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

Contents

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 TogetherControlCenter[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 [1].

The most recent version of the KeY-Tool can be downloaded from `http://i12www.ira.uka.de/~key/download.htm`.

1.1 Version Information

This tutorial was tested for TogetherCC versions 6.0, and 6.0.1.

1.2 Logical Foundations

Deduction with the KeY-Prover is based on a sequent calculus for a Dynamic Logic for JavaCard (JavaDL) [2]. 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-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 two panes where the left pane is additionally tabbed. Each pane is described below.

Left pane, Proof Obligations: Some properties you may want to prove require to prove several proof obligations. E.g., following the principle *Design by Contract* you have to prove that the pre-condition pre_{sub} of a method m in a subclasses implies the pre-condition $pre_{super}$of $meth$ in its superclass and, vice versa, that the post-condition $post_{sub}$ of $meth$ in the subclass implies the post-condition $post_{super}$ of $meth$ in its superclass. Thus, the two proof obligations $pre_{super} \rightarrow pre_{sub}$ and $post_{sub} \rightarrow post_{super}$ are generated and displayed in tab *Proof Obligations*. Each proof obligation in this tab requires its own separate proof. You can switch between different proofs by selecting the according proof obligation.

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

Left pane, 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.

Left pane, Goals: In this pane the open goals of a certain proof (corresponding to one entry in tab *Proof Obligations*) 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 heuristics:

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 heuristics which can be seen as a collection of rules suited for a certain task. For example, the heuristic *simplify_boolean* contains rules to simplify boolean expressions. To determine which heuristics should be used select menu item *Options* → *Heuristics*. A dialog pops up where you can define the active heuristics (right pane) from a set of available heuristics (left pane). Furthermore, you can set the maximal number of automatic rule applications. To save your settings for further proofs select menu item *Options* → *Save Settings*.

To start (respectively continue) the proof push button *Apply Heuristics*. In the status line of the KeY prover a button appears to stop the heuristics. Furthermore, this it contains a progress bar showing the relation of already applied rules and the maximal number of automatic rule applications. If the checkbox *Autoresume heuristics* is selected, the prover automatically resumes applying heuristics after an interactive rule application.

There is one exception from this behaviour: If the heuristics *simplify_updates* is selected — which is recommended to always be the case — this heuristics is always applied after each interactive proof step, even if *Autoresume heuristics* is not selected. This comes from the particular importance of simplifying updates to the KeY JavaDL-calculus.

In the following we describe the menu items available in the main menu of the KeY-Prover.

***File* → *Load*:** Loads a previously saved proof.

Attention: loading and saving of proofs is currently (Vers. 0.9) not functional if so called "implicit" attributes are part of the proof. "Implicit" attributes are part of the object initialisation scheme of the JavaDL-calculus rules.

***File* → *Save*:** Saves current proof. Note, that if there are several proof obligations (see tab *Proof Obligations* in the 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 views. If unselected, terms and formulas are displayed in their internal representation which is often very hard to read. For example, the formula $5 < 6$ would be displayed as $lt(5, 6)$. For a user-friendly representation of terms and formulas select *Pretty&Untrue*. Be warned: Some rule applications require the user to provide a term or formula. However, using the user-frienly syntax to enter terms or formulas is currently not possible (therefore "untrue") and the syntax of the internal representation has to be used. Thus, if you want to enter the formula $5 < 6$ you have to type $lt(5, 6)$.

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

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

***Options* → *Heuristics*:** Allows to define the set of activated heuristics.

***Options* → *LDT Models*:** Here you can choose between different semantics for Java integer arithmetic. 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.

What is behind all this goes beyond the scope of this quicktour. If you want to know more, please refer to [5]. Otherwise, just ignore this menu item and use the default settings.

***Options* → *Update Simplifier*:** Here you can define policies how updates should be simplified. As the description of LDT Models above, this goes beyond the scope of this quicktour. Please use the default settings unless you know what you are doing.

