Appendix for FASE System Description

The KeY System: Integrating Object-Oriented Design and Formal Methods – Appendix –[*]

Wolfgang Ahrendt[2], Thomas Baar[1], Bernhard Beckert[1], Martin Giese[1], Reiner Hähnle[2], Wolfram Menzel[1], Wojciech Mostowski[2], and Peter H. Schmitt[1]

[1] Universität Karlsruhe
Inst. f. Logik, Komplexität und Dedukt.-Syst.
D-76128 Karlsruhe, Germany
[2] Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden

1 Introduction

This document is a short description of the contents of a demo of the KeY system.

1.1 Prerequisites

The KeY system currently runs on PCs under Linux.[1] To install the system, you need the following items:

- Together Control Center (TogetherCC), version 5.01 or 5.5. An evaluation version of TogetherCC can be obtained free of charge from: `http://www.togethersoft.com/`
- A Java runtime environment, version 1.3 or higher.
- Perl.
- The KeY system itself, which can be downloaded from: `http://i12www.ira.uka.de/ key/download.htm`
  The tutorial example described in this document can be downloaded from the same page.

[*] The KeY project is supported by the Deutsche Forschungsgemeinschaft (grant no. Ha 2617/2-1).

[1] We are planning to support Windows soon.

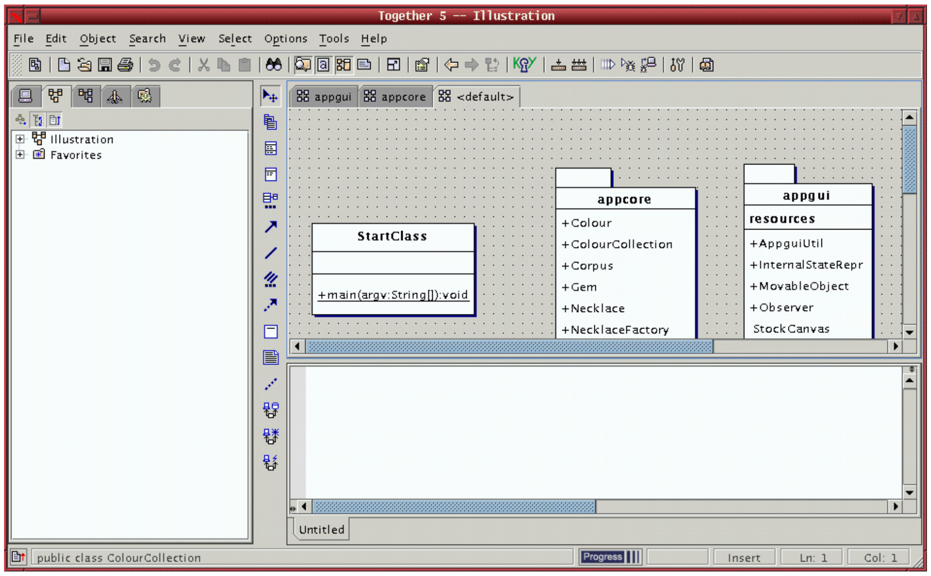


Fig. 1. Package Structure of Tutorial Example

1.2 Tutorial Example

In this tutorial we use an example application to illustrate most of the capabilities offered by the KeY system.

The tutorial example is a Java application organized in two packages, see Fig 1.

– `appgui` — the implementation of the GUI of the application
– `appcore` — the application itself

In addition, there is a class `StartClass` which provides the `main` method to start the application.

The `appcore` package realizes a simulation of a simple Order/Manufacture/ Deliver scenario. The GUI built on top of that in `appgui` offers an order dialog and makes it possible to observe the internal state of the application's most interesting object, the manufacturing centre including its stock of supplies.

The items to be ordered, manufactured, and delivered are necklaces that are decorated with gems. The customer (user of the application) fills in an order form for each necklace with the number of gems of each available colour. One can also specify that a necklace must be one-coloured. Then the order is sent to the manufacturing centre, which in turn sends a request to the associated warehouse to make sure that sufficient resources (the corpus of the necklace and enough gems of the desired colour) are in stock. Finally, the manufacturing centre assembles the necklace and checks, whether the new necklace meets all

