7

Natural Language Specifications

by

Kristofer Johannisson

This chapter describes how to use the KeY tool to bridge the gap between formal and informal specifications. Specifications need to be understood, maintained and authored by people with varying levels of familiarity with a formal specification language such as OCL. While a user of the KeY theorem prover should know a formal specification language, we cannot expect the same from a typical software developer, manager or customer. Hence there is need for specifications of different levels of formality, and we need to keep these different versions synchronised.

The KeY tool addresses these problems by making it possible to automatically translate formal (OCL) specifications to natural language (English and German),[1] and by providing a multilingual editor in which specifications can be edited in OCL and natural language in parallel.

This chapter starts with an overview of the natural language features of KeY in Section 7.1. Sections 7.2 and 7.3 describe basic principles and components. The multilingual editor is described in Section 7.4. We outline how domain specific vocabulary is handled in Section 7.5, and conclude with pointers to further reading and a summary in Sections 7.6 and 7.7.

7.1 Feature Overview

This section gives an overview of the natural language features of the KeY tool. While the later sections give a more thorough description, this should give you an idea about what is possible to achieve, and what limitations there are.

7.1.1 Translating OCL to Natural Language

Using the KeY tool, it is possible to translate all OCL specifications in a Borland Together project to natural language. Fig. 7.1 shows an example

[1] As explained in Section 7.5, the support for German is limited.

B. Beckert et al. (Eds.): Verification of Object-Oriented Software, LNAI 4334, pp. 317–333, 2007.

English translation provided by KeY, based on the class diagram and OCL specifications in Fig. 7.2 and 7.3. To get an unbiased impression of what the KeY tool can do, the reader is encouraged to consider the English translation before reading the formal description provided by the class diagram and OCL specifications.

The translation in Fig. 7.1 is produced automatically, no user interaction is required (unless we want to customise the translation). The output is formatted, using either LaTeX (as shown here) or HTML.

Note that the structure of the natural language text is very similar to the structure of the OCL specification, and has the same level of abstraction. We get a direct translation of the OCL specification, not an informal explanation of what it means.

For translating the domain specific concepts from a class diagram (classes, attributes, operations and associations) we use some heuristics which often work well, but not always. For instance, translating juniorLimit as "junior limit" is probably fine, while for unsuccessfulOperations we may prefer "number of unsuccessful operations" rather than the default translation "unsuccessful operations". We therefore allow user customisation of the translation of domain specific concepts, as described in Section 7.5.

The OCL to natural language translation can be accomplished either from within the KeY tool, or by using stand-alone command line tools.

7.1.2 Multilingual Specification Editor

The KeY tool provides a multilingual, syntax-directed editor for editing of OCL and natural language specifications in parallel. The editor is started from the KeY submenu of the context menu of any class or operation in Borland Together. It allows the user to construct 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 to the user in both OCL and natural language.

Figure 7.4 shows an example editing session, where we have just started editing an invariant for the class PayCard. There are three main parts of the editor window: the syntax tree display (top left), the linearisation area (top right), and the refinements menu (bottom). The syntax tree display shows the abstract representation of the specification, while the linearisation area presents the specification in OCL and natural language (English and German). Unfinished parts of the specification—called goals, or metavariables—are shown as question marks. The refinements menu presents possible ways of filling in the goals. Basic editing proceeds by selecting a goal (by clicking in the text or in the tree) and a refinement (by choosing from the hierarchical refinements menu). Since the tree is presented in both OCL and natural language, knowledge of OCL is not required for using the editor.

Assume that we wish to complete the unfinished invariant in Fig. 7.4 into for instance `balance >= 0` (OCL) or "the balance is at least 0" (English).

For the operation **charge (amount : Integer)** of the class **PayCard** ,
given the following precondition :

- *amount* is greater than 0

then the following postcondition should hold :

- the balance is at least the previous value of the balance

For the operation **available () : Integer** of the class **PayCard** ,
the following postcondition should hold :

- the result is equal to the balance or the unsuccessful operations is greater than 3

For the class **PayCardJunior** the following invariant holds :

- the following conditions are true
  - the balance is at least 0
  - the balance is less than the junior limit
  - the junior limit is less than the limit

For the operation **createCard () : PayCardJunior** of the class **PayCardJunior** ,
the following postcondition should hold :

- the limit of the result is equal to 10

For the operation **charge (amount : Integer)** of the class **PayCardJunior** ,
given the following precondition :

- *amount* is greater than 0

then the following postcondition should hold :

