KeY Quicktour for JML

Christian Engel, Andreas Roth and Abian Blome[*]

1 Introduction/Prerequisites

This document constitutes a tutorial introduction to the KeY-Tool using the JML interface, called JMLKeY in the sequel. The KeY-Tool is designed as an integrated environment for creating, analyzing, and verifying software models and their implementation. The focus of the KeY-Tool is to consider UML/OCL models, especially UML class diagrams. Due to the increasing popularity of the *Java Modeling Language* (JML) [LPC+02, LBR04] and the success of the prover component of KeY, which implements a calculus for the complete JavaCard language, the KeY group has provided an interface for this specification language to KeY, which is described in this document.

For a longer discussion on the architecture, design philosophy, and theoretical underpinnings of the KeY-Tool please refer to [ABB+05]. A tutorial introduction to the KeY-Tool with its UML/OCL interface is provided in [BHS], which this document is also based on.

The most recent version of the KeY-Tool can be downloaded from `http://download.key-project.org`. We assume that the KeY-Tool has already been installed successfully.

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 0.99 (internal 0.2246).

1.2 Logical Foundations

Deduction with the KeY-Prover is based on a sequent calculus for a Dynamic Logic for JavaCard (JavaDL) [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 $(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$ $(m, n \geq 0)$.

1.3 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://* with

---

[*]Universität Karlsruhe. This article is a variant of [BHS] by Thomas Baar, Reiner Hähnle, and Steffen Schlager.
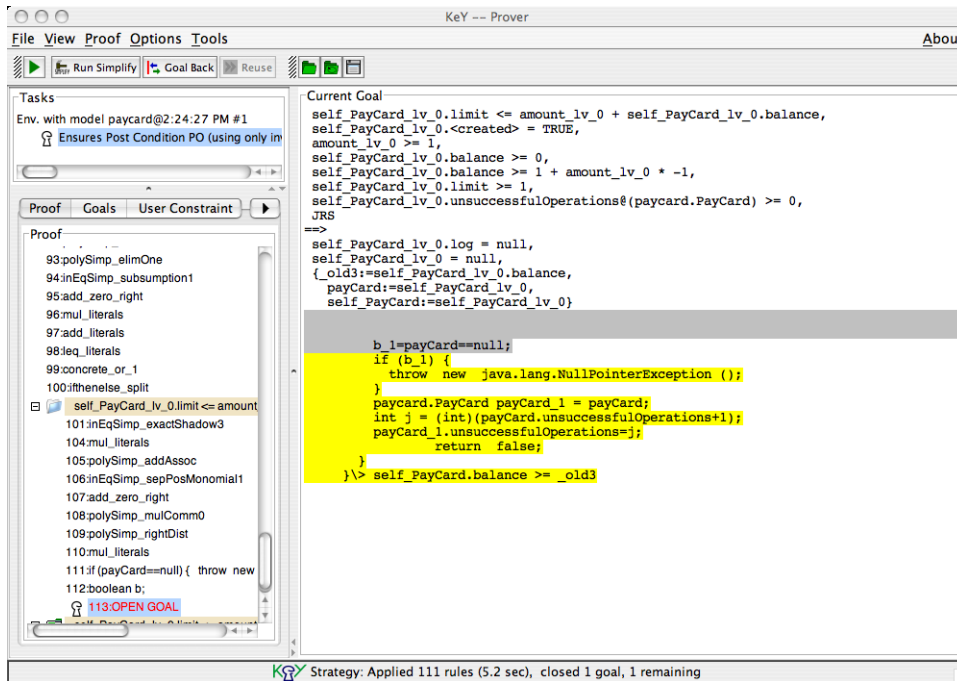
Figure 1: The KeY-Prover.

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. In this menu you can also find the *JML Specification Browser* whose function is described below.

1.4 The KeY-Prover

In this section we give a short introduction into the handling of the KeY-Prover which is shown in Figure 1. 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.[1]

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

> **Proof:** This pane 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

---

[1] 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*.

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 the node you had clicked on belongs to.

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.