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

2 Tutorial Example

In this tutorial we use a simple `paycard` application to illustrate most of the 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).

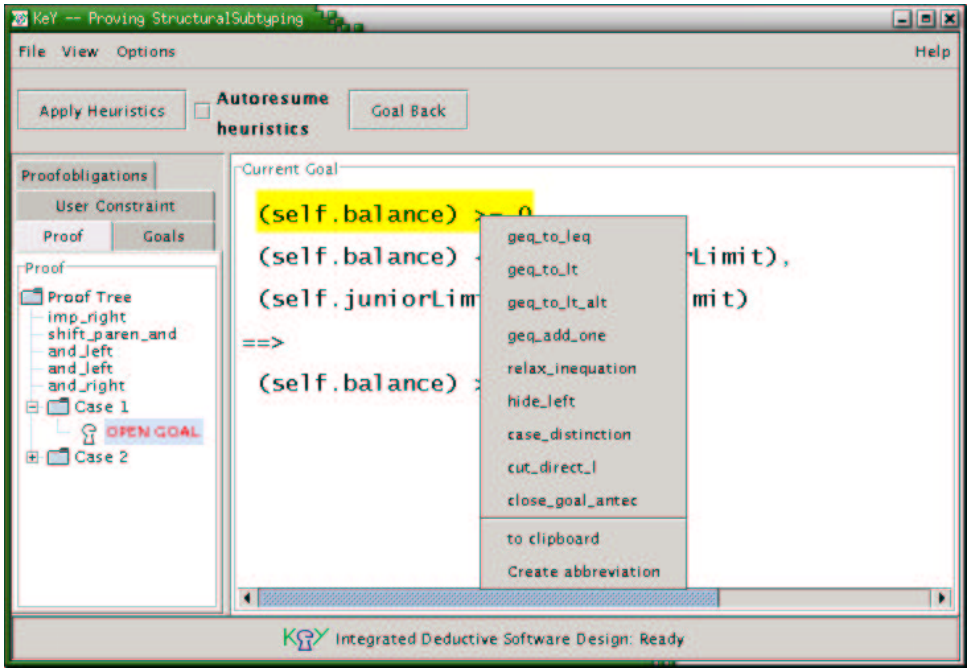

Figure 1: The KeY-Prover.

The class diagram shown in Figure 2 consists of the six classes `PayCard`, `PayCardJunior`, `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.

