Translating Formal Software Specifications to Natural Language

A Grammar-Based Approach

David A. Burke and Kristofer Johannisson

Department of Computing Science, Chalmers University of Technology and Göteborg University, SE-41296 Göteborg, Sweden
krijo@cs.chalmers.se

Abstract. We describe a system for automatically translating formal software specifications to natural language. The system produces natural language which is acceptable to a human reader, and it supports by-hand optimization by users who are not experts of our system. The translation system is implemented using the Grammatical Framework, a grammar formalism based on Martin-Löf's type theory. We show that this grammar-based approach scales well enough to handle a non-trivial case study: translating the Object Constraint Language specifications of the Java Card API into English.

1 Introduction

The goal of this work is to automatically translate formal software specifications into natural language. Our motivation is a wish to link formal specifications (as needed for formal methods) to informal ones (as found in software engineering practice). Our work is a part of the KeY project [1], which integrates formal software specification and verification into the industrial software engineering processes.

We have implemented a system, earlier described in [2], using the Grammatical Framework (GF), a grammar formalism based on Martin-Löf's type theory [3, 4]. In this paper we show that our grammar-based approach scales to such a degree that we can handle a non-trivial case study.

The case study consists of specifications for the Java Card API [5], written in the Object Constraint Language, which have been translated into English by our system. To improve the quality of the translation, we have extended our system with formatting and automatic generation of grammar modules for domain-specific vocabulary, which can then be modified without requiring GF expertise. We have also added various simple stylistic improvements inspired by techniques familiar from Natural Language Generation [6]. As far as possible, all these improvements are implemented in a declarative way in the GF grammars; some of them also require manipulation of syntax trees by a separate program, external to the grammars.

P. Blache et al. (Eds.): LACL 2005, LNAI 3492, pp. 51–66, 2005.
© Springer-Verlag Berlin Heidelberg 2005

1.1 Paper Overview

We start with background on the GF formalism, formal specifications, the KeY project, and the Java Card API specification case study in Sect. 2. In Sect. 3 we then give a motivating example from the case study, showing an example formal specification, as well as English translations before and after our improvements.

Sect. 4 explains the overall architecture of the translation system, while Sect. 5 is concerned with grammar engineering: how to design the grammar based system to meet our goals.

Sect. 6 describes related work, mainly Natural Language Generation. Some figures on the size of the case study are given in Sect. 7, and we then conclude in Sect. 8.

2 Background

2.1 The Grammatical Framework

The GF Formalism. The Grammatical Framework (GF) is a formalism for defining grammars [3]. A GF grammar consists of one part which describes abstract syntax, and another part which describes concrete syntax. The abstract syntax part is formulated in a version of Martin-Löf's type theory [4], and can be seen as a description of how to construct abstract syntax trees. The concrete syntax then consists of *linearization* rules telling how to present these trees as expressions of a particular language. This is a distinguishing feature of GF as compared to many other grammar formalisms: grammars are written from the perspective of linearization rather than parsing. In fact, we can consider the GF formalism as a linearization (or generation) oriented typed functional language.

The concrete syntax is based on record types, strings and finite parameter types, enabling the representation of e.g. inflection tables and discontinuous constituents. A central notion in GF is *compositionality*: the linearization of a tree is always expressed in terms of the linearization of its subtrees, we have no access to the subtrees themselves in a linearization rule. This restriction is important for the implementation of GF.

By having multiple concrete syntaxes for the same abstract syntax we achieve *multilinguality*: we can present the same tree in several languages in parallel, and we can translate (within the language fragment described by the grammar) by parsing using one concrete syntax and linearizing with another. Compositionality imposes a restriction of structural similarity on the languages sharing the same abstract syntax, however, this restriction is to some degree countered by the expressiveness of the concrete syntax.

The GF System. The GF system [7] provides functionality such as parsing and linearization for grammars written in the GF formalism. The system also includes a syntax editor [8] in which the user can load a GF grammar and then edit the abstract syntax trees described by the grammar. The trees are all the time presented in the languages defined by the concrete syntaxes of the grammar. By

editing the abstract syntax tree and observing the results in a familiar language, a user can then interactively produce texts in foreign languages.

The GF Resource Grammar Library. An important part of the GF project is the resource grammar library [9], which provides an API of types and functions for common linguistic structures. There are resource grammars available for English, Finnish, French, German, Italian, Russian and Swedish, which to a large extent share the same interface.