Proof Search Strategy: This tab allows you to define the active strategy from a set of available strategies. 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 pure logic problems use the strategy *FOL*. Furthermore, you can set the maximum number of automatic rule applications by moving the slider. If the checkbox *Autoresume strategy* is selected, the prover automatically resumes applying the strategy after an interactive rule application. The *Loop-*, *Method-* and *Query treatment* options allow you to specify which strategy KeY ought to follow when these constructs are encountered and are, except for the latter, set to *Expand* by default. The other two options (*OCL simplification* and *Specification extraction*) are of no relevance for the remainder of the quicktour.

Rules: In this pane 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.

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.

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 → Save: Saves current proof. Note, that if there are several proofs loaded (see the upper left pane) only the one currently worked on is saved.

File → Exit: Quits the KeY-Prover (be warned: the current proof is lost!).

View → 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.

View → Font size → Smaller: Decreases the font size in the right prover pane.

View → Font size → Larger: Increases the font size in the right prover pane.

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

Options → Taclet Options Defaults : 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.

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

  testGeneration: Turned only be activated during test generation (testOff).

  programRules: Changes between different program languages (Java)[2].

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

  transactionAbort: Turned on by default (transactionAbortOn).

  throughout: Turned on by default (toutOn).

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

  - 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.
  - 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 fulfill the correctness criteria.
  - 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.

  assertions: Turned on by default (on).

---

[2]Ensure that *Java* is selected.

nullPointerPolicy: Specifies if nullpointer-checks should be done or not (nullCheck).

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

policies on how updates should be simplified. Like the description of Taclet
Options Defaults above, this goes beyond the scope of this quicktour. Please
use the default settings if you are not familiar with it.

Options → Decision Procedure Configuration: Distincts between different in-
teger decision procedures, e.g. *Simplify*[3] [Sim] and *ICS* [ICS]. During this
quicktour, the procedure *Simplify* should be selected.

Options → Compute Specification: Here you can choose between different set-
tings for the automatic computation and specification.

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

Options → Sound notification: By selecting this checkbox you can turn off the
sound notifications.

Options → Proof Assistant: By selecting this checkbox you can turn off the
proof assistant.

Options → Save Settings: Here you can save changes to the settings in the menu
Options permanently, i.e. for future sessions with the KeY-Prover.

Tools → Extract Specification: Extracts the specification of a program.

Tools → JML Specification Browser: Allows browsing through the available
JML Specifications.

Tools → Check Non interference: (experimental testbed) Generates proof obli-
gation to show non interference.

Tools → Create Unittests: Creates JUnit testcases from the proof.

2 Tutorial Example

In this tutorial we use a simple paycard application to illustrate some basic ca-
pabilities offered by the KeY-Tool. The tutorial example consists of several Java
classes in a folder `paycard`: `PayCard`, `PayCardJunior`, `CardException`, `ChargeUI`,
`IssueCardUI`, and `Start`. The class `Start` provides the `main` method of the ap-
plication. You can compile and execute the application as usual, for instance with
`javacc` and `java`. Try this now.

The tutorial scenario executed by the main method in `Start` is as follows: A
dialog asks the customer (i.e. the user of the application) to obtain a paycard with
a certain limit: 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 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

---

[3]Simplify is part of ESCJava2. We have been allowed to offer a binary download version on
our website.

greater or equal the limit of the paycard, the charge operation does not change the balance on the paycard and, depending on the class, either an attribute counting unsuccessful operations is increased or an exception is thrown. 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⁺02, 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⁺02]. JMLKeY is no substitute for the JML parser / type checker.

The UML/OCL version of the KeY-Tool provides more support for creating a formal specification such as stamping out formal specifications from design patterns or natural language support [BHS].

3 How to Verify JML Specifications with the KeY-Tool

Please activate *minimize interaction* in the `Options` menu as this will reduce the need for interactive steps.

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.

1. To verify, you first need to start the KeY prover. This is done by calling the `runProver` or `startProver` script of your KeY distribution, e.g. by running

   `bin/runProver` or `bin/startProver`

2. 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 name of the top level directory under which your Java sources are collected. In the tutorial example:

     `bin/runProver paycard` or `bin/startProver paycard` or

   - opening *File → Load* and selecting the top level directory under which your Java sources reside after having started `runProver` without any arguments. For the tutorial example this is again the `paycard` directory.

