KeY Quicktour

Thomas Baar
University of Karlsruhe
Dept. of Computer Science
D-76128 Karlsruhe
baar@ira.uka.de

Reiner Hähnle
Chalmers University of Technology
Dept. of Computing Science
S-41296 Gothenburg
reiner@cs.chalmers.se

Steffen Schlager
University of Karlsruhe
Dept. of Computer Science
D-76128 Karlsruhe
schlager@ira.uka.de

Sonja Lauer
University of Karlsruhe
Dept. of Computer Science
D-76128 Karlsruhe
lauer@ira.uka.de

Contents

1 Introduction/Prerequisites **2**
1.1 Version Information 2
1.2 Logical Foundations 2
1.3 The KeY-Prover 2

2 Tutorial Example **6**

3 Creating a Formal Specification in the OCL **7**
3.1 The Basic Idea 7
3.2 Application in the Tutorial Example 8
3.3 Constraints in Natural Language and OCL 9

4 How to Parse a Specification **11**
4.1 Application in the Tutorial Example 12

5 How to Analyse/Verify a Specification **12**
5.1 Informal Description of Options for Analysis and Specification . . . 13
5.2 Application in the Tutorial Example 15

6 Current Limitations and Restrictions **17**

A Formal Description of Generated Proof Obligations **19**
A.1 Options Offered in the Class Menu 19
A.2 Options Offered in the Method Menu 19

1 Introduction/Prerequisites

This document constitutes a tutorial introduction to the KeY-Tool. The KeY-Tool is an integrated environment for creating, analysing, and verifying UML/OCL models and their implementation. The main focus of the KeY-Tool are class diagrams. Other kinds of diagrams are currently not supported yet.

The KeY-Tool is an extension of the commercial CASE tool TOGETHER CONTROLCENTER[1] (in the following referred to as TOGETHERCC). We assume that the reader is familiar with the CASE tool TOGETHERCC. Here we concentrate on the description of the KeY extensions. Furthermore, we assume that the KeY-Tool has been already installed successfully.

The KeY-Tool is designed as an add-on to TOGETHERCC. Thus, all features offered by TOGETHERCC are available and the user can work with a powerful UML CASE tool in a familiar environment. The design philosophy of the KeY-Tool is to encourage but not to force users to take advantage of formal methods. Users are able to decide themselves at which point the KeY extensions are useful.

For a longer discussion on the architecture, design philosophy, and theoretical underpinnings of the KeY-Tool please refer to [3].

The most recent version of the KeY-Tool can be downloaded from `http://download.key-project.org`.

1.1 Version Information

This tutorial was tested for TOGETHERCC version 6.2.

1.2 Logical Foundations

Deduction with the KeY-Prover is based on a sequent calculus for a Dynamic Logic for JavaCard (JavaDL) [4]. 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 $(\phi_1 \land \ldots \land \phi_m) \rightarrow (\psi_1 \lor \ldots \lor \psi_n)$ $(m, n \geq 0)$.

1.3 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.[2]

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

- **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 all the rules available in the system are indicated. KeY distincts between *axiomatic taclets* (rules that are always true in the given logic), *lemmas* (that are derived from and thus provable by axiomatic

[1] To obtain the tool TOGETHERCC please contact `http://www.borland.com/together/`. A free time-restricted trial version is available.

[2] 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*.

taclets) and *built-in rules* (for example how certain expressions can be simplified).

By doubleclicking on a rule of the list, a window comes up where the corresponding rule is explained.

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 is a leaf). Leaf nodes of an open proof branch are coloured red whereas leaves of closed branches are coloured 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.

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: 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 menu item *Proof* → *Strategy*. A dialog pops up where you can 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 *Simple JavaCardDL*, as in the sequel of this quicktour. For pure logic problems use the strategy *Simple FOL*. Furthermore, you can set the maximum number of automatic rule applications. If you want to save your settings (chosen strategy and maximum number of rule applications) for further proofs push the button *Save as Default*. To save them only for the current proof just push the *OK* - button. To start (respectively continue) the proof push the *run strategy*-button on the toolbar labelled with the ▷ - symbol. If the checkbox *Autoresume strategy* is selected, the prover automatically resumes applying the strategy after an interactive rule application.

Another way to define the strategy that should be used during the current proof is to click on the field right to the *run strategy*-button. In this field

the current strategy is shown. After clicking on it, a list of all available strategies comes up from which you can select one. By moving the blue arrow to the left or to the right you can also set the maximum number of automatic rule applications.

In the following we describe some menu items available in the main menu of the KeY-Prover. In this quicktour we will confine on 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 → Pretty&Untrue: This menu item allows you to toggle between two different syntax representations. If checked a nicer and easier to read syntax is used. Formerly the pretty syntax was not parseable (that's why **Untrue**), but this does not longer hold.

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

View → 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. What is behind all this goes beyond the scope of this quicktour. Please use the default settings unless you know what you are doing.

transactionsPolicy: Specifies how to handle the JavaCard Transactions (abortTransaction).

programRules: Changes between different program languages (Java)[3].

initialisation: Specifies if static initialisation should be considered or not (disableStaticInitialisation).

intRules: Here you can choose between different semantics for Java integer arithmetic (for details see [6]). 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.
- Arithmetic semantics ignoring overflow (default): Treats the primitive finite Java types as if they had the same semantics as mathematical integers with infinite range.
- 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.

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*.

[3]Ensure that *Java* is selected.