Grammar Engineering. A typical GF application grammar describes a well-defined fragment of natural language for a restricted domain, e.g. in our case software specifications. The resource grammar library provides a division of labour: the author of an application grammar can be a domain expert, who does not need to be familiar with linguistic details. His or her task is to come up with an abstract syntax which models the domain, and to link the abstract syntax to concrete language by using the resource grammars. The linguistic expert is in turn responsible for the implementation of the resource grammars, where no knowledge of a particular domain is needed.

2.2 Formal Specifications and the KeY Project

The KeY Project. The KeY project [1] attempts to integrate formal software specification and verification into the industrial software engineering processes. The starting point is a commercial CASE (Computer Aided Software Engineering) tool, which is augmented by capabilities for formal specification and verification. The ultimate goal is to make the verification process transparent for the user with respect to the informal object-oriented model.

Formal and Informal Specifications. Formal methods require formal specifications, but in software engineering practice, informal specifications are commonly used. We cannot expect everyone who needs to deal with specifications—e.g. customers, managers, or software engineers—to master a formal notation (cf. [10] p. 131 "... most customers don't understand formal specifications and are reluctant to accept it as a system contract"). This motivates the need for a systematic link between formal and informal specifications: to support authoring of specifications as well as synchronizing and maintaining formal and informal versions of specifications, and to present the specifications to different audiences using different levels of formality.

The Object Constraint Language. The Object Constraint Language (OCL) is a formal specification language used to specify precise requirements for object-oriented software systems [11]. It is a sub-standard of the Unified Modelling Language (UML) [12]. An OCL specification is always given in the context of some particular UML model,[1] and it consists of a boolean expression which

[1] In this paper, a *UML model* is simply a class diagram, containing classes, attributes, methods and associations. Side-effect free methods are called *queries*.

is used as an invariant of a class, or as a pre-condition or post-condition of a method. The attributes, queries and associations from the UML model, as well as a library of predefined types (e.g. integers, strings and collections) are available for constructing OCL expressions. For example, given a class `OwnerPIN`, with attributes `tryCounter` and `maxTries`, we can specify the requirement "the try counter is at most the maximum number of tries" using the OCL:

```
context OwnerPIN
inv: self.tryCounter <= self.maxTries
```

As we see in this example, each OCL expression is given in the context of a particular class, and the expression `self` refers to an instance of that class.

2.3 The Java Card API Specification

Java Card technology [5] allows software developers to write Java programs that run on smart cards and other devices with very limited memory and processing capabilities. The Java Card API (Application Programming Interface) is a set of library classes used in Java Card programs. It is a subset of the standard Java API and is specifically designed for smart card programming.

Due to the size and nature of the applications that use Java Card, formal methods could be useful in verifying the correctness of these programs. With this in mind, OCL constraints have been defined for the Java Card API in [13] (based on JML specifications in [14]). These OCL specifications provided the basis for a case study using our translation tool. The specifications of 37 Java classes were fully translated, and examples from this are used throughout this paper. Details on the case study are available in [15] and on the web [16].

3 Motivating Example

In this section we will take an OCL specification from the Java Card API case study (Fig. 1) and show how it gets translated into natural language. For comparison, we start with a translation produced with an earlier version of our system, Fig. 2. Then we discuss some possible improvements of this translation, which leads to the translation in Fig. 3, which is the output from the current system. The machinery behind the improvements will be explained later.

3.1 The OCL Specification

We consider the OCL specification for the method `check` of the class `OwnerPIN`. `OwnerPIN` stores the PIN code of a smart card, and keeps track of the maximal number of attempts allowed to present the correct PIN before the card is locked. The purpose of the method `check` is to compare a given PIN number with the PIN value in the `OwnerPIN` class itself. If they match and the PIN is not blocked, it sets the validated flag and resets the try counter to its maximum. If it does not match, it decrements the try counter and, if the counter has reached zero, blocks the PIN. The try counter is specified in the first element of

```
context OwnerPIN
def: let tryCounter = self.triesLeft->at(1)

context OwnerPIN::check(pin: Sequence(Integer),
         offset: Integer, length: Integer): Boolean
post: self.tryCounter = 0 implies result = false
post: (self.tryCounter > 0 and pin <> null and offset >= 0 and length >= 0
         and offset+length <= pin->size()
         and Util.arrayCompare(self.pin, 0, pin, offset, length) = 0
      ) implies (result = true and self.isValidated() and tryCounter = maxTries)
post: (self.tryCounter > 0 and not (pin <> null and offset >= 0 and length >= 0
            and offset+length <= pin->size()
            and Util.arrayCompare(self.pin, 0, pin, offset, length) = 0)
      ) implies (not self.isValidated() and self.tryCounter = tryCounter@pre-1 and
          (( not excThrown(java::lang::Exception) and result = false)
             or excThrown(java::lang::NullPointerException)
             or excThrown(java::lang::ArrayIndexOutOfBoundsException)))
```