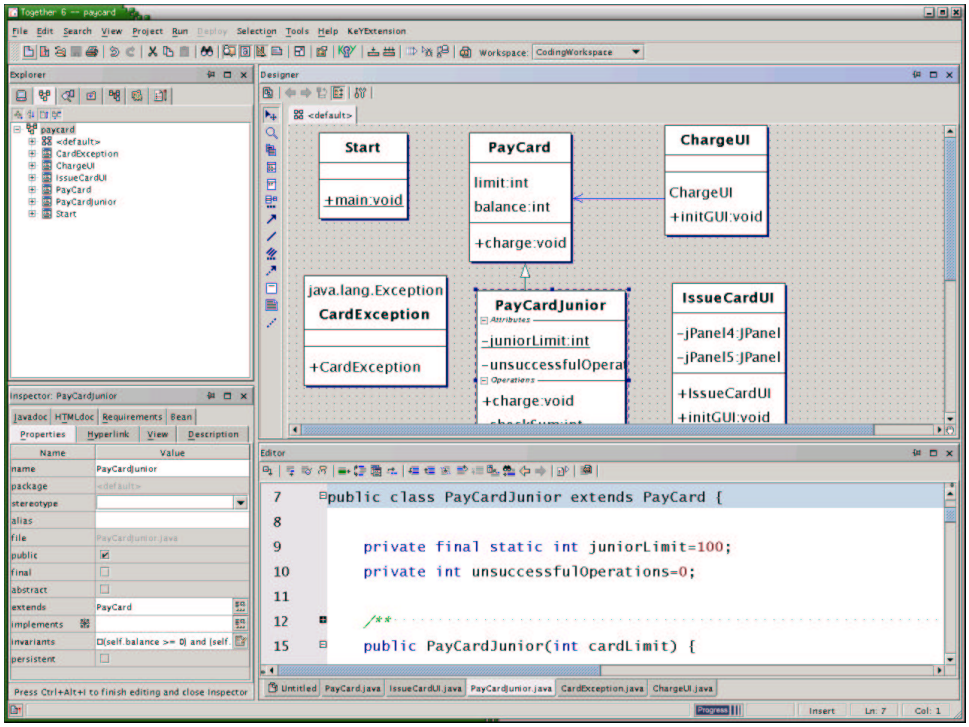

Figure 2: Class Structure of Tutorial Example.

3 Creating a Formal Specification in the OCL

Rigorous specification is a necessary prerequisite to discuss the "correctness" of an 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, `ProduceForAssociationSet`, `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.

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, use the mouse to select class `PayCard` 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`", "`Attribute with lower bound`" should be "`balance`", "`lowerOperator`" should be "≥", "`Upper bound for attribute`" should be "`limit`", and "`upperOperator`" should be "<" (see Figure 3). After pushing the *Finish* button, the pattern dialog disappears and the intended specification is generated and added to class `PayCard`. As this example shows, it is possible to select several blueprints simultaniously. 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

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

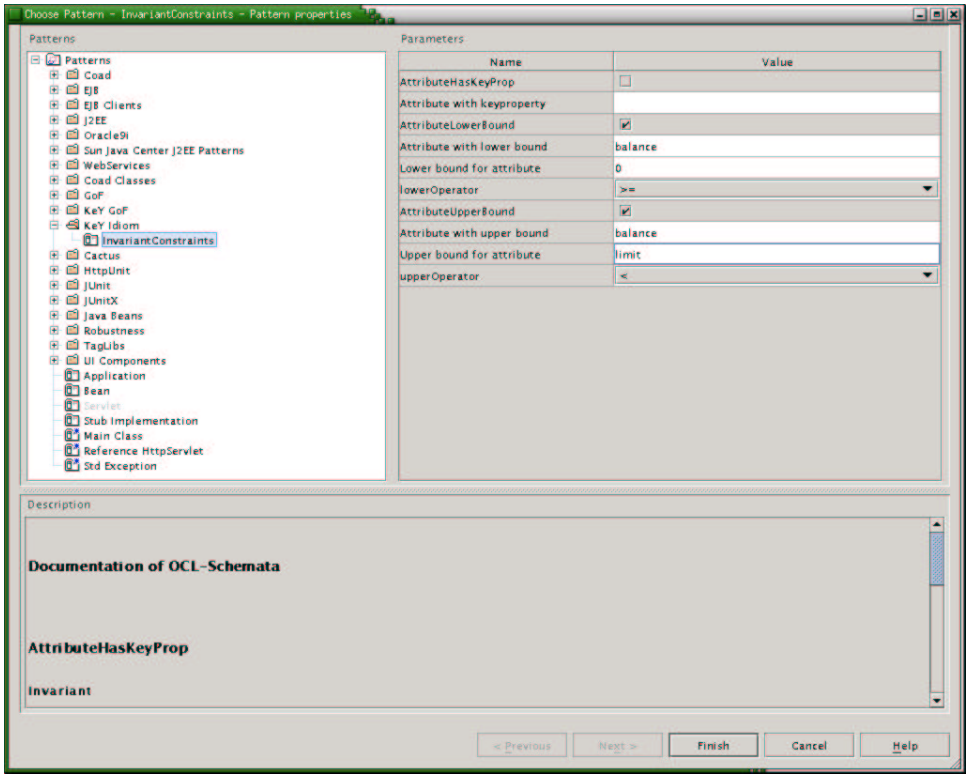

Figure 3: Generation of OCL Expressions.

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`. Then, just right-click on the class in the "Designer" pane of TogetherCC, and select *Edit Invariant [GF]* from the *KeY Extension* 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 questions 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 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.