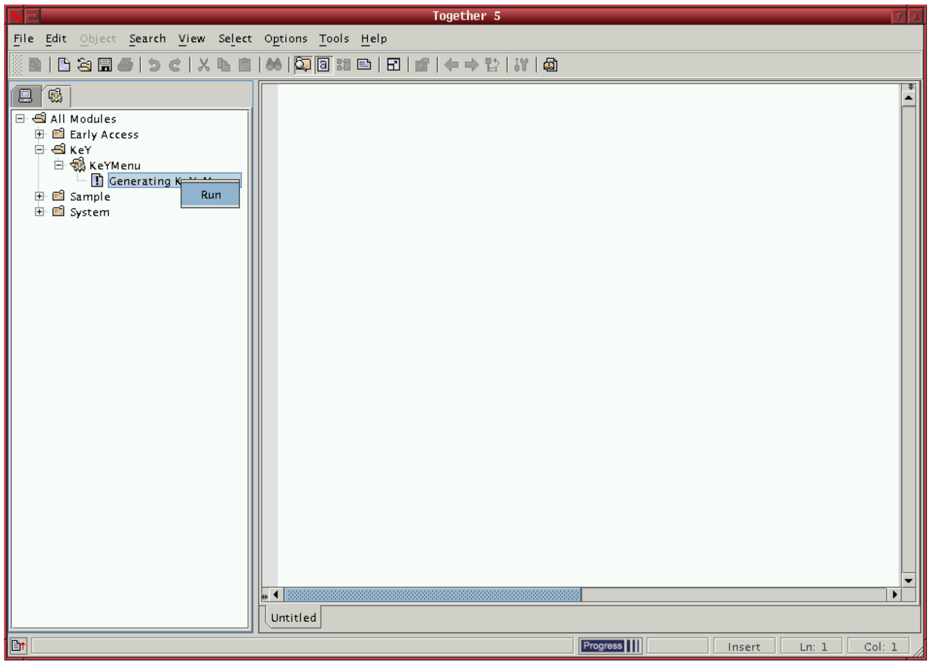


Fig. 2. Activation of KeY Extensions

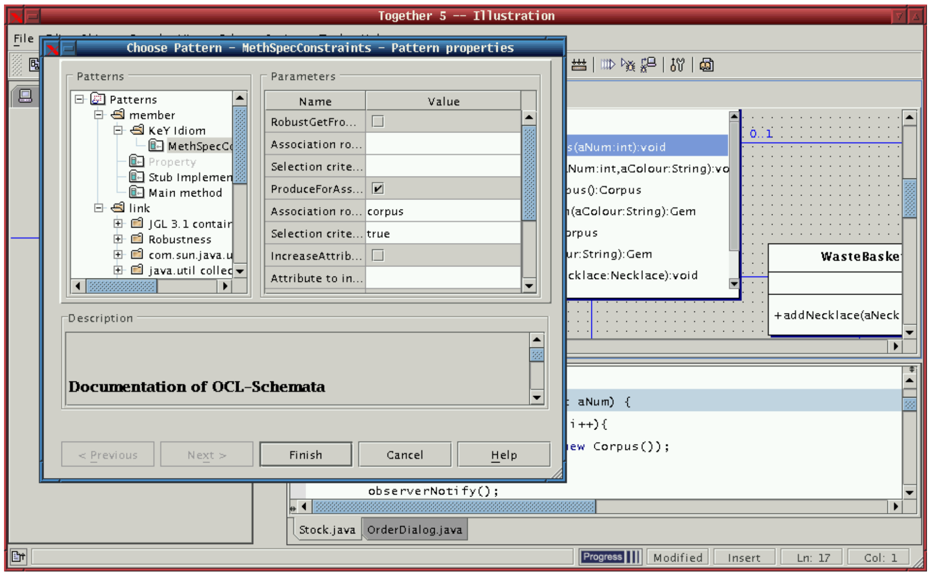
requirements stated in the order. If this is successful, the necklace is ready for delivery. Otherwise, the necklace is thrown into a wastebasket. The latter can happen, because the order may contain contradictory information, such as a one-coloured necklace with red gems and blue gems. This possibility can be seen as a bug of the current implementation. The KeY system aims to *formally prove* the *absence* of such bugs. For example, one could formally verify an invariant stating that the wastebasket is always empty.

The static structure of the example application is modelled in the class diagram `appcore`. 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 OCL.

The demo outlined in this document shows how to create and modify the OCL specifications, how to parse them, how to generate proof obligations and how to prove these proof obligations.