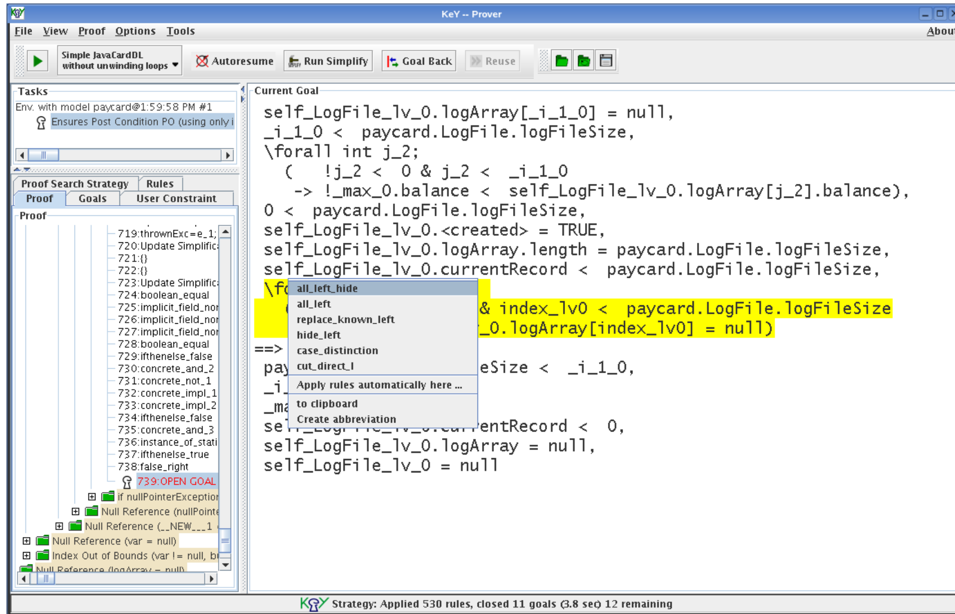

Figure 1: The KeY-Prover.

Options → Update Simplifier: Here you can define policies how updates should be simplified. As 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 the two different integer decision procedures *Simplify*[4] [1] and *ICS* [2]. During this quicktour, the procedure *Simplify* should be selected.

Options → Compute Specification: Here you can choose between different settings 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 → Outer renaming: If this checkbox is selected, name clashes are resolved by renaming the variables outside of the modality. .

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 menu *Options* permanently, i.e. for future sessions with the KeY-Prover.

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

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

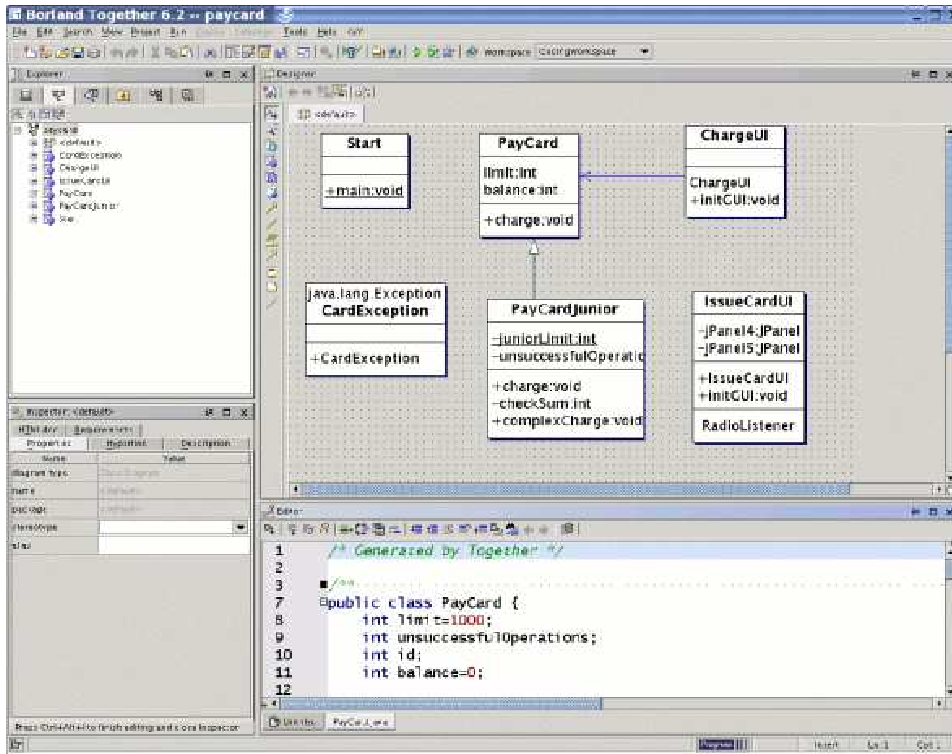

Figure 2: Class Structure of Tutorial Example

2 Tutorial Example

In this tutorial we use a simple paycard application (project `paycardInteractive`) to illustrate some basic capabilities offered by the KeY-Tool. The tutorial example is a standard TogetherCC project (contained in the file *paycard.tpr*) and can reside anywhere in the file system of your computer. After opening the project you can inspect the class structure of the project as depicted in Figure 2 (select tab *<default>*, if necessary).

The class diagram shown in Figure 2 consists of the six classes `PayCard`, `PayCard-Junior`, `CardException`, `ChargeUI`, `IssueCardUI`, and `Start`. The class `Start` provides the `main` method of the application. You can compile and execute the application from within TogetherCC by selecting the menu item *Run* → *Run* or by using the function key *F9*. Try this now. TogetherCC first compiles the Java source code and immediately executes it afterwards. If TogetherCC reports errors during compilation one reason could be a wrong setting in project options. Please change in the project options (*Tools* → *Options* → *Project Level*) the value of *Builder* → *Built-in Javac* → *Compiler Options* → *Destination directory* to `$PROJECT_DIR$` and try again.

The tutorial example is a simple paycard scenario. Running the application, in the first dialog the customer (user of the application) can 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 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 static structure of the example application is modelled in the class diagram. The intended semantics of some classes is defined with the help of invariants denoted in the Object Constraint Language (OCL). Likewise, the behaviour of most methods is described in form of pre-/postconditions in the OCL.

3 Creating a Formal Specification in the OCL

Rigorous specification is a necessary prerequisite to discuss the "correctness" of a UML model and its implementation in a meaningful way. This is a considerable obstacle, in particular, for novice users in formal methods. The KeY-Tool helps users to come up with meaningful requirement specifications in the OCL.

3.1 The Basic Idea

Probably only few software developers feel happy when faced with the task of writing a specification in a formal language like the OCL. Many developers are not familiar with that kind of activity and refuse to learn how to write formal constraints for the system they intend to build. The situation is not helped by the fact that most CASE tools treat formal constraints just as a kind of comment. In practice, requirement specifications are mostly written in natural language, with all its ambiguities. Formal specification languages are rarely, if ever, used.