- if the previous value of the balance plus *amount* is less than the junior limit then:
  - the balance is incremented by *amount*

  otherwise:
  - the balance does not change and the unsuccessful operations is incremented by 1

For the operation **checkSum (sum : Integer) : Integer** of the class **PayCardJunior**
,
the following postcondition should hold :

- if the result is equal to 1 then:
  - *sum* is less than the junior limit

  otherwise:
  - *sum* is at least the junior limit

For the operation **complexCharge (amount : Integer)** of the class **PayCardJunior**
,
given the following precondition :

- *amount* is greater than 0

then the following postcondition should hold:

- if the previous value of the balance plus *amount* is less than the limit then:
  - *amount* is equal to the balance minus the previous value of the balance

  otherwise:
  - the balance does not change and the unsuccessful operations is incremented by 1

Fig. 7.1. Example natural language translation of OCL constraints

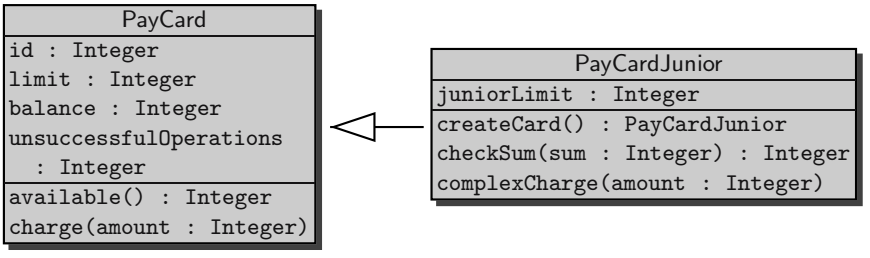

Fig. 7.2. Example class diagram

We would then proceed in a top-down fashion, first adding the comparison operator, and then the left and right argument to it. As shown in Fig. 7.4, the refinement "greater than or equal" is found in the submenu of "comparison operators". Figure 7.5 shows the editor after selecting this refinement.

We now have a specification `? >= ?` (OCL) or "? is at least ?" (English). Since the comparison operator takes two arguments, we have two new goals to fill in. In the figure, the leftmost goal has been selected. The refinements menu presents only type correct alternatives, which in this case means that we are only allowed to fill in instances of the OCL library type `Real` or any of its subtypes.

To complete the example, we have to fill in the left goal with `balance`, and the right with `0`, but we omit these steps here. The syntax editor is further explained in Section 7.4.

7.1.3 Suggested Use Cases

Translation of OCL to Natural Language

Being able to automatically translate OCL to natural language means that OCL specifications can be presented to people who do not know OCL. The translation can for instance be shown to a customer, who can then validate if it captures the desired behaviour of a system, or to a programmer who does not know OCL but needs to implement a system according to the specifications.

However, the provided natural language translations are on the same abstraction level as the original OCL specifications (as noted above). The intended reader of the translations must therefore be comfortable with this abstraction level. For instance, we cannot expect a translation of OCL specifications involving low-level implementation issues to be understandable to a customer.

The Multilingual Editor

The editor supports editing of OCL and natural language in parallel, and only allows the construction of specifications which are correct with respect

OCL

```
context PayCard::charge(amount : Integer)
pre: amount > 0
post: balance >= balance@pre

context PayCard::available() : Integer
post: result = balance or unsuccessfulOperations > 3

context PayCardJunior
inv: self.balance >= 0 and self.balance < juniorLimit
     and juniorLimit < limit

context PayCardJunior::createCard() : PayCardJunior
post: result.limit = 10

context PayCardJunior::charge(amount : Integer)
pre: amount > 0
post: if balance@pre + amount < juniorLimit
      then balance = balance@pre + amount
      else balance = balance@pre and
        unsuccessfulOperations = unsuccessfulOperations@pre + 1
      endif

context PayCardJunior::checkSum(sum : Integer) : Integer
post: if result = 1 then sum < juniorLimit
          else sum >= juniorLimit endif

context PayCardJunior::complexCharge(amount : Integer)
pre: amount > 0
post: if balance@pre + amount < limit
      then amount = balance - balance@pre
      else balance = balance@pre and
        unsuccessfulOperations = unsuccessfulOperations@pre + 1
      endif
```

OCL

Fig. 7.3. Example OCL constraints

to the OCL syntax and type system. It should therefore be useful for instance to a person who is not an OCL expert, but who needs to modify existing OCL specifications, as well as to people learning OCL.

For people who are already proficient in OCL, and who are not concerned with natural language translation, a traditional text editor is a more suitable tool for creating and modifying OCL specifications.