3. Now a window, the *JML specification browser*, pops up and you are asked to choose a property you would like to verify. The properties are specific to a method belonging to a class. So, in the browser, first a class of the project and, then, a method of that class must be selected. Do this by first opening the folder containing the loaded classes and then clicking with the mouse pointer on a class displayed in the left most pane. The middle pane of the browser then displays the methods of the selected class. A click on one of them shows—in the right pane—properties of the selected method that can

6

be verified with KeY. The properties are only available if there is a suitable specification for the selected method.

The next subsection will discuss the various kinds of proof obligations the KeY-Tool offers and will describe applications to the tutorial examples.

When you have selected a property, you can continue by clicking on the button *Load Proof Obligation*. Subsequently, the KeY-Tool generates a suitable proof obligation in terms of a logical formula. For verification, proof obligations will contain code of the target programming language (JAVA CARD, in our case). For these we use a Dynamic Logic[4] that is able to express properties of JAVA CARD programs.

Once you have loaded a proof obligation, you can always return to the JML specification browser by selecting *Tools → JML Specification Browser* and then load another proof obligation.

4. Proof obligations are loaded in the integrated interactive theorem prover KeY-Prover (see Section 1.4), which 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. 4.2, we show how to prove the proof obligations generated for the tutorial example.

4 Provable properties with JMLKeY

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 Appendix B. For further details on the mapping between JML specifications and the formulas 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 Section 4.2.

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 *fulfills* its method contract

- *properties for class specifications:* we show that a method *preserves* invariants or history constraints of a class.

4.1.1 Method Specifications

In JML, there are two kinds of method specifications: lightweight and heavyweight specifications. The main difference is that lightweight specifications set default values to be `not_specified` whereas heavyweight specifications have fixed "real" default values. Essentially we assume for `not_specified` values the least restricting value. As an exception to this we require, if not specified otherwise, that methods terminate, i.e. `diverges` has `false` as default value in lightweight specifications. Since the default values are then the same for (our treatment of) lightweight and

---

[4]Dynamic Logic can be seen as an extension of Hoare logic.

7

heavyweight specifications, this implies that KeY treats lightweight specifications like heavyweight specifications. Nested specifications are treated as being *desugared* correctly.

We distinguish between the following specification cases:

Normal Behavior Specification Cases. These specifications have the following general form:

```
normal_behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
```

To reduce the cognitive burden when proving interactively, the KeY-Tool has currently no proof obligation that checks for the complete correctness criteria imposed by JML (see Sect. A). Instead there are several lightweight proof obligations available. The first one is the one titled *normal_behavior speccase* in the right part of the JML specification browser. Informally this property checks:

if	the method is invoked in a pre-state that satisfies `P` and *the invariants of the considered class* (Sect.4.1.3) hold
then	if the method does not terminate (i.e. loops forever or aborts) **D** holds in the pre-state
or	the method terminates without throwing an exception **and** `Q` holds in the post-state

The generated proof obligation does *not* check for invariants in the visible states nor for assignable clauses and it does only assume invariants of the class the considered method is declared in. KeY always assumes that no Errors are thrown.

See below for the preservation of invariants and the checking of assignable clauses.

Exceptional Behavior Specification Cases. The general form is:

```
exceptional_behavior
  requires P;
  diverges D;
  assignable A;
  signals (Exception_1 e1) R1
  ...
  signals (Exception_n en) Rn
```

Again the generated proof obligation of the KeY-Tool does not check for all the conditions that JML requires (see Sect. A). Instead the following property (*exceptional_behavior speccase*) can be loaded:

if	the method is invoked in a pre-state that satisfies `P` and *the invariants of the considered class* (Sect.4.1.3) hold
then	if the method does not terminate (i.e. loops forever or aborts) **D** holds in the pre-state
or	the method completes abruptly by throwing an exception **and** if an instance of `Exception_i` ($i = 1, \ldots, n$) is the reason for abrupt completion, `Ri` holds in the post-state

Generic Behavior Specification Cases. The generic `behavior` specification case is the most general one and subsumes both aforementioned cases, that is, they may be *desugared* into a behavior specification case. The general form is:

```
behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
  signals (Exception_1 e1) R1
  ...
  signals (Exception_n en) Rn
```