Fig. 1. OCL specification from the Java Card API

the attribute `triesLeft`, which is an array. The validated flag can be accessed using the `isValidated()` method. The PIN comparison can be done using the `arrayCompare()` method which is defined in the `Util` class of the JavaCard API. Fig. 1 shows the OCL specification of `check` (including a definition of a helper attribute `tryCounter`).

3.2 A First Attempt

In Fig. 2 we show the translation of the OCL specification produced by an earlier version of our system. The English text is basically correct, but it is clumsy and very hard to read.

for the class OwnerPIN introduce the following definition : the tryCounter is defined as the element at index 1 of the triesLeft of the ownerPIN for the operation check (pin : Seq(Integer) , offset : Integer , length : Integer) : Boolean of the class javacard::framework::OwnerPIN the following holds : the following postconditions should hold : (*) if the tryCounter of the ownerPIN is equal to 0 , the result is equal to false (*) if the tryCounter of the ownerPIN is greater than 0 and pin is not equal to null and offset is at least 0 and length is at least 0 and offset plus length is at most the size of pin and the query arrayCompare (the pin of the ownerPIN , 0 , pin , offset , length) to Util is equal to 0 , the result is equal to true and the query isValidated ( ) holds for the ownerPIN and the tryCounter of the ownerPIN is equal to the maxTries of the ownerPIN (*) if the tryCounter of the ownerPIN is greater than 0 and it is not the case that pin is not equal to null and offset is at least 0 and length is at least 0 and offset plus length is at most the size of pin and the query arrayCompare (the pin of the ownerPIN , 0 , pin , offset , length) to Util is equal to 0 , it is not the case that the query isValidated ( ) holds for the ownerPIN and the tryCounter of the ownerPIN is equal to the tryCounter of the ownerPIN at the beginning of the Operation minus 1 and it is not the case that an exception is thrown and the result is equal to false or a nullPointerException is thrown or an arrayIndexOutofBoundsException is thrown

Fig. 2. Translation of OCL specification (before)

3.3 An Improved Translation

Fig. 3 shows an improved version of the translation: the output of the current version of our system. Below we go through the improvements made. Each one is quite simple in itself, but the end result is in our opinion a text of acceptable quality, which shows that our approach works for non-trivial specifications.

for the class **OwnerPIN** introduce the following definition :

- the try counter is defined as the element at index 1 of the `triesLeft` attribute

for the operation **check (pin : Sequence(Integer) , offset : Integer , length : Integer) : Boolean** of the class **javacard::framework::OwnerPIN** ,
the following post-conditions should hold :

- if the try counter is equal to 0 then this implies that the result is equal to false
- if the following conditions are true
  - the try counter is greater than 0
  - *pin* is not equal to null
  - *offset* is at least 0
  - *length* is at least 0
  - *offset* plus *length* is at most the size of *pin*
  - the query `arrayCompare ( the pin , 0 , pin , offset , length )`[1] on `Util` is equal to 0

  then this implies that the following conditions are true
  - the result is equal to true
  - this owner PIN is validated
  - the try counter is equal to the maximum number of tries
- if the try counter is greater than 0 and at least one of the following conditions is not true
  - *pin* is not equal to null
  - *offset* is at least 0
  - *length* is at least 0
  - *offset* plus *length* is at most the size of *pin*
  - the query `arrayCompare ( the pin , 0 , pin , offset , length )`[2] on `Util` is equal to 0

  then this implies that the following conditions are true
  - this owner PIN is not validated
  - the try counter is equal to the previous value of the try counter minus 1
  - at least one of the following conditions is true
    - an exception is not thrown and the result is equal to false
    - a null pointer exception is thrown
    - an array index out of bounds exception is thrown

[1] Compares the specified source array, beginning at the specified position, with the destination array beginning at the specified position from left to right. A result of 0 indicates that the arrays are equal.