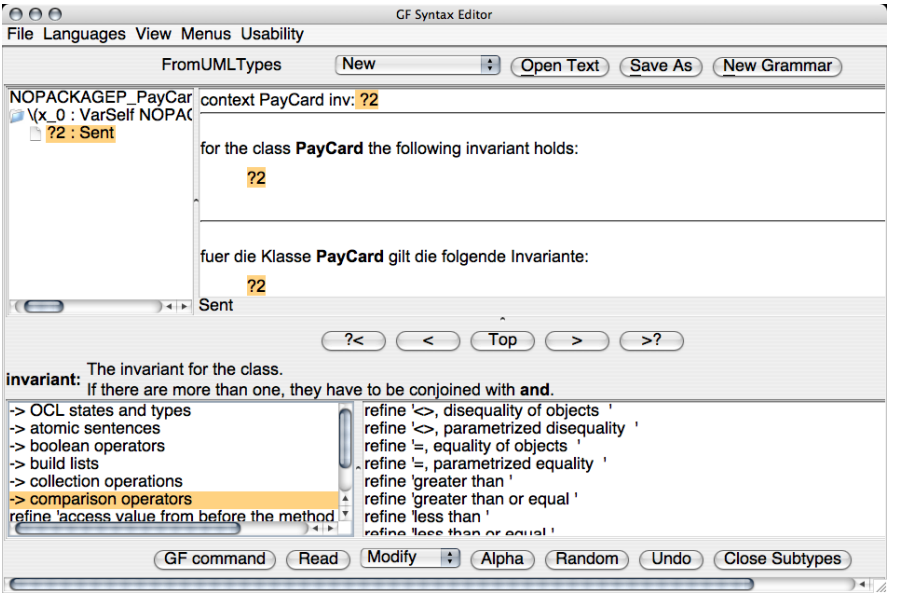

Fig. 7.4. Example editor session 1

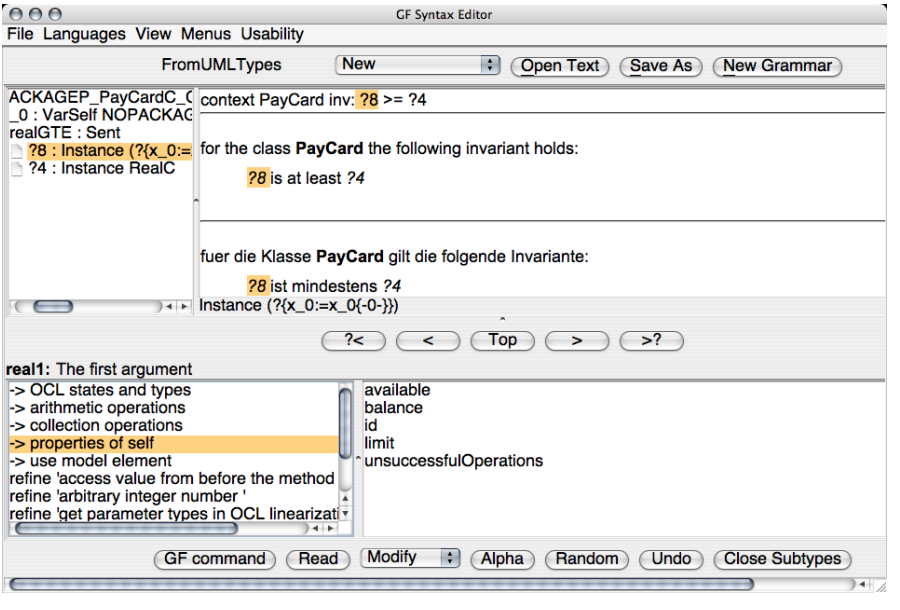

Fig. 7.5. Example editor session 2

OCL as Single Source

An important part of our approach is to use OCL as "single source": by creating and maintaining specifications in OCL (possibly using the multilingual editor), and then automatically translating them to natural language, we avoid the problem of having different versions of the same specification which need to be synchronised.

7.2 The Grammatical Framework

The natural language functionality in KeY is based on a multilingual grammar of specifications written in the Grammatical Framework (GF) formalism [Ranta, 2004].

A GF grammar defines abstract and concrete syntax. The abstract syntax gives rules for how to form abstract syntax trees. In a typical GF application grammar these trees are used as a non-linguistic, semantic representation of a restricted domain. In our case, we use abstract syntax trees to represent requirements specifications.

The concrete syntax defines how to present abstract syntax trees as expressions of a particular language, which can be a formal or a natural one. By having several concrete syntaxes for the same abstract syntax we get a multilingual grammar. We have defined concrete syntaxes for OCL, English, and German, which means that specifications represented in GF abstract syntax can be presented in these three languages.