The proof obligation generated by the KeY-Tool does not check for all the conditions imposed by this specification (Sect. A). Instead the following property (*behavior speccase*) can be loaded:

if		the method is invoked in a pre-state that satisfies `P` and
		the invariants of the considered class (Sect.4.1.3) hold
then		if the method does not terminate (i.e. loops forever or
		aborts) `D` holds in the pre-state
	or	the method terminates normally or completes abruptly by
		throwing an exception **and**
		`Q` holds in the post-state if the method returns normally
		if an instance of `Exception_i` ($i = 1, \dots, n$) is the reason
		for abrupt completion, `Ri` holds in the post-state

4.1.2 Type Specifications

The parts of type specifications in JML that the KeY-Tool checks are invariants and history constraints. Proofs are performed on a per (non-abstract) method basis. For every implemented method the KeY-Tool checks whether the method preserves the invariants and satisfies the history constraints imposed on the type the method is declared in.

The KeY-Tool offers a proof obligation, called *Class specification* generated for the following property:

if	the method is invoked in a pre-state that satisfies an arbitrary
	precondition of the method **and** *the invariants I of the considered*
	class (Sect.4.1.3) hold **and** the method terminates
then	*I* hold in the post-state
	and the history constraints of the considered class (Sect.4.1.3) hold
	in the post-state (w.r.t. the pre-state)

4.1.3 Proof Obligation Options

Invariants and Constraints Taken into Account The proof obligations above restricted invariants (and history constraints) to be assumed and established to those of the considered class. Correctly, the JML semantics requires to assume and / or establish *all applicable* invariants. Though this obviously leads to more complex proof obligations, it is possible to enforce the correct JML variant by checking the *Use All Applicable Invariants* checkbox at the bottom of the specification browser. By default the proof obligation described above is taken, as already advocated in the JML reference manual [LBR04].

Ensuring Invariants already in the Method Specification For method specifications it is usually not required to prove that the invariants and history con-

straints are established. Instead, this is done separately by using the *Class Specification* proof obligation. Alternatively you may select the option *Add Invariants to Postcondition* checkbox at the bottom of the specification browser to obtain a proof obligation that requires showing this property already when proving the method specification. This leads to more intricate proof obligations. Thus, by checking *Add Invariants to Postcondition* and *Use All Applicable Invariants* you get the most comprehensive proof obligation for a method that the KeY-Tool provides.

Integer Semantics   As already pointed out in Sect. 1.4, the KeY-Tool provides several alternatives for treating integer values. Which integer semantics you have chosen affects the generated proof obligations. A JML expression `a+b` may either be translated into a mathematical addition expression or into the Java `+` operation (with overflow), depending on the selection.

4.2   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 1.4), especially that the current strategy is *Java DL* and the maximum number of automatic rule applications is 1000. Be warned that the names of the proof rules and the structure of the proof obligations may be subject to changes in the future.

4.2.1   Method Specifications

Normal Behavior Specification Case.   In the left part of the JML specification browser, expand the directory `Paycard`. From the now available classes select `PayCard` and then the method `charge` in the second list. This method is specified by the JML annotation

```
public normal_behavior
  requires amount>0 ;
  assignable unsuccessfulOperations, balance;
  ensures balance >= \old(balance);
```

Choose the proof obligation *normal_behavior speccase for method charge* in the right list and press the button *Load Proof Obligation*. This property says that if the parameter `amount` is greater than zero and the invariants hold then a call to this method implementation terminates normally and the `balance` attribute is greater or equal to `balance` in the pre-state. 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 proof is closed automatically by the strategies. In older versions it was necessary to perform a few interactive proof steps or to run the decision procedure Simplify. The latter is done by pushing the button *Run Simplify*, which will succeed right away.

Exceptional Behavior Specification Case.   The exceptional behavior specification of the method `charge0` in the class `PayCardJunior` is

```
public exceptional_behavior
  requires amount <= 0 || checkSum(this.balance + amount) == 0;
  assignable \nothing;
```

```
    signals (CardException) amount <= 0
                            || checkSum(this.balance + amount) == 0;
```

The KeY proof obligation for this specification requires that if the parameter `amount` is negative or equal to 0 or if the pure method `checkSum` returns 0 with the argument `this.balance + amount` and the invariants of `PayCardJunior` hold, then the method throws a `CardException` in a state that satisfies the pre-condition.