For the user of the KeY-Tool the situation is different. Since the KeY-Tool can analyse and give feedback on OCL constraints (see Section 5) they are actually and immediately useful. Hence, the user has a new motivation to formulate constraints in a formal language. But the KeY-Tool can even support the user in generating formal specifications in the first place. The technology behind this is a template-like and easy-to-understand mechanism. Consider, for example, the behavioural specification of class `PayCard` where we require that the value of the attribute `balance` is always greater or equal zero and less than `limit`. Such requirements where the value of an attribute `attr` of a class `aClass` has to be within a certain interval occur quite often. The specification of such a requirement has the following form in general:

```
context aClass:
       inv:  lowerBound < attr and attr < upperBound
```

There is a plethora of similar constraints needed in related situations (for example `AttributeHasKeyProp`, and, as examples for pre-/postconditions, `ProduceFor-AssociationSet`, `GetFromAssociationSet`, and `IncreaseAttribute`). The KeY-Tool contains predefined blueprints (or templates) of such constraints which we call *KeY-Idioms*.

In addition to KeY-Idioms there is a slightly more complicated way to generate a specification, called *KeY-Pattern*. Again, the basic idea is to use blueprints. In contrast to KeY-Idioms, where the blueprints are merely attached to a single class or method, they are now attached to OO design patterns like `Composite`, `Observer`, etc. The KeY-Patterns can be used in the same way as the other design patterns that are available in TogetherCC.

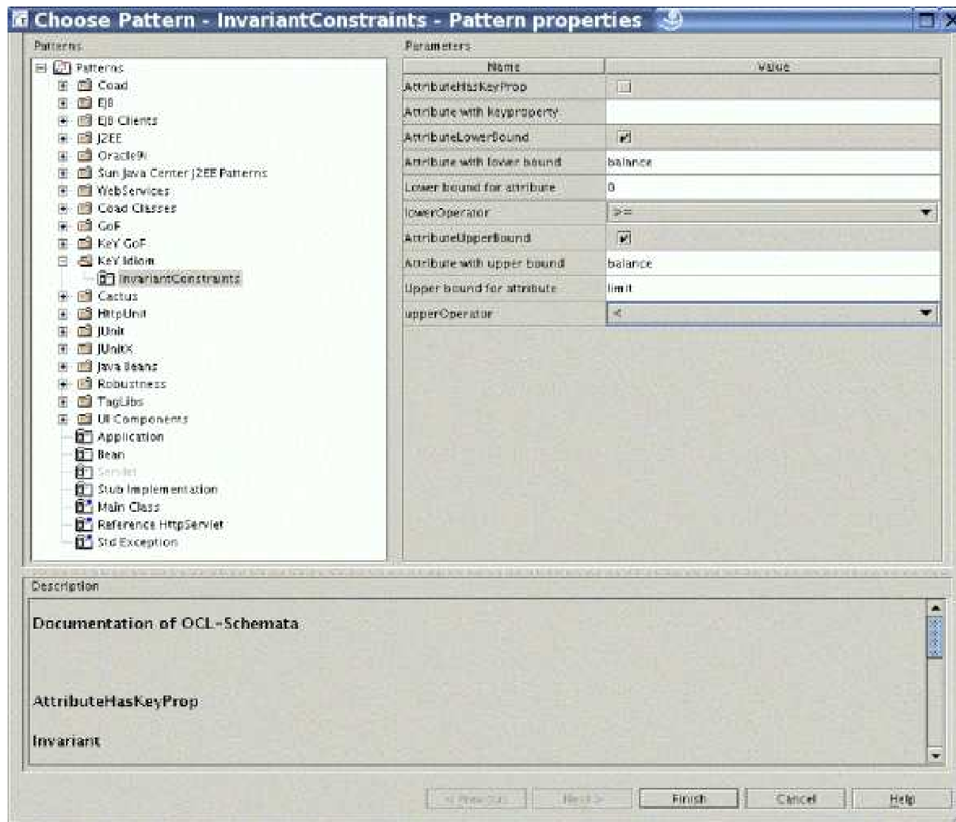

Figure 3: Generation of OCL Expressions.

Each KeY-Pattern contains a set of blueprints that are selected and instantiated by the user during a customisation dialog. As it is the case for standard patterns, TogetherCC generates a concrete design after finishing the dialog. In addition, concrete OCL constraints are generated as instances of OCL blueprints.

3.2 Application in the Tutorial Example

We demonstrate how to generate a specification for the class `PayCard`, i.e. an invariant, which states that the value of the attribute `balance` is always greater or equal zero and less than `limit`. First, make sure that a specification for this class does not already exist. Should the occasion arise, please delete it from the sourcecode. Now, use the mouse to select class `PayCard` in the UML diagram and push the right mouse button to get a pop-up context menu. Please select the menu item *Choose Pattern...*. A pattern selection dialog lists all predefined patterns and you should now select *KeY Idiom* → *InvariantConstraints*, which lists a number of available OCL blueprints for class invariant specification.

Please choose the blueprints `AttributeLowerBound` and `AttributeUpperBound` by clicking on the according checkboxes and fill in the required slots: “`Attribute with lower bound`” should be “`balance`”, “`Lower bound for attribute`” should be “`0`”, “`lowerOperator`” should be “$\geq$”, “`Attribute with upper bound`” should be “`balance`”, “`Upper bound for attribute`” should be “`limit`”, and “`upperOperator`” should be “$<$” (see Figure 3)[5]. After pushing the *Finish* but-

[5] The order of the parameters may differ, don't mind if that is the case.

ton, the pattern dialog disappears and the intended specification is generated and added as a comment to the sourcecode of class `PayCard`. As this example shows, it is possible to select several blueprints simultaneously. Then, the resulting OCL specification is the conjunction of the instantiated OCL blueprints.