The multilingual grammar for OCL, English and German specifications is written in the GF *formalism*. The GF *system* then provides functionality based on this grammar: it derives parsers and linearisers for the three languages as shown in Fig. 7.6. We can, for instance, parse an OCL specification (resulting in an abstract syntax tree) and then linearize it into English or German. Although we can also parse English or German specifications, the fragment of these languages described by our grammar is very small: we cannot expect to successfully parse arbitrary informal English or German specifications.

As noted above in Section 7.1.1, the structure of the natural language translation of an OCL specification provided by our tool is very similar to the structure of the original OCL specification. We can now explain the reason for this: the translation and the original specification both share the same abstract syntax, and the linearisation rules as defined by the concrete syntaxes for OCL, English and German cannot be arbitrarily complex. GF linearisation rules must be *compositional*, meaning that the linearisation of a tree is always expressed in terms of the *linearisation* of its subtrees, not the subtrees themselves.

An important aspect of our multilingual GF grammar is that it consists of a static as well as a dynamic part. The static part captures the OCL type system, basic OCL constructions such as invariants or if-then-else expressions,

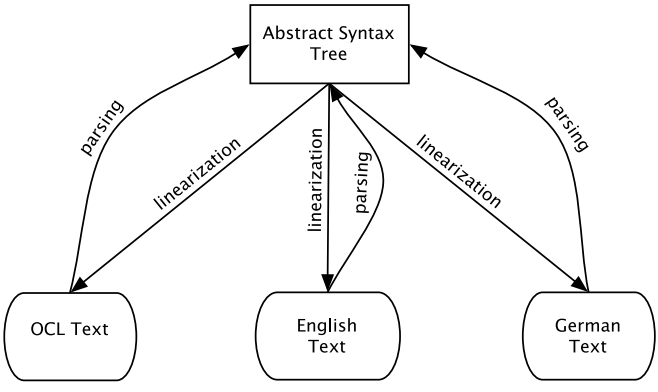

Fig. 7.6. GF parsing and linearisation

and the predefined types and operations of the OCL library. The dynamic part is a description of the domain specific concepts—classes, attributes, operations and associations—found in the class diagram of the current Borland Together project. This part of the GF grammar is generated from the current class diagram. Section 7.5 describes the basics of this generation, and how it can be customised.

7.2.1 GF Examples

To illustrate the general principles of GF we give some examples of abstract and concrete syntax rules, loosely based on our multilingual GF grammar (without explaining all the details of the GF formalism).

In the abstract syntax, we want to represent the domain of OCL specifications, for instance, we have to represent classes, expressions and queries. The following is one way to do this in GF abstract syntax:

GF

```
cat Class;
cat Expr (c:Class);
fun IntegerC : Class;
fun maxQ : (x,y : Expr IntegerC) -> Expr IntegerC;
fun intLit : Int -> Expr IntegerC;
```

GF

This defines two categories `Class` and `Expr`: If c is a `Class`, then `Expr` c represents expressions of type c (`Expr` is a *dependent type*, since it requires an argument). Using these two categories, we can then introduce the GF functions `IntegerC` and `maxQ` to represent the OCL library class `Integer`, and the query `max` (which returns the maximum of two integers). The function `intLit` allows us to use the built-in integer type of GF for integer literals.

The concrete syntax then gives rules for how to linearize abstract syntax trees in OCL, English, or German. Here we consider some examples for English. Writing GF concrete syntax is much like working in a functional programming language with record-types, strings, and finite algebraic data types. We must provide a record type for each category in the abstract syntax, and a function building records of the correct type for each abstract syntax function.

To express that a class in OCL corresponds to a noun in English we use the following concrete syntax:

GF

```
param Number = Sg | Pl;
lincat Class = {s : Number => Str};
lin IntegerC = {s = table {Sg => "integer"; Pl => "integers"}};
```

GF

Here we introduce a parameter type for representing singular and plural number. The `lincat` judgement states that the category `Class` corresponds to records containing a field `s`, which is a string inflected in number (a finite function from `Number` to `Str`). Then, `IntegerC` is linearised as an inflection table with the singular and plural form of the noun "integer".