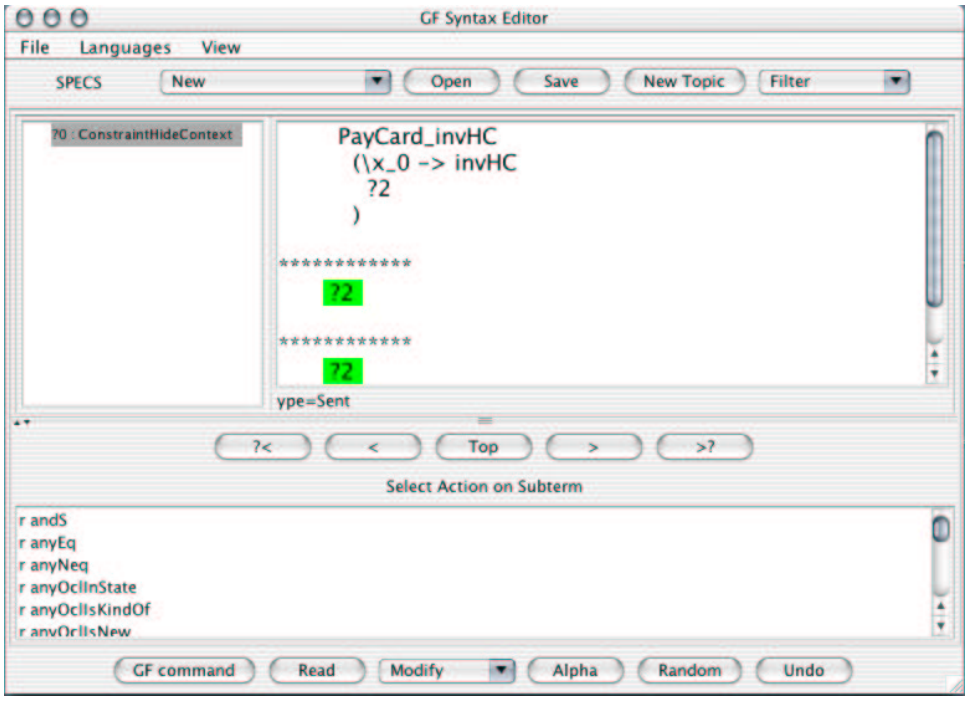

Figure 4: Editing a new invariant in the syntax editor

Editing proceeds by filling in questions marks 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.

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[2], and we can choose to show type information or not.[3] 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).

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

[3]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.