Using the JML specification browser and selecting the property *exceptional_behavior* for `charge0` loads a proof obligation with a JavaDL formalization of this property.

As this specification contains a query to `checkSum`, you should set *Query Treatment* to *Expand*, or expand the query interactively. 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 `createCard` in `PayCardJunior` 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 `PayCardJunior`. By selecting the `createCard` method and then *behavior speccase 0 for method createCard*, an appropriate JavaDL formula is loaded in the prover. The proof can be closed automatically by the strategy *Java DL*.

4.2.2 Type Specifications

The instance invariant of type `PayCardJunior` is

```
  this.balance >= 0
  && this.balance < juniorLimit && juniorLimit < limit;
```

There is also a static invariant available:

```
  juniorLimit==100;
```

The method `charge` of `PayCardJunior` must preserve these invariants unless it does not terminate. The KeY proof obligation to check this property is called *Class specification for class PayCardJunior*.

The proof is done automatically with the strategy *Java DL*.

4.2.3 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.

Open the *normal_behavior speccase* of `LogFile`'s `getMaximumRecord()` with the JML specification browser. Choose the strategy *Java DL* and the *Loop treatment None*. Make sure *Autoresume strategy* is selected and start the prover. Now select the while loop including the leading updates and execute the rule *unwind_while*. Several goals remain after the strategies did most of the work, of which all but two can be closed by running *Simplify*. In both goals you will find a formula similar to `self_LogFile_lv_0.logArray[i_0] = null` respectively `self_LogFile_lv_0.logArray[0] = null` and all-quantified formula similar to

```
\forall int index_lv0;
  !index_lv0 <  0 & index_lv0 < paycard.LogFile.logFileSize ->
    !self_LogFile_lv_0.logArray[index_lv0] = null
```

and

```
\forall jint j;
  (j >= 0 & j <= -1 + i_0 ->
    self_LogFile_lv_0.logArray[j].balance <= max_0.balance)
```

Highlight the term i_0 (resp. 0) with the mouse, and drag and drop it onto an all-quantified formula (the complete all-quantified formula must be highlighted (incl. `\forall`)) to create an instance of the all-quantified formula with the given variable. If you have instanciated the formula with the correct variable you can close the proof now. This should happen automatically, if `Autoresume` is activated, otherwise you need to restart the strategies.

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. We refer to future work on this issue.

4.2.4   Using the KeY-plugin for Eclipse

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   Current Limitations and Restrictions

The current version of the KeY-Tool is far from being a polished and universally applicable tool. Especially the JML interface, JMLKeY, is very much work in progress. Moreover the JML semantics are still subject to discussions, not to mention that there is no formal semantics for JML. Thus, bugs and inadequacies between the original JML semantics and the (implicitly given) semantics in KeY are be possible.

Concerning KeY, here is an (incomplete) list of open issues we are now working on and intend to resolve in near future:

1. Restrictions on JML translation:

   - default variable initialization is null.
   - currently not completely functional in all cases : translation of model methods and model fields. Fix is pending.
   - unsupported constructs, support planned: data groups, assignable clauses containing `*`, `\reach`, asserts, `\fresh`, `\not_modified`,
   - unsupported constructs, future support possible:
     - Accessible, Callable clauses
     - `\is_initialized`, `\num_of` `\product`, `\sum` expressions.

- unsupported constructs, support in KeY not planned:
    - When, Working Space, Duration, Measured By clauses and model programs
    - `\duration`, `\space`, `\working_space`, `\invariant_for`, `\type`, `\typeof`, `\elemtype` expressions.

2. 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.

3. Restrictions of the KeY-Prover:

    - manual not yet available
    - automatic insertion of quantifiers is pending

4. Restrictions on JDK:

    - problem: tool tips are flickering occasionally
      workaround: reduce the number of tool tip lines in the menu **View**

A   Informal Semantics of JML Method Specifications

For the convenience of the reader a brief summary of the informal semantics of JML specification cases as described in [LPC+02] can be found below.