To linearize an abstract tree (`maxQ` x y), where x and y have type `Expr IntegerC`, as an English noun phrase "the maximum of x and y", we give the following rules to complete our small GF grammar (we also need to include a linearisation for the `intLit` function):

GF

```
lincat Expr = {s : Str};
lin maxQ x y = {s = "the" ++ "maximum" ++ "of" ++ x.s ++
                    "and" ++ y.s};
lin intLit i = {s = i.s};
```

GF

Loading this grammar into the GF system, we can then parse for example the string "the maximum of 2 and 7", which gives us the abstract syntax tree (`maxQ` (`intLit` 2) (`intLit` 7)).

When writing larger GF application grammars, such as the one used to link OCL and natural language, you normally work on a higher level of abstraction than in these small examples. Instead of defining your own types for nouns and number (or for gender and case, as we would need for German), you make use of the resource grammar library which is supplied with the GF system. This library provides an interface of linguistically motivated types (e.g., types for number, gender, nouns, verbs and sentences) and functions (e.g., for building a sentence from a verb phrase and a noun phrase). Implementations of the interface are provided for several languages. By making

use of the resource library interface we can therefore share code between the English and German concrete syntax in our multilingual grammar.

7.3 System Overview

There are a number of components involved in linking OCL to natural language: a multilingual GF grammar, the GF system, a syntax-directed editor, a GF grammar generator taking class diagrams as input, and also a stand-alone OCL parser and typechecker. Fig. 7.7 shows how these components relate to each other in terms of input and output.

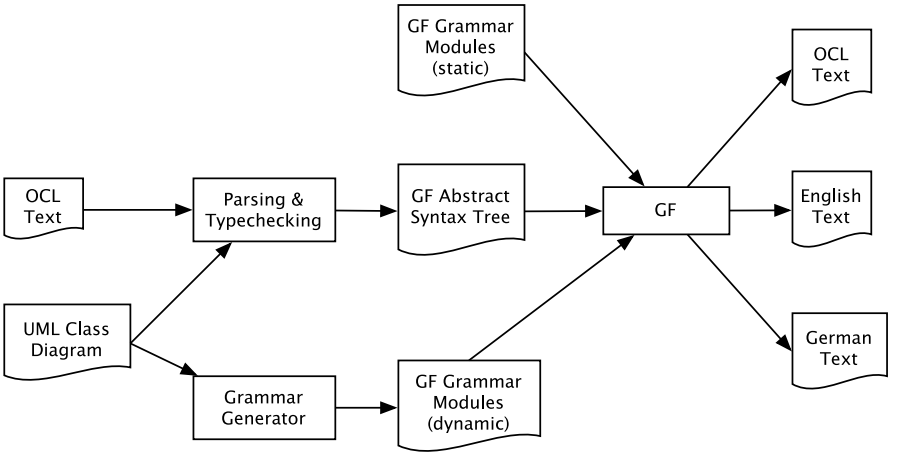

Fig. 7.7. System components

Grammar Generation

All functionality relies on the existence of the GF grammar for specifications, and as described in Section 7.2 above, parts of this grammar are dynamically generated from a class diagram. The class diagram is in turn extracted from Borland Together.

OCL Parsing and Typechecking

When translating an OCL specification to natural language, or when starting the multilingual editor for a given OCL specification, the first step is to turn the OCL text into a GF abstract syntax tree. To do this, we are not using the parser automatically derived by GF, but a custom parser and typechecker. Note that typechecking OCL requires also the class diagram as input.

There are a number of reasons for using a custom parser and typechecker: we need to work around a limitation in the parser derived by GF for our particular grammar, it makes it simpler to deal with all the various implicit forms in OCL concrete syntax, and it also makes it possible to give better error messages when encountering type errors. Finally, we expect the external parser, which is derived using a standard context-free parser generator, to be more efficient than the GF parser when parsing large specifications.

GF

The input to GF is the grammar (static and dynamic parts) and an abstract syntax tree. To translate OCL to natural language, the tree is then just linearised into English and German. In case of the editor, the user manipulates the syntax tree in the editor, while viewing the result in OCL, English and German in parallel.

7.4 The Multilingual Editor

The multilingual editor allows you to edit specifications in OCL, English and German in parallel. It is based on the generic GF syntax editor [Khegai et al., 2003] but has been customised for the domain of software specifications [Daniels, 2005]. The editor is started from the KeY submenu of the context menu of any class or operation in Borland Together. If the class or operation is already annotated with an OCL specification, it is parsed and shown in the editor, otherwise the editor starts up with an empty invariant (for classes) or with empty pre- and postconditions (for operations). The editor is intended for editing the OCL specification of one class or operation at a time.