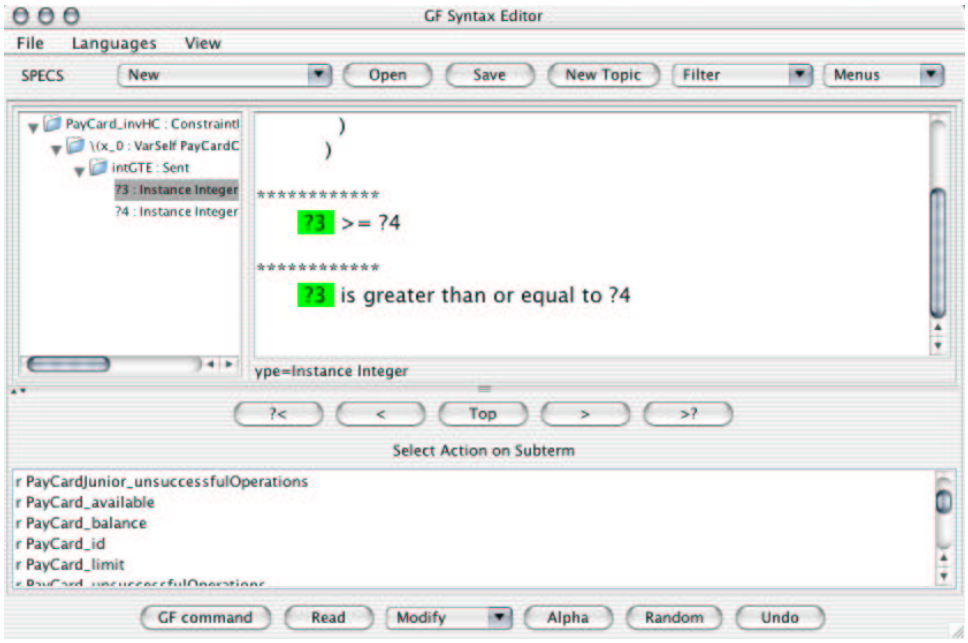

Figure 5: The first refinement step

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

The left metavariable can in the example be filled in by choosing the refinement `PayCard_balance` 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 "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 an 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.

As an example let us invoke *ParseInvariant* in class `PayCard` and the parser will tell you that the invariant `(balance` $\geq$ `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* iff it actually implies properties expressed in its specification. The KeY-Tool includes functionality to *verify*[4] 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 is 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[5] 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 heuristic information. In the near future a state-of-the-art automatic proof engine based on saturated tableaux will be integrated. This will increase deductive power of the KeY-Prover dramatically, at least for predicate logic problems.

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.

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

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

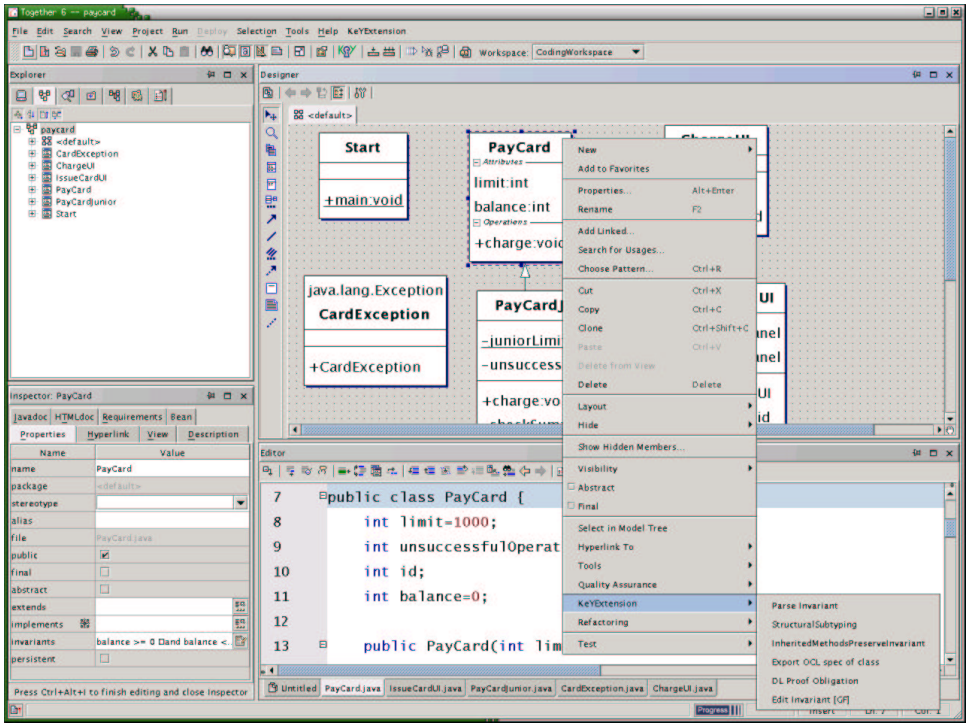

Figure 6: Options offered by the Class Menu.

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 [4]).

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).