In a similar way, the blueprints of the KeY-Patterns are instantiated. Note, that in case of KeY-Patterns the relevant part of the current class diagram is generated and selecting a class prior to invocation of the dialog is not necessary.

3.3 Constraints in Natural Language and OCL

This chapter is currently not up-to-date as we are migrating to a new GF version, which leads to an improved natural language rendering (see `http://www.cs.chalmers.se/~krijo/gfspec/`).

A tool for simultaneous development of natural language and OCL constraints is currently being integrated into the KeY-Tool. It consists of a syntax directed editor for constraints. Here, we will see an example of how to use this syntax editor to construct a simple invariant.

At the current level of integration, the syntax editor can be started from the context menu of classes and methods in TogetherCC, for the editing of invariants or pre- and postconditions, respectively. This tool is work in progress, we refer to the (forthcoming) manual for important details and limitations which we omit here.

The basic idea of the editor is that the user constructs an abstract syntax tree of a specification (for instance, an invariant of a class), by selecting alternatives from menus. The syntax tree is at all times presented in both OCL and English to the user. Since the editing always takes place by selection from menus, the editor can ensure that only syntactically correct specifications are constructed. Type correctness is also ensured.

Creating a New Class Invariant for PayCard: First, delete any previously added invariant for the class `PayCard` from the sourcecode. Then, just right-click on the class in the "Designer" pane of TogetherCC, and select *Edit Invariant [GF]* from the *KeY* part of the context menu which appears.

The syntax editor will now start. This usually takes a little while: the editor window might appear quickly, but it is not ready for input until some text has appeared in the three main areas of the window (which are blank to start with). Figure 4 shows what the editor window will look like when it is ready for input.

The Editor Window: The editor window consists of three main parts. The upper left part shows an abstract syntax tree of the invariant we are editing. The upper right part also shows the abstract syntax tree (as a string), but also the rendering of the abstract syntax into OCL and English. Unfinished parts of the invariant are shown as question marks (also referred to as *metavariables*)—these mark spots which have yet to be filled in by the user. The current metavariable is highlighted. When we start editing a new invariant, the OCL and English parts are empty (i.e. just a "?" is shown). The abstract representation (the first few lines in the upper right window) contains some information which is not explicit in English or OCL (for instance that the class of the invariant we are editing is `PayCard`), and is therefore not completely empty.

Editing proceeds by filling in question marks which is done by selecting *refinements* from the menu in the lower half of the editor window. The "r" in each list item just stands for "refine", what comes after the "r" is the name of the refinement (from the abstract syntax). To select a refinement, just double-click on it. The current metavariable (question mark) will then be filled in.

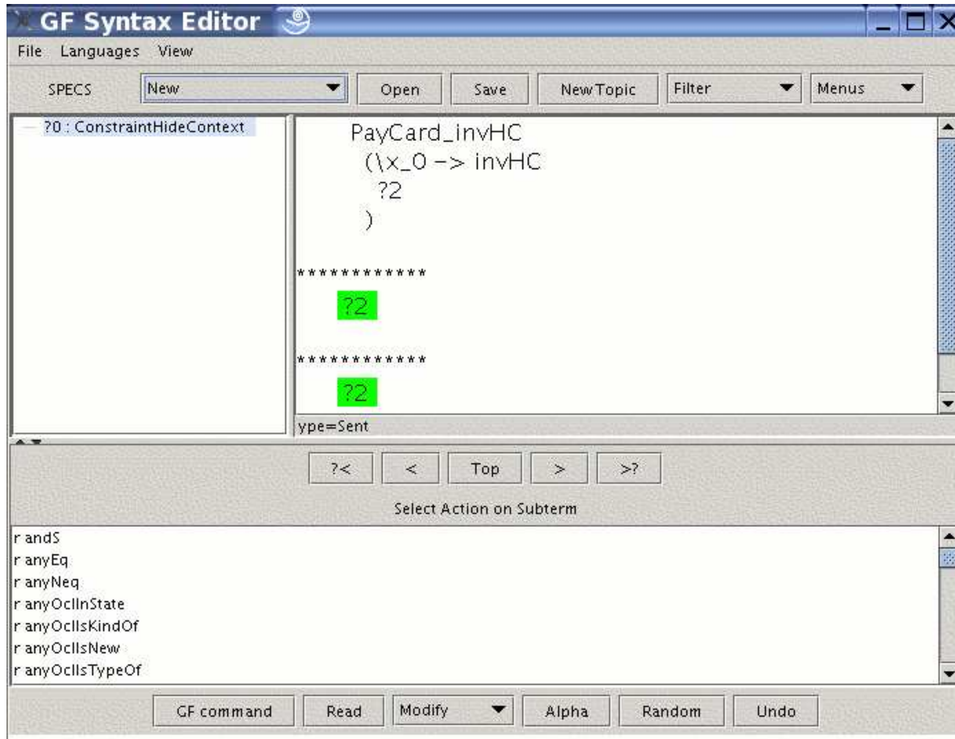

Figure 4: Editing a new invariant in the syntax editor

Choosing Refinements: We use an even simpler example than above: we will add the invariant that the balance of a `PayCard` is always greater than or equal to zero. The first step would be to find a refinement corresponding to the greater-than-or-equal relation for integers. To do this, we need to know that there are ways to make the list of refinements more informative than just showing a name from the abstract syntax. We can choose to show refinements in abstract syntax or in English[6], and we can choose to show type information or not.[7] These choices are made using the *Menus* menu in the upper right corner of the editor window. In this menu, *plain* and *printname* refers to abstract syntax and English, respectively. The alternatives *typed* and *untyped* refers to type information.

To find a suitable refinement, we can scroll through the list of refinements in the lower half of the editor window, possibly switching between abstract syntax and English, or typed and untyped presentation using the *Menus* menu. Eventually, we should find the refinement **`intGTE`**, "? is greater than or equal to ?" in English. The type for **`intGTE`** is `Instance Integer` → `Instance Integer` → **`Sent`**, i.e. it takes two instances of a class of integers and creates a sentence. Figure 5 shows the editor after the selection of this refinement (by double-clicking).