7.4.1 Syntax-Directed Editing

The editor is syntax-directed: editing consists of manipulating the abstract syntax tree of a specification, rather than a string of characters as in a typical text editor. The tree is at all times presented in OCL, English and German, as defined by the GF grammar for specifications (the user can choose which languages to show). Since we are editing a syntax tree, we can only construct syntactically correct specifications. The editor also includes a type system and ensures that the syntax tree is always type-correct.

There are two basic ways of manipulating a syntax tree in the editor: *refinement* (top-down editing) and *wrapping* (bottom-up).

7.4.2 Top-Down Editing: Refinement

Refinement consist of selecting a goal—an unfinished part of the tree, displayed as a question mark—and filling in this goal by selecting a refinement

from a menu. The selected refinement may in turn contain new goals which need to be filled in.

Each goal has a type, and the refinements menu only lists refinements of this type. A type can for instance be "integer expressions", "sentences", or "attributes". The types and refinements available are given by the underlying GF grammar.

We consider the example from the beginning of this chapter again, as shown in Fig. 7.8. In the upper left part of the editor window we see the abstract syntax tree of a specification, which is presented in OCL and natural language in the upper right part of the window. There are two unfinished parts (goals), one for each argument to the comparison operator (>=).

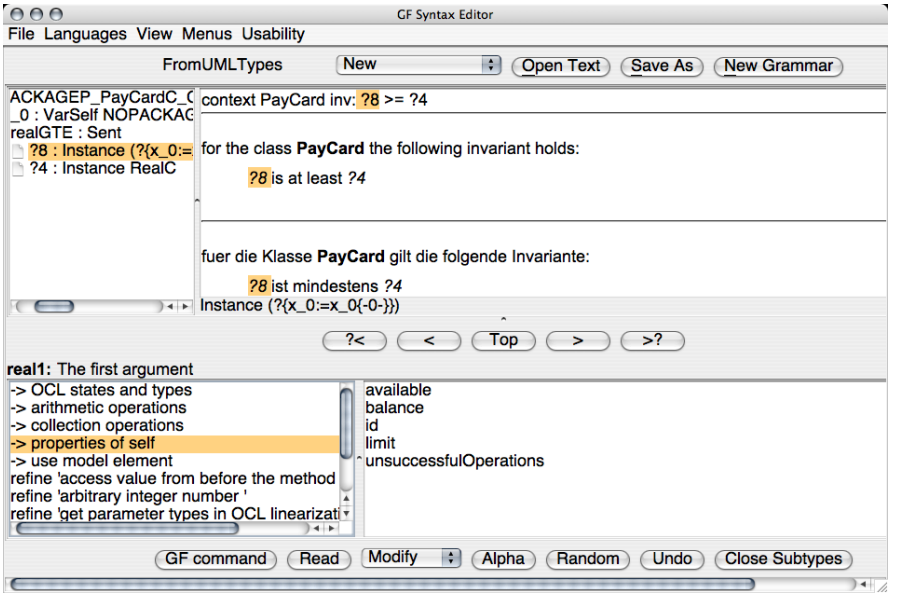

Fig. 7.8. Editing by refinement

7.4.3 Bottom-Up Editing: Wrapping

Wrapping consists of selecting any part of the syntax tree—with or without unfinished parts—and replacing it with a new construction, which contains the previously selected subtree as a part. For instance, if we have constructed the invariant `self.balance >= 0`, and would like to add that balance should also be smaller than limit, we do this by wrapping it using and. The first step is to select the subtree corresponding to `self.balance >= 0`, as shown in Fig. 7.9.