2 Creating a Formal Specification in OCL

The first step is to activate the KeY extensions to TogetherCC. This is done by a simple mouse click, as seen in Fig. 2.



Name	Value
RobustGetFro...	☐
Association ro...	
Selection crite...	
ProduceForAss...	✓
Association ro...	corpus
Selection crite...	true
IncreaseAttrib...	☐
Attribute to in...	

Documentation of OCL-Schemata

```
s(aNum:int):void
Num:int,aColour:String):vo
pus():Corpus
(aColour:String):Gem
rpus
ur:String):Gem
cklace:Necklace):void
```

```
 aNum) {
i++){
ew Corpus());
    observerNotify();
```

Fig. 3. Generation of OCL Expressions

The KeY system supports authoring of OCL constraints that express typical requirements. The technology behind this is a template-like and easy-to-understand mechanism. Consider, for example, the behavioural specification of typical *production-and-store* methods that create a certain amount of new objects and store them in an associated collection of objects. The specification of such methods has the following form in general:

```
production-and-store-method(tint i):
  pre:  i >= 0
  post: associatedObjects->size = associatedObjects@pre->size+i
```

There is a plethora of similar constraints needed in related situations (e.g., `ProduceForAssociationSet`, `GetFromAssociationSet`, `IncreaseAttribute`, and, as an example for an invariant constraint, `AttributeHasKeyProp`). The KeY system 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, 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 customization dialog, see Fig. 3. As 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 Parsing a Specification

When a number of OCL constraints has been added to the model, the KeY system can parse them to check that they are syntactically correct and well-typed. The parser currently integrated in the KeY system was developed at Dresden University of Technology.[2]

As OCL expressions only makes sense in the context of a UML diagram, the parser needs to consult an XMI file containing model information of the current UML model. At the current stage, the user must ensure manually that this XMI file was generated and is up-to-date. There is a menu item to do this.

The user can now chose to parse a class invariant or the pre- and postconditions of a method: The KeY system extends the TogetherCC context menus for classes and methods (see Fig. 4) by a number of additional items. Selecting *Parse Invariant* passes the invariant of a class to the OCL parser and reports any syntax or typing errors found.

4 Generation of Proof Obligations

When some classes or methods have been annotated with OCL constraints that are accepted by the parser, the KeY system can generate proof obligations expressing statements about the specification (*analysis*) or about the relationship between the specification and the implementation (*verification*).

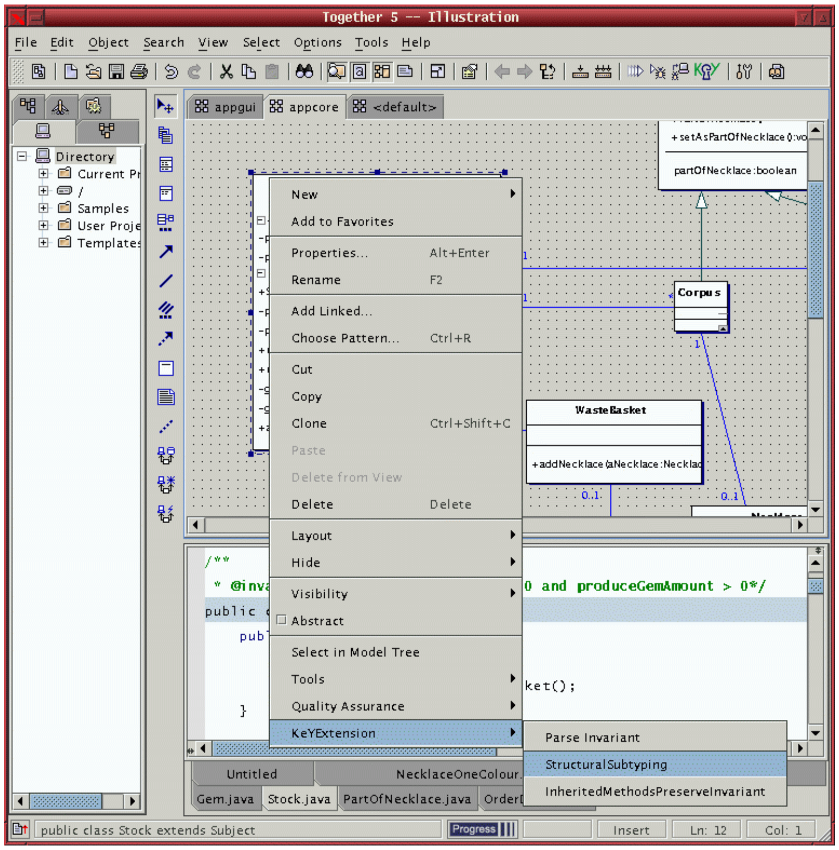
For instance one might want to show that the postcondition of some method implies the postcondition of that method specified in some superclass. Maybe one might also be able to show that the pre- and postconditions of a method are strong enough to ensure that a method preserves the class invariant if only it satisfies its pre- and postcondition. These are analysis statements. A typical verification statement would be that the implementation of a method ensures that the postcondition holds after execution.