Navigation: If there is more than one metavariable (as in Figure 5), we can fill them in in any order. The button bar in the middle of the editor window makes it possible to navigate the syntax tree. For instance, the buttons marked "?<" and ">?" are used to step back and forward among the metavariables. The list

[6]This is a current limitation, the user should be able to see refinements in abstract syntax, in OCL, or in English.

[7]A current bug is that showing refinements in English and also with type information results in a somewhat garbled display, where the type information partly overlaps the English text.

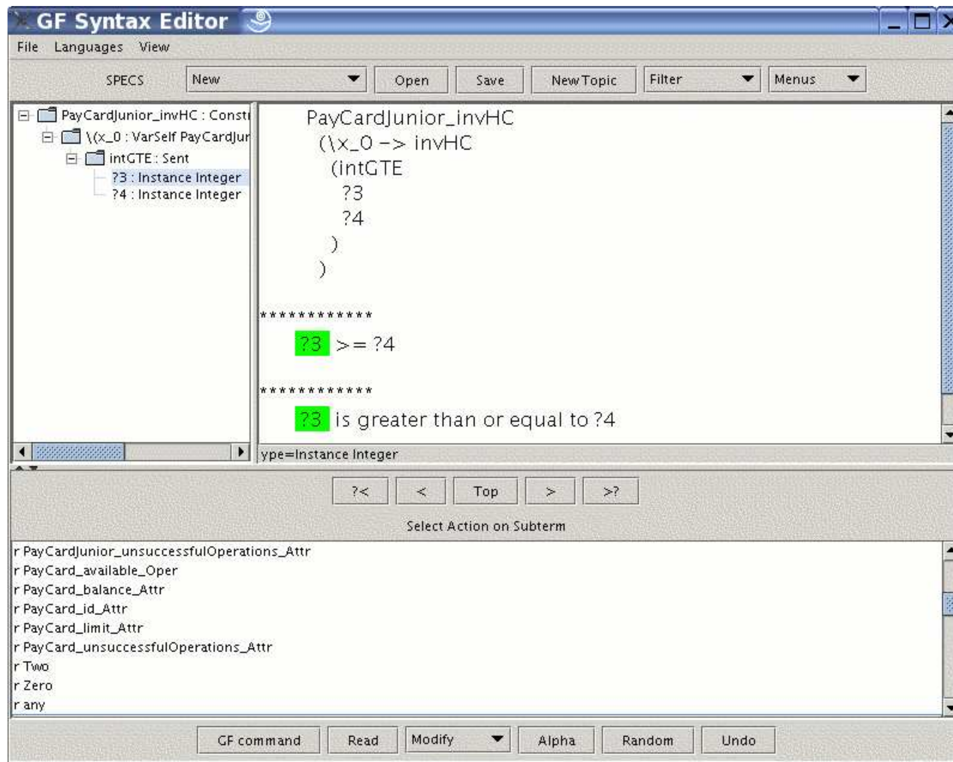

Figure 5: The first refinement step

of refinements always refers to possible ways of filling in the current metavariable (which is highlighted).

The left metavariable in the example can be filled in by choosing the refinement `PayCard_balance_Attr` and then `self`. For the right one we can simply choose the refinement `Zero`.

Finishing Up: Editing proceeds until there are no more metavariables to fill in. In the case of the example, we end up with the OCL constraint `self.balance >= 0`, and the English rendering "the balance of the payCard is greater than or equal to zero". We can then just close the editor window, and the invariant will appear in the sourcecode of class `PayCard` as well as in the tab "Properties" of the "Inspector" pane in TogetherCC.

4 How to Parse a Specification

We are now ready to take a closer look on the ways how to make use of OCL constraints in model analysis and verification of correctness properties.

A specification consists of OCL expressions for invariants of classes and for pre-/postconditions of methods. Of course, OCL expressions can only live in the context of a UML diagram (and therefore UML diagrams and even the implementation in a target language are also part of the specification), but we concentrate on OCL expressions for now.

The first step is to ensure syntactical correctness of OCL expressions. The KeY-Tool features an integrated OCL parser which can be invoked via a menu item in the

context menu. The currently used parser was developed at Dresden University of Technology (see http://dresden-ocl.sourceforge.net/index.html for details). It can also be used as a stand-alone system.

4.1 Application in the Tutorial Example

To parse OCL constraints, the KeY-Tool offers menu items *ParseInvariant* and *ParseMethodSpec*, respectively, as part of the context menus of classes and methods (to open it push the right mouse button on a class respectively a method and then select *KeY* on the shown pop-up menu).

As an example let us invoke *ParseInvariant* in class `PayCard` and the parser will tell you that the invariant `(balance ≥ 0) and (balance < limit)` is syntactically well-formed. Try to modify the invariant into a syntactically incorrect OCL expression (say, by misspelling `balance` as `ballance`). The parser points to the position, where the error occurred.

Please try also to invoke *ParseMethodSpec*, e.g., in class `PayCard` on method `charge`.

5 How to Analyse/Verify a Specification

Analysis: OCL constraints make the semantics of a class diagram more precise. A minimal requirement that must be fulfilled by these constraints is that it is actually possible for a model/implementation to satisfy them. In other words, OCL constraints must be consistent or free of contradictions. The KeY-Tool includes functionality to *analyse* the constraints.

Verification: OCL constraints, 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*[8] the correctness of an implementation with respect to its specification.

In each case, the KeY-Tool generates suitable proof obligations in terms of logical formulas. When *analysing* a specification no code needs to be considered, hence the resulting proof obligations are formulas of sorted first-order predicate logic. On the other hand, if the correctness of an implementation has to be verified, proof obligations will contain code of the target programming language (Java Card, in our case). For these we use a Dynamic Logic[9] that is able to express properties of Java Card programs.

In both cases, proof obligations are passed to the integrated interactive theorem prover KeY-Prover (see Section 1.3), 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 the following the ideas behind the various options for analysis and verification are described informally. A formal description of the generated proof obligations is contained in Appendix A. Examples of application within the context of the case study in this tutorial are described in Section 5.2.