[2] Compares the specified source array, beginning at the specified position, with the destination array beginning at the specified position from left to right. A result of 0 indicates that the arrays are equal.

Fig. 3. Translation of OCL specification (after)

Formatting. Two of the most important problems in Fig. 2 are (1) the specification is just a big piece of text, where the structure is very hard to discern, and (2) it is hard or impossible to determine the scope of the and:s and or:s, which

makes the specification ambiguous. To address these problems we introduce formatting: line breaks are inserted, keywords are printed in bold and arguments to the method are italicized. Furthermore, lists of constraints, as well as sequences of and/or statements are made into bullet lists. The formatting consists of HTML or LaTeX tags in the text, what we see in Fig. 3 is the LaTeX version.

Negation. The scope of the negations is hard to determine, and negating sentences as in e.g. "it is not the case that an exception is thrown" is a clumsy construction. Using "an exception is not thrown" instead solves both these problems.

Making Use of the Context. In Fig. 2 the text "of the ownerPIN" is used very frequently. Since the specification is given as postconditions in the context of a method of the class `OwnerPIN`, we should be able to just leave out all occurrences of "of the ownerPIN", resulting in a much less repetitive text. (In OCL we are allowed to do the same thing, by leaving out `self`.) This can be seen as a simple case of referring expressions generation [6].

Domain-Specific Vocabulary. Based on the type and capitalization of identifiers, we can improve the translation of domain-specific vocabulary. For instance, the class `OwnerPIN` can be automatically translated as "owner PIN", instead of just "ownerPIN" as in Fig. 2.

The attributes `triesLeft` and `maxTries` are similarly translated as "the tries left" and "the max tries" by default. Although these translations are quite adequate, we can improve this even further by making manual changes. Thus, using our system we can by hand change the translation to "the triesLeft attribute" and "the maximum number of tries", for these attributes.

Two methods are used in this OCL constraint: `isValidated` and `arrayCompare`. For `isValidated`, which returns a boolean, we introduce some simple heuristics which by default translates it to "... is validated" instead of "the query isValidated() holds". The second method used, `arrayCompare`, gets linearized to "the query `arrayCompare ( the pin, 0 , pin , offset , length )` on `Util`". Unfortunately this method is not so easily translated into simple English. The task it carries out does not fit nicely as part of the translated constraint. To solve this problem, we use the 'note' facility provided in the grammar. We can by hand add a note for the method, and this will then be displayed as a tool-tip when HTML formatting is used or, as in this case, as a footnote when LaTeX formatting is used (a current limitation is the needless duplication of footnotes).

4 System Overview

The system is built around a GF grammar for specifications: there is an abstract syntax giving rules for how to form abstract syntax trees of specifications, as well as three concrete syntaxes to present abstract syntax trees in OCL, English and German, respectively.

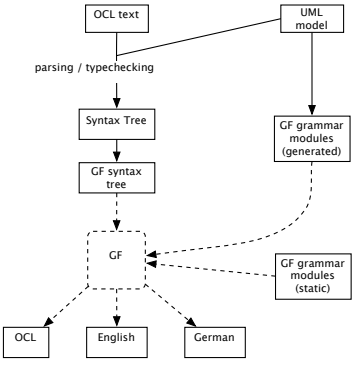

Fig. 4. From OCL to Natural Language

Given this grammar, the GF system provides us with a syntax editor where we can edit specifications in OCL, English and German in parallel. We also get parsers and linearizers for OCL and (fragments of) English and German, which we can use for translating e.g. OCL specifications into English, by first parsing OCL into an abstract syntax, and then linearizing the tree into English.

However, our system does not just consist of a GF grammar and the functionality provided directly by GF. What makes things more complex is that (1) parts of the grammar are dynamically generated depending on the context (the grammar is not closed), and (2) we use a separate program for parsing OCL specifications and (3) turning them into GF abstract syntax trees. These external programs (as well as the GF system itself) are implemented in the functional language Haskell. Fig. 4 shows the overall structure of the system.