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

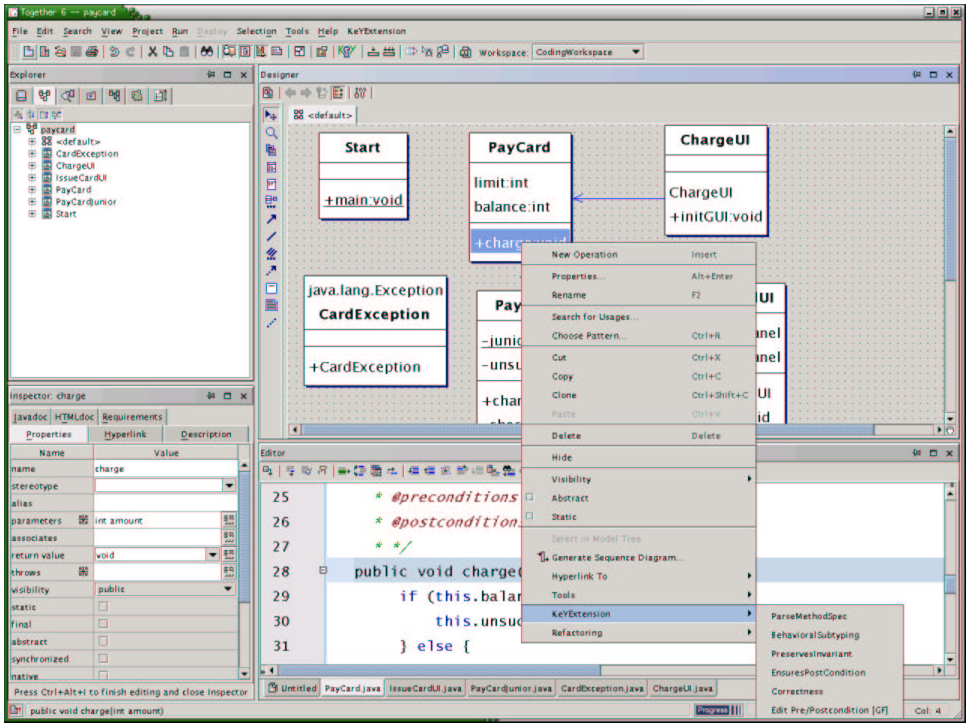

Figure 7: Options offered by the Method Menu.

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.

InheritedMethodsPreserveInvariant (PL) Suppose you are writing code for a new class CC which inherits from a library class PC. The implementation of PC is not available or it could be changed without notice. The only reliable information about PC are its interface and the specification of PC, that is, the invariants, pre- and postconditions of its methods.

Assume $m(\ldots)$ is a method implemented in class PC, but not reimplemented in subclass CC. What can we say about m and the invariant of CC? The generated proof obligation ensures that the invariant of CC can be inferred provided that the implementation of m meets its specification (invariant and pre-/postconditions) in PC.

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).

BehaviouralSubtyping (PL) Behavioural subtyping is another consequence of Liskov's substitution principle. If we consider a method specification as a contract between the supplier (the class that implements the methods) and the client (the class that calls the method) we conclude that for contracts to be satisfied, the following must hold:

1. A method in a subclass of the supplier class must be applicable at least in all those situations, where the specification of the method in the supplier

class promises it to be applicable.

2. Application of a method in a subclass of the supplier class results in a state that has all the properties promised by the specification of the method in the supplier class.

In other words: preconditions of any method must become logically weaker and postconditions must become logically stronger in subclasses.

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* [4, 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*.

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. 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
$\Rightarrow$`((self.balance` $\geq$ `0) & (self.balance` $<$ `self.juniorLimit)) &`
`(self.juniorLimit` $<$ `self.limit) ->`
`(self.balance` $\geq$ `0) & (self.balance` $<$ `self.limit).`[6]

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`.[7] There are two ways to prove this with the KeY-Prover:

Automatic: Focus on the window titled *PO for StructuralSubtyping* and via menu item *Options* $\rightarrow$ *Heuristics...* select heuristics *simplify* and *simplify_int* and set *Maximum heuristics applications* to 20. Then, push button *Apply Heuristics* to start the proof.

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* $\rightarrow$ *Exit*.