[8]Sometimes *analysis* of a specification is called *horizontal verification* and what we call *verification* is called *vertical verification*.

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

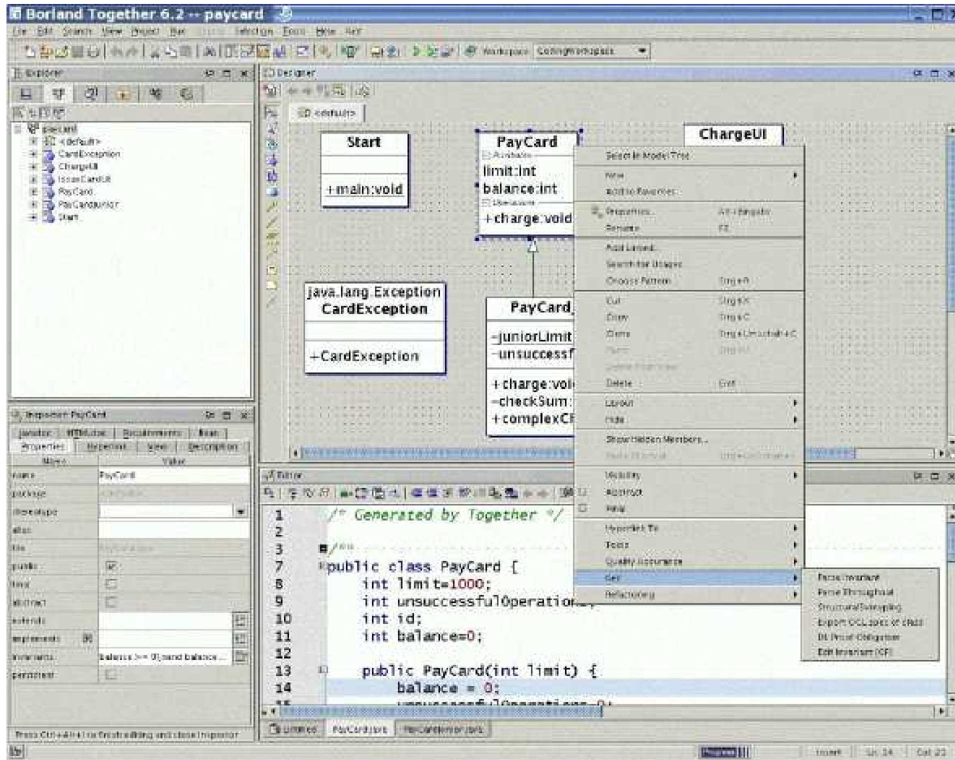

Figure 6: Options offered by the Class Menu.

5.1 Informal Description of Options for Analysis and Specification

All options can be invoked via context-sensitive menu items. Therefore, every option "knows" the model element it is applied to: in case of a class menu item this is the current class and in case of a method menu item this is the current method in the current class.

The proof obligations generated by invoking one of the options are derived from the invariant, the pre- and postconditions, and possibly the target code attached to the current model element.

Both, invariants and pre-/postconditions can be empty. In this case, the KeY-Tool assumes them to be `true` by default. Note that this differs from some other approaches. In Eiffel, for example, invariants are "inherited" from the parent class (see [5]).

In the following, options whose proof obligations are formulated in predicate logic are marked with (PL) and proof obligations formulated in Dynamic Logic are marked with (DL).

5.1.1 Options Offered in the Class Menu

Figure 6 shows the options offered in the class context menu (to open the menu select a class and push the right mouse button, then select *KeY*).

StructuralSubtyping (PL): Structural subtyping is one aspect of Liskov's substitution principle, namely that objects of classes inherited from class C may be used in place of objects from class C itself. The principle implies that an

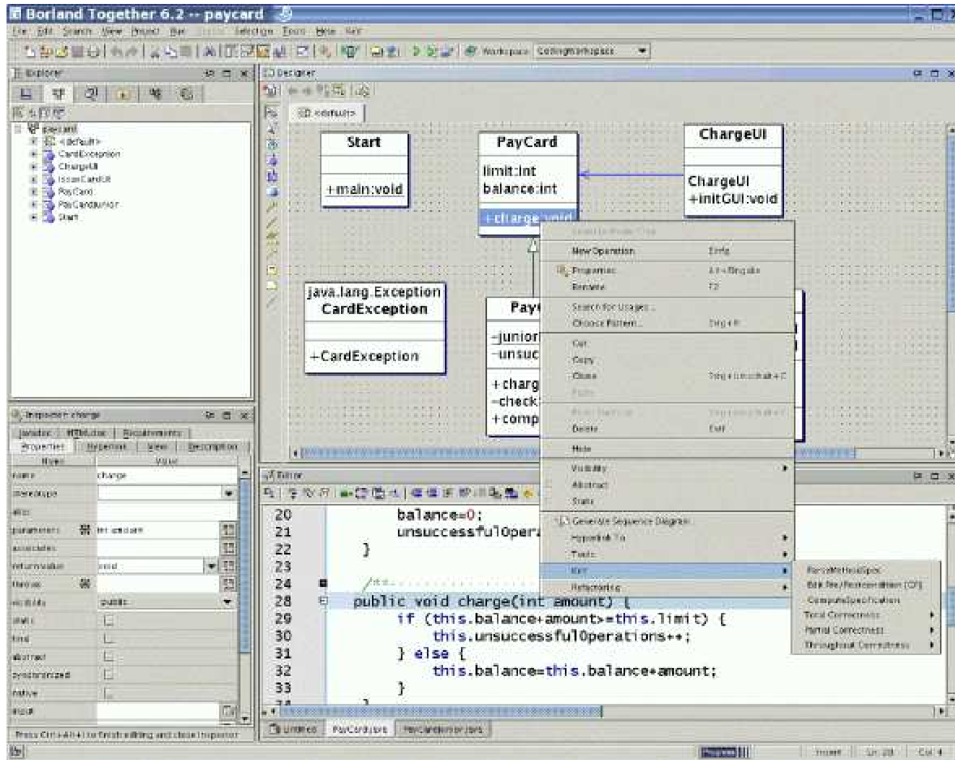

Figure 7: Options offered by the Method Menu.

object of current class CC must satisfy all constraints declared in all parent classes of CC.