There are two current prototypes of our system: one which allows syntax editing of specifications in OCL and English inside KeY, another which allows batch-translation from OCL to English (yet to be integrated more closely with KeY). The concrete German grammar has so far only been used for small examples [17]. The system is available for download [16].

Syntax editing of specifications is briefly described in [2], except for the integration with the KeY system. In this paper we focus on taking existing OCL specifications into English.

4.1 External Programs

Grammar Generation. An OCL specifications uses domain specific vocabulary as defined by a UML model. When a user adds e.g. a class or an attribute to the model, he also extends the language of specifications of that model. We therefore generate GF grammar modules from the UML model to dynamically extend the grammar with domain specific vocabulary. This is described in more detail in Sect. 5.2. The general idea of dynamically extending a grammar with user-defined concepts is also used in [18].

External Parser. Given an OCL specification and a UML model, the OCL is first parsed using a standard context free parser, and then type checked with respect to the UML model, resulting in an annotated syntax tree of the OCL specification. This step is described in detail in [19].

The original motivation for adding an external parser was that the parser derived by GF for our particular grammar had termination problems.[2] However, there are also more general reasons for using an external parser: (1) *Efficiency:* An external, context free parser for a formal language—in our case OCL—is more efficient than a parser derived from a GF grammar. (2) *Modularity:* The GF abstract syntax does not have to handle all particularities of OCL. For instance, OCL has various implicit forms which require disambiguation (see e.g. [19]). This can be done by the external parser and typechecker.

Transformation Into GF Trees. The annotated trees returned by the external parser are transformed into GF abstract syntax trees. The context free structure for most parts maps into the GF abstract syntax in a straightforward way. However, we also perform some structural transformations in order to improve the quality of the natural language, e.g. the transformation described in Sect. 5.3. These transformations could probably be avoided by extending the GF abstract and concrete syntax instead, but we believe that expressing the transformations in the Haskell programming language instead of the GF formalism is in this case a simpler and more modular solution.

5 Grammar Engineering

We start this section by giving a very brief introduction to our GF grammar, to give a general idea of what writing an application grammar for specifications amounts to (Sect. 5.1). We then explain how the improvements described in the motivating example section are implemented in the GF grammar. There is not enough room for describing everything, so we give two representative examples: dynamically extending the grammar with domain specific concepts (Sect. 5.2), and formatting (Sect. 5.3). Throughout this section, we make use excerpts from the grammars without explaining all details of the GF formalism.

5.1 An Application Grammar for Specifications

Representing Specifications: Abstract Syntax. In a typical GF application grammar, the abstract syntax part is used for defining a semantic domain, without any linguistic considerations. We define categories (types) and functions which gives the rules for how to form trees in these categories. In our case, we are

[2] As explained in [2], this is because our grammar makes use of dependent types in such a way that the derived GF parser, which in a first step disregards dependent types, contained cyclic rules. In the meantime, this problem has been given a general solution in [20] (the implementation of which is in progress).

interested in the domain of (OCL) specifications. We consider OCL specifications as expressions formed using the attributes and queries of the classes in the UML model (including the predefined OCL types). For each class in the UML model, there will be a corresponding GF function c as well as a (dependent) category `Instance` c representing expressions of that class. As an example, this is how the `size` query of the OCL library class `String` (which returns the length of a string as an integer) is represented in GF abstract syntax judgements:

```
cat Class; cat Instance (c:Class); fun StringC, IntegerC: Class;
fun size : Instance StringC -> Instance IntegerC;
```

This defines the GF function `size` as taking trees of type `Instance StringC` and returning something of type `Instance IntegerC`. Note that with the dependent category `Instance`, we have introduced type-checking into the grammar: only trees representing type correct specifications can be built. As already mentioned, this leads to complications for the GF derived parser, see [2] for an explanation of this.

There are many choices to be made on exactly how to model specifications in abstract syntax, most of which we do not discuss here. One example, however, is introducing a category `Sent` for representing sentences. It is for instance used for the equality operator, with which we can state that any two instances x and y of the same class c are equal:

```
cat Sent;
fun equal : (c:Class) -> (x,y : Instance c) -> Sent;
```

The introduction of `Sent` is motivated by the fact that in natural language, we want to distinguish between expressions and sentences. In OCL, however, there is no such distinction – sentences just correspond to an expression of boolean type. This is an example of an interlingua problem: if a semantic distinction is made in one language, it has to be introduced into the abstract syntax, even if it is not present in the other languages.

