclipse and open the *PayCard* project using the *Import* dialog from the *File* menu. The *paycard* directory should appear on the right hand side. Now open the proof visualization by selecting *Other* in the *Show View* submenu inside the *View* menu. Once there select *Proof Visualization* in the *KeY* branch and click *OK*. Now it is time to actually open one of the classes, in this example we will use LogFile.

Open the KeY-Tool by clicking on the KeY-logo in the toolbar. As before select the *PayCard* project and mark the *normal_behavior speccase* for the method *getMaximumRecord* in the *LogFile* class. Now start the proof. A number of open goals will remain but this time we won't deal with them, our focus is on the Eclipse plugin.

It is time to take a closer look at the visualization options in Eclipse. Return to it and press the *Show Execution Traces* button from the *Proof Visualization* view. A new window should pop up with a number of execution traces available. Checking the *Filter uninteresting traces* option hides those traces that appear to be irrelevant to the understanding of the current proof. In this case it should leave you with a single trace. Mark it and click *OK*. Now you will actually see the execution trace of the selected node in the *Proof Visualization* view. Additionally all executed statements, except for the last one, are highlighted in yellow inside the Java editor. In case of an exception the last statement is highlighted in dark red, otherwise in dark yellow. You can navigate through the trace using the buttons *Step Into* and *Step over*. The first one allows you to mark the next executed statement while the last one jumps over substatements and method calls. By right-clicking on a branch you can also choose to go into it. To return to the main execution trace press the *Home* button. Pushing the *Mark All Statements* button remarks all statements of the trace in the Java editor. If you want to clear all markers you can press the red cross. It is possible to receive more information on single statement, like the node at which the statement was executed, by moving the mouse over the marker bar left of the Java editor.

In case Eclipse is not available you can use a more rudimentary visualization built directly in the KeY-Tool. You can access it by right-clicking on a node and selecting *Visualize*. This opens a new window with a list of traces. Again you have the chance to *Filter uninteresting traces* and you get to see the trace in a tree-like structure. Statements that produced exceptions are highlighted in red.

The visualization options presented above concentrate on the symbolic execution. They allow an intuitive way for analyzing the current proof branch in a way that is similar to classic debuggers.

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 KeYeven 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:** Loads a problem or proof file; selecting a directory opens the proof obligation browser with the generated proof obligation for the chosen specification language (see **Options** | **Specification Languages**)
- **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.
- **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:** This menu item allows you to toggle between two different syntax representations. If checked a nicer and easier to read syntax is used.
- **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.

Proof Proof specific options

- **Abandon Task:** Quits the currently active proof. All other loaded problems will stay in the KeY-Prover.
- **Show Active Taclet Options:** Shows the taclet options chosen for 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

- **Default 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.

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

- **on (default)** 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.
- **arithmeticSemanticIgnoringOF** (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.
- **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.

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.

nullPointerPolicy: (nullCheck) Specifies if nullpointer checks should be performed when evaluating reference access expressions. If turned off, no `NullPointerException`s will be raised when dereferencing a `null` reference.

programRules: (Java) Changes between different program languages [10].

throughout: (toutOn) Depending on the chosen value toutOn or toutOff KeY features a throughout operator allowing to verify strong invariants. These are invariants that must hold in each intermediate state of execution.

transactionAbort: (abortOn) Turns on or off support for the abort case of transactions.

transactions: (transactionsOff) Specifies how to handle the JavaCard Transactions.

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

| **Update Simplifier:** Fine tuning of the update simplifier. For example, the deletion of superfluous updates can be switched off.

| **Taclet Libraries:** Allows to activate and deactivate theories given as taclet collection in a `.key` file.

| **Decision Procedures:** This option allows you to choose an external decision procedure that can be invoked during proofs. There is a native interface to the provers Simplify and ICS. A variety of other provers CVC3, CVCLite, SVC and Yices are directly supported via SMT [BRST08]. In addition the SMT translation can be written to a text file (SMT Translation) to be loaded by any SMT prover. There are further options to turn on/off quantifier support (Translate Quantifier (SMT)), packing all SMT problems (one per goal) into one zip file (Archive SMT Benchmarks) with or without the original problem files (Zip problem dir into archive).

| **Specification Extraction:** Here you can choose between different settings for the automatic computation and specification.

| **Specification Languages:** There are three values for this option None, JML and OCL. If JML or OCL are chosen the loaded files are scanned for JML resp. OCL annotations from which if found proof obligations are generated. The generated proof obligations can be invoked via **Tools** | **Proof Obligation Browser**.

| **Minimize interaction:** If this checkbox is selected, checkbacks to the user are reduced. This simplifies the interactive rule application.

| **DnD Direction Sensitive:** Some taclets can be instantiated via drag and drop. Activating or deactivating this option determines if the direction of drag and drop action shall be taken into account when filtering the applicable taclets.

| **Sound:** By selecting this checkbox sound notifications can be turned on or off (on).

[10] Ensure that *Java* is selected.

| **Proof Assistant:** Kiki the proof assistant can be turned on or off via this option (on).

Tools Additional tools

| **Extract Specification:** Extracts the specification of a program.

| **Proof Obligation Browser:** Allows browsing through the available proof obligations. Proof obligations can be generated from JavaDL, JML or OCL specifications, option **Options** | **Specification Languages** allows to select from which one.

| **Check Non interference:** (experimental testbed) Generates proof obligation to show non interference.

| **Create Unittests:** Creates JUnit testcases from the proof.

| **Create JML-Wrapper** Generates a JML specification (requires/ensures) pair from a proof [BG07]. The program with the generated JML specification can then be feed into an automatic test generation tool [EH07].

B Setting Up Own Projects

B.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].

B.2 The `classpath` Directive

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.

[BG07] Bernhard Beckert and Christoph Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In B. Meyer and Y. Gurevich, editors, *Proceedings, International Conference on Tests and Proofs (TAP), Zurich, Switzerland*, LNCS 4454. Springer, 2007.

[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.

[EH07] Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Bertrand Meyer and Yuri Gurevich, editors, *Proc. Tests and Proofs (TAP), Zürich, Switzerland*, volume 4454 of *LNCS*. Springer-Verlag, 2007.

[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+08] 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`, May 2008.

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

[Rot06] Andreas Roth. *Specification and Verification of Object-oriented Components*. PhD thesis, Fakultät für Informatik der Universität Karlsruhe, June 2006.

[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.