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

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

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 `((0 < self.balance) | self.balance=0) & (self.balance < self.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.
5. Now the formula `(self.balance ≥ 0)` is contained as well in the antecendent as in the succedent. This goal can be closed by applying rule *close_goal* to `(self.balance ≥ 0)` in the succedent because a sequent of the form $\phi, \Gamma \vdash \phi, \Delta$ is an axiom.
6. In the succedent we have the formula `(self.balance < self.limit)` and in the antecedent we have `(self.balance < self.juniorLimit)` and `(self.juniorLimit < self.limit)`. To close this goal we have to make use of the transitivity of the relation "`<`". This is done by applying rule *less_trans* to formula `(self.juniorLimit < self.limit)`. A dialog pops up where you can enter the formula the transitivity should be applied on. In this case the only possible formula `(self.balance < self.juniorLimit)` is already suggested by the KeY-Prover. Thus, just push button *Apply*.
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

BehaviouralSubtyping Select method `charge` of class `PayCardJunior` in the class diagram and invoke *Behavioural Subtyping* from the context menu. You will obtain two proof obligations (displayed in tab *Proof obligation* in the left pane of the KeY-Prover). The first proof obligation `⇒(amount>0)` → `(amount>0)` is trivial and proven automatically if heuristic *simplify* is selected.

To prove the second proof obligation select all heuristics and set the number of automatic proof steps to 100. When the heuristics stop three goals are left and you have two possibilities to continue:

1. **Interactive proof steps + heuristics.**
   Select checkbox *Autoresume heuristics* in the main window of the prover.
   To close the first goal apply rule *insert_eq* to the formula `(self.balance) < PayCardJunior::balance(self)` in the antecedent of the sequent. This will replace `self.balance` with `PayCardJunior::balance(self)+amount`.[8]

[8] This proof step is sound because of the equation `self.balance = PayCardJunior::balance(self)+amount` in the antecedent.

Then apply rule *add_less_back* to `PayCardJunior::balance(self)+amount < CardJunior::balance(self)` in order to subtract `PayCardJunior::balance(self)` on both sides of the inequation. Now the goal will be closed automatically by applying heuristics.

To close the remaining two goals apply rule *insert_eq*[9] to the the left-hand side of the inequation `(self.balance) < PayCardJunior::balance(self)`.

2. **Automatic proof using a decision procedure.**

   We have integrated a so-called *decision procedure* for integer arithmetics in the KeY-Prover. This decision procedure which is called *Simplify*[10] can decide whether a formula in a certain fragment of integer arithmetic (Presburger arithmetic) is valid. You can invoke Simplify by highlighting the whole sequent and clicking the left mouse button. Then there is a menu entry *Decision Procedure "Simplify"*. You can close all remaining goals by running Simplify.

   Instead of doing this for all goals, you can just use the button *Run SIMPLIFY* in the main toolbar above the sequent window to apply the decision procedure to *all* open goals.

   Note: You do not have to check whether the formulas in the sequent are really Presburger arithmetic or not. You can always try to run Simplify.[11] If the validity of the formula cannot be established by Simplify you get an according message from the KeY-Prover.

PreservesInvariant Try to apply this to method `charge` in class `PayCardJunior`. 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 all available heuristics and set the maximal number of automatic proof steps to 1000. Then select checkbox *Autoresume heuristics* and start the proof by pushing the button *Apply Heuristics*. When the heuristics stop one goal is still open. This goal cannot be proven automatically by applying heuristics, thus we have to continue either interactively or by running the decision procedure Simplify.

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 of the rule instantiation dialog. After that apply *switch_params* to the first part of the new created formula. Then the goal can be proven by running the heuristics.

EnsuresPostCondition We demonstrate this options by means of method `checkSum` in class `JuniorPayCard`. 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`.

To proof that the implementation of method `checkSum` ensures the postcondition select all heuristics and set the maximal number of heuristic proof steps to 1000. The goal can be proven automatically applying heuristics.

[9] One of the two goals contains two inequations with the same left-hand side `self.balance` in the antecedent and, thus, two rules *insert_eq* are offered. In this case choose the second one. We are currently working on improving the naming of the rules.