As Fig. 4, the generation of such proof obligations is also done using the context menus of methods and classes. The OCL constraints are parsed again, and translated into a dynamic logic for Java(JavaDL). The resulting formulae are then suitably combined to provide the proof obligations.

5 Using the Prover

The generated proof obligations are passed to the KeY prover, a specialized deduction system for JavaDL. The prover features a graphical user interface

[2] See `http://dresden-ocl.sourceforge.net/index.html` for details.



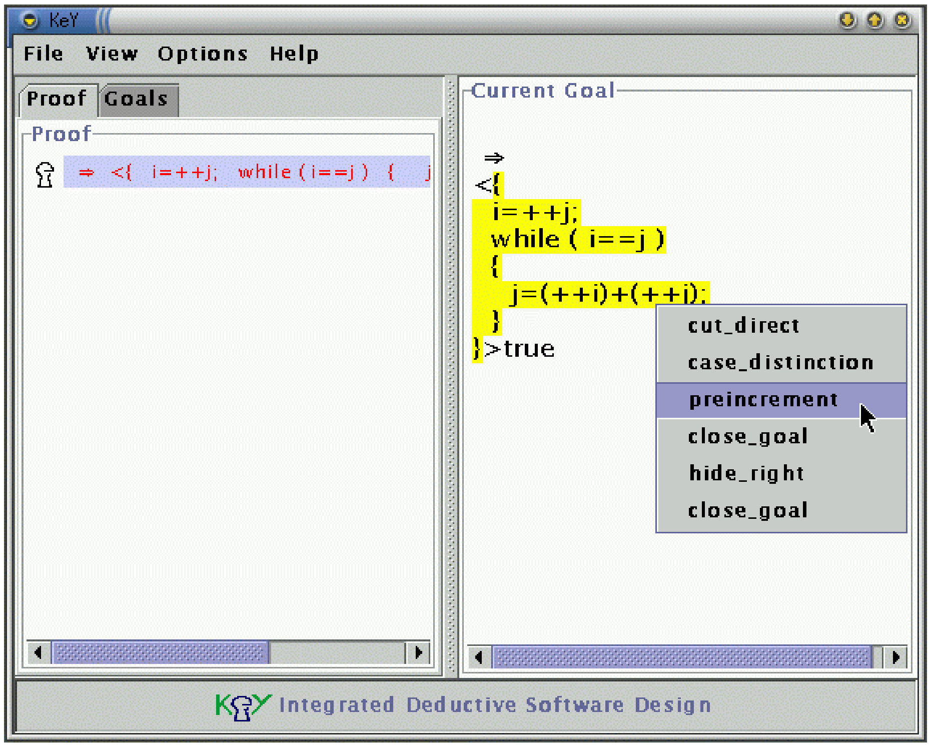
```
/**
 * @inva                                   0 and produceGemAmount > 0*/
public c
    pub
                                           ket();
    }
```

public class Stock extends Subject

Fig. 4. Options offered by the class menu

in which proof goals are displayed. The user conducts a proof by clicking on a formula or term that should be used in a proof step. The prover then displays a menu of rules applicable at that position, see Fig. 5. Together with the ability to define rules specific to the domain theory and datatypes involved in a particular application, this provides a very intuitive and easy to use prover interface.

A certain degree of automation is possible by activating heuristics that automatically perform possible rule applications. In the future we plan to extend the prover by powerful automated reasoning technology to reach a high degree of automation for common proof tasks.



```
⇒ <{  i=++j;  while (i==j)  {  j
```

```
⇒
<{
  i=++j;
  while ( i==j )
  {
    j=(++i)+(++i);
  }
}>true
```

Fig. 5. Applying a Rule in the KeY Prover