In particular, the generated proof obligation ensures that the invariant (that is the structural aspect) of the current class CC is logically stronger than the one in the immediate parent class.

5.1.2 Options Offered in the Method Menu

Figure 7 shows the options offered in the method context menu (to open the menu select a method and push the right mouse button, then select *KeY*).

Here, KeY distincts between *Total Correctness*, *Partial Correctness* and *Throughout Correctness* of a method. Under the first two menu items you will find the same options, namely *PreservesInvariant*, *EnsuresPostCondition* and *Correctness* (which will be explained in a moment). The difference is, that if you want to prove one of these properties with respect to total correctness, the prover ensures additionally the termination of the method. Whereas the termination won't be considered if you choose this property with respect to partial correctness. That means that in the first case it will be guaranteed that the method always meets this property, and in the second that the property will be satisfied in case that the method terminates. The menu item *Throughout Correctness* refers to some throughout property of the method.

In the following, the properties of a method that can be checked by the KeY-Prover will be explained.

Total/Partial Correctness: The following three options are offered.

PreservesInvariant (DL): Correctness of an implementation of a method means that the implementation obeys the invariant of the class and ensures the postcondition of the method. Here it is checked that the execution of a method obeys the invariant under the assumption that the invariant and a possibly existing precondition hold.

Note that an invariant can be violated even if the obligation generated here is verified for every method (and constructor) of its class C_m, because methods of other classes might manipulate the state of objects of C_m. This phenomenon is sometimes called *Indirect Invariant Effect* [5, p.405] or *Representation Exposure*.

EnsuresPostCondition (DL): A proof obligation is computed ensuring that the postcondition of the current method holds after being executed under the assumption that possibly existing precondition and invariant hold.

Correctness (DL): Identical to the combination of *PreservesInvariant* plus *EnsuresPostCondition*.

Throughout Correctness → PreservesThroughout (DL): It is checked that the throughout property of the current method is satisfied. That means that the method obeys the strong invariant during the execution of the method, therefore after each execution of a statement.

5.2 Application in the Tutorial Example

Now we apply the described options in 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 1.3), especially that the current strategy is *Simple JavaCardDL* 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.

5.2.1 Options Offered in the Class Menu

StructuralSubtyping: Invoked from the context-sensitive menu of class `PayCardJunior`, this option starts the KeY-Prover with the proof obligation[10]

```
⇒
((self.balance ≥ 0 &
self.balance < PayCardJunior.juniorLimit) &
PayCardJunior.juniorLimit < self.limit)
-> (self.balance < self.limit & self.balance ≥ 0).
```
[11]

This proof obligation is a pure first-order predicate logic formula. The premiss of the implication is the translated invariant of the subclass `PayCardJunior` and the conclusion is the translated invariant of the superclass `PayCard`.[12] There are two ways to prove this with the KeY-Prover:

Automatic: Just push the *run strategy*-button (which is the button with the $\triangleright$ - symbol) to start the proof.

[10]please ensure that you have generated the invariant of PayCard as described in section 3.2

[11]If the formula displayed in the KeY-Prover looks different enable the checkbox *Pretty&Untrue* from the menu group *View*.

[12]The translation is necessary because the syntax of the OCL and predicate logic differs.

The KeY-Prover simplifies the open goal automatically and informs you that the goal could be proven. You can quit the KeY-Prover with menu item *File* → *Exit*.

Interactive: To prove the proof obligation in the example interactively, perform the following proof steps:

1. Apply rule *imp_right* to the whole formula in the succedent in order to remove the implication. As a result, the premiss of the implication becomes the antecedent of the sequent and the conclusion becomes the succedent.
2. Apply rule *and_left* to the whole formula in the antecedent. This rule removes the conjunction operator.
3. Apply rule *and_left* to formula
`self.balance` $\geq$ `0 &`
`self.balance` $<$ `PayCardJunior.juniorLimit`
to remove the conjunction.
4. Apply rule *and_right* to the whole formula in the succedent. Formulas in the succedent of a sequent are implicitly disjuntively connected. Thus, the conjunction operator cannot just be replaced by a comma as it is the case in the antecedent, where the formulas are implicitly conjunctively connected. Rather, applying rule *and_right* results in two sequents, each of them containing one of the conjunctives in the succedent. After the rule application, the proof tree (which is shown in the tab *Proof* of the lower left pane) will therefore contain a branch.
5. Now consider *Case 1* in the proof tree at first. The formula
`self.balance` $\geq$ `0` is contained as well in the antecedent as in the succedent. This goal can be closed by applying rule *close_goal* to `self.balance` $\geq$ `0` in the succedent because a sequent of the form $\phi, \Gamma \vdash \phi, \Delta$ is an axiom.
6. There is still an open goal in *Case 2*. In the succedent of this sequent we have the formula `self.balance < self.limit` and in the antecedent we have
`self.balance < PayCardJunior.juniorLimit`
and `PayCardJunior.juniorLimit < self.limit`. To close this goal we have to make use of the transitivity of the relation "`<`". Therefore we apply rule *less_trans* to formula `(PayCardJunior.juniorLimit < self.limit)`.
After doing that you can click on the node preceding the current open goal to see the explanation of this rule. It has the following effect: The formula in the antecedent it was applied to (`i0 < i1`) is highlighted in dark green. If a formula of the form `i < i0` exists in the antecedent of the sequent (highlighted in light-green), then the new formula `i < i1` will be added. Thus, the formula `self.balance < self.limit` is added in the antecedent of our example.
7. Now, on both sides of the sequent we have the formula `self.balance < self.limit` and, as above, we can close this goal by applying rule *close_goal* on `self.balance < self.limit` in the succedent.

5.2.2 Options Offered in the Method Menu

We only consider the options *PreservesInvariant*, *EnsuresPostCondition* and *Correctness* with respect to *Total Correctness*, because we want the prover to ensure

that the corresponding method terminates. A non-terminating method would not be as requested in this example.