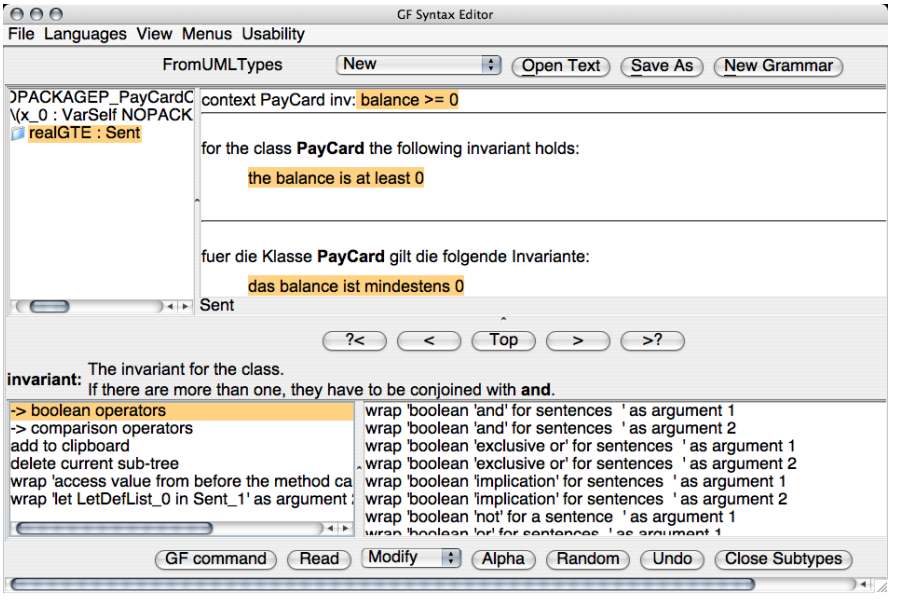

Fig. 7.9. Editing by wrapping, step 1

The current selection is now a sentence. Since `and` is a construction which takes two sentences into a new sentence, we can wrap the current selection using `and`. This is done by selecting "wrap boolean 'and' for sentences as argument 1" in the refinements menu. The result is shown in Fig. 7.10: the previously selected subtree `balance >= 0` has now been wrapped as the first argument to `and`, resulting in `balance >= 0 and ?`.

7.4.4 Other Editor Features

The editor also includes other features, for instance, as you would expect there is a clipboard for copying and pasting syntax trees, as well as an undo command. Another feature is refinement by parsing: instead of filling in a goal by selecting a refinement, one can enter a text string. The string is then parsed and (if parsing was successful) the goal is filled in with the resulting syntax tree. In this case, it is the parser derived by GF which is being used, not the custom OCL parser and typechecker.

7.4.5 Expressions and Sentences

The editor makes a distinction between expressions and sentences. Expressions are instances of any of the classes from the class diagram, or of the OCL library types such as Integer or Boolean. Sentences are used to express invariants, pre- and postconditions. An example expression is `self.balance`

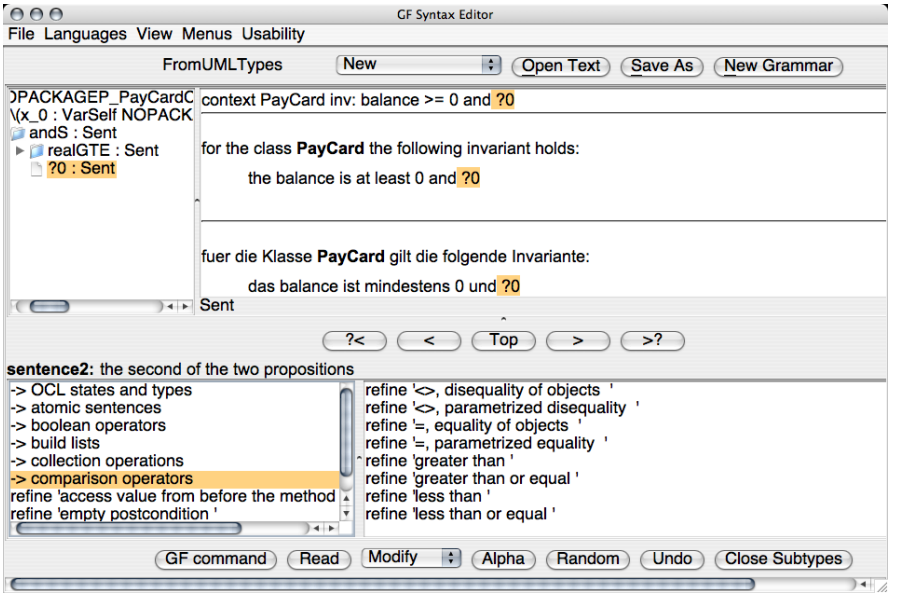

Fig. 7.10. Editing by wrapping, step 2

("the balance"), an example sentence is `self.balance >= 0` ("the balance is at least 0"). We mention this distinction since it is not present in OCL itself: there is no concept of sentences in the OCL language specification. Instead, expressions of type Boolean are used for invariants, pre- and postconditions. However, in the editor expressions and sentences are two different types: goals of expression type cannot be filled in with a sentence, and vice versa.

All OCL library operations as well as all domain specific attributes and operations which return Boolean from the point of view of OCL are considered as sentences in the editor. It is always possible to convert a sentence into a Boolean expression, but this has to be done explicitly.

7.4.6 Subtyping

The OCL type system includes subtyping: wherever an expression of a type T is expected, we can also use an expression of type T' as long as T' is a subtype of T. For instance, the OCL comparison operators <, >, <=, and >= are all defined for the class Real. However, since Integer is a subtype of Real, we can also use them to compare integers.