Normal Behavior Specification Cases.   Given the following method specification

```
normal_behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
```

JML requires for this specification that

if		the method is invoked in a pre-state that satisfies `P` and all applicable invariants hold
then		an error is thrown
	or	if the method does not terminate (i.e. loops forever or aborts) `D` holds in the pre-state
	or	the method terminates without throwing an exception **and** during execution of the method the modified locations already existing in the pre-state and not-local to the method are listed in `A` **and** all applicable invariants and history constraints hold in all visible states **and** `Q` holds in the post-state

Exceptional Behavior Specification Cases.   The general form is:

```
exceptional_behavior
  requires P;
  diverges D;
  assignable A;
  signals (Exception_1 e1) R1
  ...
  signals (Exception_n en) Rn
```

JML requires for such a specification that

if		the method is invoked in a pre-state that satisfies `P` and all applicable invariants hold
then		an error is thrown
	or	if the method does not terminate (i.e. loops forever or aborts) `D` holds in the pre-state
	or	the method completes abruptly by throwing an exception **and** the modified locations already existing in the pre-state and not-local to the method are listed in `A` **and** all applicable invariants and history constraints hold in all visible states **and** if an instance of `Exception_i` ($i = 1, \ldots, n$) is the reason for the abrupt completion, `Ri` holds in the post-state

Generic Behavior Specification Cases.   The general form is:

```
behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
  signals (Exception_1 e1) R1
  ...
  signals (Exception_n en) Rn
```

JML requires for such a specification that

if		the method is invoked in a pre-state that satisfies `P` and all applicable invariants hold
then		an error is thrown
	or	if the method does not terminate (i.e. loops forever or aborts) **D** holds in the pre-state
	or	the method terminates normally or completes abruptly by throwing an exception **and**
		the modified locations already existing in the pre-state and not-local to the method are listed in **A and**
		all applicable invariants and history constraints hold in all visible states **and**
		`Q` holds in the post-state if the method returns normally
		if an instance of `Exception_i` ($i = 1, \ldots, n$) is the reason for abrupt completion `Ri` holds in the post-state

B   Formal Description of Generated Proof Obligations

Proof obligations generated from JML specifications are always generated with respect to a certain method m in a class CC and express that m satisfies certain correctness aspects of the regarded specification. For the proof obligations discussed in this section we will use the following abbreviations:

- `MBS(m)` is an abbreviation for the body of method m (in the selected class CC), *appropriately* instantiated with receiver and parameter variables, which depends on the signature of m.

- α stands for the following Java block:

```
java.lang.Exception exc = null;
try {
    MBS(m)
} catch (java.lang.Exception e) {
    exc = e;
}
```

For readers not familiar with JavaDL we note that JavaDL treats abrupt termination by any exception or error as *non-termination*. Therefore it is necessary to use the program α as defined above. Furthermore, JavaDL has an operator called *update*, written $\{loc := val\}$, for locations loc and side-effect free expressions val, which makes the formula behind it be interpreted in a state where the assignment `loc=val;` has been executed. Finally we mention that the initialisation of classes and objects is realized by virtual attributes, as for instance $<classInitialized>$ or $<created>$.

B.1 Proof Obligations for Method Specification Cases

The proof obligations for method specification cases express that a method satisfies the (exceptional and normal) postcondition of a certain specification case if it is called in a state that implies the corresponding precondition P and that it terminates if it is called in a state that implies $P \wedge \neg D$, where D is the JavaDL counterpart of the expression specified by the *diverges* clause.

B.1.1 Normal Behavior Specification Cases

Given the following method specification

```
normal_behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
```

the corresponding proof obligation is:

$$\forall self{:}CC \ (P \wedge inv \rightarrow (((\neg D \rightarrow \langle \alpha \rangle true) \wedge [\alpha]Q')))$$

- P and D are the JavaDL representations of P and D.

- We use inv to denote the invariants used in the proof obligation. It is defined as follows:

    1. if *Use All Applicable Invariants* is checked:

    $$inv := \bigwedge_{T:T \in \mathcal{T}} \forall o{:}T \ \Big(\{self_T := o\} \\ \big(((o \neq null \wedge o. {<}created{>} = true) \rightarrow inv_o) \wedge \\ (T. {<}classInitialized{>} \rightarrow inv_T)) \big) \Big)$$

        - $\mathcal{T}$ is the set of known classes and interfaces.
        - inv_o is a conjunction of the invariants applicable to o, i.e. the instance invariants of T.
        - inv_T are the static invariants of T.
        - $self_T$ is a program variable of type T introduced during the translation of the specification of T.