PreservesInvariant: Try to apply this to method `charge` in class `PayCard`. Proving this property requires to verify the actual implementation of `charge` against the invariant. Therefore, the generated proof obligation contains JAVA code in the generated Dynamic Logic formula.

First, select checkbox *Autoresume strategy* and then start the proof by pushing the *run strategy*-button. When the strategy stops, one goal is still open. This goal cannot be proven automatically by applying the strategy, thus we have to continue either interactively or by running the decision procedure Simplify. The latter is done by pushing the button *Run SIMPLIFY*, which will succeed right away.

If you want to prove the goal interactively apply rule *add_less* to the formula `(self.balance) < 0` in the succedent of the sequent and enter `amount` in the input slot *Instantiation* for variable `i1` of the rule instantiation dialog. The formula will be replaced by `(amount + self.balance) < (amount + 0)`. After that, the prover resumes automatically by applying rule *add_zero_right* (which results in deleting the redundant part `+ 0` on the right side of the new formula) and then immediately stops again. Now, apply rule *switch_params* interactively to the first part of the new created formula. Then the prover once again resumes automatically and the goal can be proven by running the strategy.

EnsuresPostCondition: We demonstrate this options by means of method `checkSum` in class `PayCardJunior`. The postcondition of this method states that the return value is 1 if the parameter `sum` is less than `juniorLimit` and 0 if `sum` is greater or equal `juniorLimit`.

Proving that the implementation of method `checkSum` ensures the postcondition can be done automatically applying the current strategy *Simple JavaCardDL*.

Correctness: Again we use method `checkSum` in class `PayCardJunior` to demonstrate this option. Once again the goal can be proven automatically applying the strategy.

6 Current Limitations and Restrictions

The current version of the KeY-Tool is far from being a polished and universally applicable tool. Here is a list of open issues we are now working on and intend to resolve in near future:

1. Supported platforms:
   - Linux is tested, Solaris and MacOS X should work as well
   - Windows NT, 2000 and XP should work when using the KeY byte code version.
2. Restrictions on UML models:
   - when invoking an analysis/verification option all involved classes (usually the current class and the parent class) must be members of the current diagram

3. Restrictions of the KeY-Prover:

    - manual not yet available (will be available soon!)
    - powerful automated deduction system not integrated

4. Restrictions on OCL translation:

    - problem: unsupported at the moment are: `allInstances, Set{1..n}` (and similar)
      workaround: express it as a dynamic logic formula in a `.key`-file.

5. Restrictions on JDK:

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

A Formal Description of Generated Proof Obligations

In general, proof obligations are based on assertions (invariants, preconditions, postconditions) attached to a current element (class or method) or its (direct) parent classes[13].

To facilitate the description of the proof obligations we take advantage of the following abbreviations, where we assume to have a total order on parent classes with index set $P = \{1, \ldots, n\}$.

Class	INV	invariant of the current class
	INV^{P_i}	invariant of the i-th parent class
Method	$m.PRE$	precondition of method m in current class
	$m.POST$	postcondition of method m in current class
	$m.PRE^{P_i}$	precondition of method m in i-th parent class
	$m.POST^{P_i}$	postcondition of method m in i-th parent class

With the exception of postconditions ($m.POST, m.POST^{P_i}$) the abbreviations stand for pure predicate logic (PL) formulas (at this stage we assume that the OCL expressions from the UML model were translated already into PL formulas).

Postconditions are PL formulas up to @*pre*-expressions and *result*-variables that need special attention when translating OCL into PL. It is assumed that this transformation has been done.

All PL formulas contain the variable $self$ referring to the current object. In some cases the occurrence of $self$ is important and needs to be emphasised. For this we write $INV(self)$, $m.PRE(self)$, etc., instead of INV, $m.PRE$, etc.

The structure of the rest of this section parallels that of Section 5.1.

A.1 Options Offered in the Class Menu

StructuralSubtyping (PL) The invariant of the current class CC is stronger than the ones of the parent classes:

(PO) $\bigwedge_{i \in P} (\forall\ self : CC\ INV(self) \rightarrow INV^{P_i}(self))$

A.2 Options Offered in the Method Menu

PreservesInvariant(DL) The implementation of the current method obeys (preserves) the invariant of the current class.

(PO) $\forall self : CC\, m.PRE(self) \wedge INV(self) \rightarrow \langle m \rangle INV(self)$

If method m is a constructor, then the subformula $m.PRE(self) \wedge INV(self)$ can be assumed to be equivalent to *true*. Therefore, in this case the proof obligation is simplified to:

(POC) $\forall self : CC\, \langle m \rangle INV(self)$

EnsuresPostCondition(DL) The implementation of the current method satisfies its postcondition.

(PO) $\forall self : CC\, m.PRE(self) \wedge INV(self) \rightarrow \langle m \rangle m.POST(self)$

As above, if m is a constructor, then $m.PRE(self) \wedge INV(self)$ can be assumed to be equivalent to *true*. The proof obligation is simplified to:

[13] We allow a class to have more than one parent class here. However, since interfaces are currently not supported, due to the JAVA class hierarchy restrictions, the actual proof obligations involve only one parent class.

(POC) $\forall self : CC\, \langle m \rangle m.POST(self)$

Correctness(DL) *PreservesInvariant* plus *EnsuresPostCondition*:

(PO) $\forall self : CC\, m.PRE(self) \wedge INV(self) \rightarrow$
$\langle m \rangle INV(self) \wedge m.POST(self)$

Again, if m is a constructor, then $m.PRE(self) \wedge INV(self)$ can be assumed to be equivalent to *true*. The proof obligation is simplified to:

(POC) $\forall self : CC\, \langle m \rangle INV(self) \wedge m.POST(self)$

References

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

[2] `www.icansolve.com`.

[3] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The Key Tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, Feb. 2003. Available at: `http://i12www.ira.uka.de/~beckert/pub/key03.pdf`.

[4] B. 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.

[5] B. Meyer. *Object-Oriented Software Construction (Second Edition)*. Prentice-Hall, 1997.

[6] S. 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`.