GF has no built-in notion of subtyping. In the GF grammars for specifications, this problem is solved by including explicit coercions (typecasts). These coercions are part of the abstract syntax tree, but are not visible in the OCL or natural language rendering of the tree. The editor usually creates these coercions automatically without requiring user interaction, but sometimes—

in particular when an existing specification is modified—the user has to be aware of the coercions.

7.5 Translation of Domain Specific Concepts

As previously mentioned, the translation of domain specific concepts is defined by GF grammar modules which are generated from the class diagram of the current project in Borland Together. This generation is based on some simple rules described below. If the automatically derived translation is not appropriate, it can be customised by hand.

The generation and customisation are both based on the assumption that the language used in class diagrams is English, and that OCL specifications are to be translated into English. The generated GF modules can be used with the German GF grammar anyway, but the resulting German contains fragments of the English used in the class diagram (as seen in the syntax editing examples in Section 7.4).

7.5.1 Grammar Generation

The grammar generation provides default translations for the concepts—classes, attributes, operations, and associations—in a class diagram. Currently, this generation is based on a few simple rules:

- Classes are treated as common nouns, or as common noun phrases. In case the name of the class is capitalised (as in PublicKey), it is split into separate words, where the last word is considered as a noun which is modified by the other words. For instance, a class Person is treated as a common noun "person", while a class PublicKey is treated as a common noun phrase "public key".
- Properties (attributes, operations and associations) are treated as noun phrases, except for Boolean properties, which are treated as sentences. Capitalization is used also for properties, e.g., an attribute juniorLimit is translated as the noun phrase "junior limit". Boolean properties which start with "is-", e.g., isEmpty or isValidated, are treated as adjectives (e.g., "... is empty", "... is validated").

7.5.2 Customising the Translation

If the translation provided by the generated grammar modules is not appropriate, it can be customised by hand. We plan to make it possible to perform such customisation by having the user add annotations to the Borland Together class diagram, but at present there is no such functionality. To

customise the translation, one must instead modify the generated GF grammar files directly. However, as described below, this can be done without requiring GF expertise.

Customisation is done on the level of concrete syntax. The generated concrete syntax makes use of a grammar-level API, which contains functions for common constructions. This API abstracts from the complexity of the rest of the grammar. To modify the generated concrete syntax it is therefore enough to have an understanding of the API, it is not necessary to be a GF expert.

This API is described in detail on the web site for the OCL-Natural Language tool (⇒ Sect. 7.6), here we just consider a small example. As mentioned in the previous example in Section 7.1.1, the default translation of the unsuccessfulOperations attribute of the PayCard class is "unsuccessful operations", although "number of unsuccessful operations" might be a more natural translation. The generated GF concrete syntax for unsuccessfulOperations is the following:

GF

```
lin unsuccessfulOperations = mkSimpleProperty (adjCN
        ["unsuccessful"] ((strCN ["operations"])));
```

GF

The left hand side of this linearisation judgement is simply the name of the construction in the abstract syntax which represents unsuccesfulOperations. The right hand side gives the linearisation of this construction, expressed using the functions `mkSimpleProperty`, `adjCN` and `strCN` of the grammar API.

This generated linearisation can be changed to produce "the number of unsuccessful operations" instead by using the `ofCN` and `strCN` functions:

GF

```
lin unsuccessfulOperations = mkSimpleProperty (ofCN
    (strCN "number") (adjCN ["unsuccessful"] ((strCN
    ["operations"]))));
```

GF

7.6 Further Reading

The basic motivations and design principles of a GF based tool to link OCL and natural language are described in a paper by Hähnle et al. [2002]. A later paper shows that the tool scales well enough to handle a case study: translating OCL specifications of the JavaCard API to natural language [Burke and Johannisson, 2005]. There is also a web site for the tool.[2]

[2] `http://www.key-project.org/oclnl/`

7.7 Summary

The KeY tool makes it possible for people who are not OCL experts to create and maintain OCL specifications, by providing a multilingual, syntax-directed editor in which specifications can be edited in OCL and natural language in parallel. OCL specifications can also be translated to natural language independently of the editor, which enables people who have no knowledge of OCL to make use of formal specifications.

A limitation is that the provided natural language translation has roughly the same structure and level of abstraction as the original OCL specification. In this sense, we do not provide informal explanations of formal specifications. Also, automatic formalisation of arbitrary informal specifications falls outside the scope of the KeY tool.

The natural language tools are built around a multilingual Grammatical Framework grammar for specifications in OCL, English and German. The translation of domain-specific concepts can be customised on the grammar level.