    2. if *Use All Applicable Invariants* is not checked and m is nonstatic:

    $$inv := inv_{self} \wedge inv_{CC}$$

    or if m is static:

    $$inv := inv_{CC}$$

    where

    - inv_{CC} are the static invariants of CC.
    - inv_{self} is a conjunction of the invariants applicable to $self$, i.e. the instance invariants of CC.

    where

- Let Q be the JavaDL representation of Q. Q' equals Q if the option *Add Invariants to Postcondition* is checked and it equals $Q \wedge inv$ otherwise.

B.1.2 Exceptional Behavior Specification Cases.

```
exceptional_behavior
  requires P;
  diverges D;
  assignable A;
  signals (Exception_1 e1) R1
  ...
  signals (Exception_n en) Rn
```

With P, D as above, the proof obligation for this specification is:

$$\forall self{:}CC \ (P \wedge inv \rightarrow ((\neg D \rightarrow \langle\alpha\rangle true) \wedge [\alpha](\texttt{exc} \neq \texttt{null} \wedge E \wedge I)))$$

- E stands for:

$$\bigwedge_{i=1}^{n} (\texttt{exc } instanceof \ \texttt{Exception\_}i) \rightarrow \{e_i := \texttt{exc}\}R_i.$$

  and R_i $(i = 1, \ldots, n)$ is the translation of the expression `Ri`.

- If *Add Invariants to Postcondition* is selected I equals *true* (i.e. is skipped from the conjunction) and, otherwise, equals *inv*.

B.1.3 Generic Behavior Specification Cases.

```
behavior
  requires P;
  diverges D;
  assignable A;
  ensures Q;
  signals (Exception_1 e1) R1
  ...
  signals (Exception_n en) Rn
```

The proof obligation for this specification is:

$$\forall self{:}CC \ (P \wedge inv \rightarrow (\ (\neg D \rightarrow \langle\alpha\rangle true)$$
$$\wedge [\alpha]((\texttt{exc} = \texttt{null} \rightarrow Q) \wedge (\texttt{exc} \neq \texttt{null} \rightarrow E)) \wedge I))$$

All abbreviations are defined as in the cases above.

B.1.4 Proof Obligations without Diverges

Usually it is required that a method must terminate, i.e. D is `false`. This is (in KeY) also the default value if the *diverges* clause of a method specification is missing. In this case we get much simpler proof obligations. For instance, for a behavior specification case, we get (using the abbreviations as above):

$$\forall self{:}CC \ (((P \wedge inv) \rightarrow \langle\alpha\rangle((\texttt{exc} = \texttt{null} \rightarrow Q) \wedge (\texttt{exc} \neq \texttt{null} \rightarrow E)))))$$

The proof obligations for *normal* and *exceptional behavior* specification cases with omitted *diverges* clauses are simplified accordingly.

B.2 Proof Obligations for Invariants and History Constraints

Let $\bigvee_{i=1}^{n} P_i$ be the disjunction of the preconditions of the specification cases of m and inv_{pre}, inv_{post} formulas representing the *invariants* one wants to prove (depending on the selected options). *con* stands for the applicable *history constraints* of CC. Then the proof obligation for a type specification is:

$$\forall self{:}CC \ ((\bigvee_{i=1}^{n} P_i) \wedge inv_{pre} \rightarrow [\alpha]con \wedge inv_{post})$$

We usually have $inv_{pre} = inv_{post}$ except for the case that *Use All Applicable Invariants* is not checked and the regarded method is a constructor ($inv_{pre} = inc_{CC}$ and $inv_{post} = inc_{CC} \wedge inv_{self}$) or the implicit method $<clinit>$ ($inv_{pre} = true$ and $inv_{post} = inc_{CC}$).

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, 2001.

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

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

[ICS] `www.icansolve.com`.

[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 `www.jmlspecs.org`.

[LPC+02] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. Jml reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, August 2002.

[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`.

[Sim] `http://secure.ucd.ie/products/opensource/ESCJava2/`.