[10] Simplify is part of ESC/Java [3] developed at Compaq.

[11] If the sequent containes formulas that are not in the fragment of Presburger arithmetic, these formulas are left out. This corresponds to the sound proof step *weakening*.

Correctness Again we use method `checkSum` in class `JuniorPayCard` to demonstrate this option. Select all heuristics and set the maximal number of heuristic proof steps to 250. The goal can be proven automatically applying heuristics.

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 should work as well
   - Windows NT, 2000 and XP should work when using the KeY byte code version.
2. Restrictions on UML models:
   - implementation classes must not define packages
   - 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 available yet (will be available soon!)
   - powerful automated deduction system only partly integrated

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[12].

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).

[12]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.

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

InheritedMethodsPreserveInvariant(PL) In principle, we have to prove that the inherited implementation of method m preserves the invariant (where we can make use of the precondition):

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

If we assume that the implementation of m meets its specification as given in parent class P_i we know in addition:

(Assumption) $\forall pself : P_i\ m.PRE^{P_i}(pself) \wedge INV^{P_i}(pself) \rightarrow \langle m \rangle m.POST^{P_i}(pself) \wedge INV^{P_i}(pself)$

As all constraints that are valid for instances $pself$ of P_i should also be valid for the instance $self$ of the current class CC, we have:

(Lemma) $\forall self : CC\ m.PRE^{P_i}(self) \wedge INV^{P_i}(self) \rightarrow \langle m \rangle m.POST^{P_i}(self) \wedge INV^{P_i}(self)$

Now, we apply a valid proof rule

$$\frac{A^l \rightarrow \langle m \rangle B^l \qquad A^g \rightarrow A^l \qquad A^g \wedge rename(B^l) \rightarrow rename(B^g)}{A^g \rightarrow \langle m \rangle B^g}$$

saying that allows to prove a formula of the form $A^g \rightarrow \langle m \rangle B^g$ with the help of a lemma of form $A^l \rightarrow \langle m \rangle B^l$ and (1) $A^g \rightarrow A^l$ and (2) $A^g \wedge rename(B^l) \rightarrow rename(B^g)$, where $rename(X)$ renames all OCL properties that can be affected by executing m (in terms of Dynamic Logic: all non-rigid symbols) in X and resolves @*pre*-constructs.

Let $MinP = \{i \in P \mid m \text{ is implemented in } P_i\}$. We can obtain as (1) and (2) in the context of (Goal) and (Lemma):

(PO1) $\bigwedge_{i \in MinP}\ \forall self : CC\ m.PRE(self) \wedge INV(self) \rightarrow m.PRE^{P_i}(self) \wedge INV^{P_i}(self)$

(PO2) $\bigwedge_{i \in MinP}\ \forall self : CC\ m.PRE(self) \wedge INV(self) \wedge rename(m.POST^{P_i}(self) \wedge INV^{P_i}(self)) \rightarrow rename(INV(self))$

The first proof obligation can be simplified using $m.PRE = m.PRE^{P_i}$:

(PO1′) $\bigwedge_{i \in MinP}\ \forall self : CC\ m.PRE(self) \wedge INV(self) \rightarrow INV^{P_i}(self)$

A.2 Options Offered in the Method Menu

BehaviourSubtyping(PL) The precondition of the current method is weaker than the precondition of the parent method. The postcondition of the current method is stronger than the postcondition of the parent method.

Let $MinP$ be defined as above.

(PO1) $\bigwedge_{i \in MinP} \forall self : CC\, m.PRE^{P_i}(self) \rightarrow m.PRE(self)$

(PO2) $\bigwedge_{i \in MinP} \forall self : CC\, m.PRE^{P_i}(self) \wedge m.PRE(self) \rightarrow rename(m.POST(self) \rightarrow m.POST^{P_i}(self))$

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:

(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] 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`.

[2] 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.

[3] ESC/Java (Extended Static Checking for Java). `http://research.compaq.com/SRC/esc/`.

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

[5] 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`.