Using the Resource Grammars: Concrete Syntax. In the concrete syntax, we give linearization rules presenting abstract syntax trees in English and German (we will not discuss the concrete syntax for OCL). To each category C in the abstract syntax, we must associate a record type: the *linearization category* of C. For each function in the abstract syntax, we define a linearization rule which builds a record in the corresponding linearization category. For instance, we might start with the abstract category `Class`, to be treated like common noun phrases in concrete syntax:[3]

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

[3] This example is simplified: in the real grammar, `Class` has a more complex linearization category which represents something more than just common noun phrases.

Here we define a parameter type for number. The linearization category of `Class` is a record type which has one field `s`, which is a string inflected in number. Then we define the linearization of `IntegerC` as the noun "integer" (in singular and plural form). To complete the concrete syntax, we would then have to go on defining `lincat`:s and `lin` rules for the rest of the abstract syntax types and functions, along with the required record and parameter types. However, we instead make use of the GF resource grammar library [17].

The resource grammars provide an API of linguistically motivated record and parameter types, along with utility functions to be used in linearization rules. For instance, there is a parameter type `Number`, as well as a record type `CN` for common noun phrases. Using the resource grammars raises the level of abstraction in the concrete syntax: instead of dealing directly with issues of e.g. inflection or word order, we can use the linguistic structures provided by the resource API. As long as we use only the API, all type-correct uses of the resource grammar preserve grammaticality. Since the API is available for seven languages including English and German, we can often reuse the same concrete syntax. Without explaining any details, an example of our use of the resource grammar is the following English linearization rule for the `size` method:

```
lin size x = DefOneNP (AppFun (funOfCN
                (useN (nNonhuman "length"))) x);
```

The functions used on the right hand side in the linearization rule are part of the resource grammar API. The linearization of the tree `size` x using this rule will be "the length of x". To provide the German linearization "die Länge von x", the rule is almost the same:

```
lin size x = DefOneNP (AppFun (funVonCN (useN (nFrau "Länge")))x);
```

5.2 Domain-Specific Vocabulary

In order to translate from OCL to English, the grammar needs to contain information about the UML model upon which the OCL is based. A grammar generation program therefore generates GF modules based on the UML model. The automatic generation does not always produce the most suitable translation, therefore it is also possible for the user to manually improve the translation by modifying the generated grammars, in particular the linearization rules in the concrete syntax.

To aid in the construction of the domain-specific concrete module, a resource module has been defined, which contains many operations that are useful when linearizing classes, attributes etc. We call this the API for Domain-specific Vocabulary. This API provides a layer of abstraction which hides some of the complexity of the rest of the grammar, making it easier to generate the linearizations for the domain entities. It also makes subsequent hand modifications possible without full knowledge about GF and the resource grammars.

Using the API. Although the API uses concepts taken from the OCL grammar and from the resource grammars, the interface provided is simple enough to not

require a deep knowledge of the underlying grammars. We will consider OCL classes as an example. The following operation is provided for constructing the linearization of a class (`ClassL` is the linearization category of `Class`):

```
oper mkClass: CN -> Str -> ClassL;
```

Classes are defined as consisting of a common noun phrase (`CN`) that corresponds to the class name as it will appear in natural text, and an identifier (a string), which is the actual name of the class in the UML diagram. The class identifier is used when formally specifying the class name.

Common ways of constructing the `CN` are included in the API, such as constructing a `CN` from a String, or adding an adjective to the `CN`. Irregular ways of constructing a `CN` can be found in the resource grammar modules. Take for example the class `OwnerPIN`, which is linearized using operations defined in the API.

```
lin OwnerPIN = mkClass (adjCN "owner" (strCN "PIN")) "OwnerPIN";
```

This will result in the class name being represented as a common noun phrase in natural text: "the maximum PIN size of the owner PIN is greater than 0" while the class identifier is used in a more formal setting: "for the class OwnerPIN the following invariants hold:".

Grammar Generation. The grammar generator uses some heuristics to derive a reasonable linearization for a domain entity (a class, an attribute, a method or an association) from its name and type. Given a UML model, it produces an abstract syntax module with one function for each domain entity, and a concrete module with corresponding English linearizations. A concrete module with OCL linearizations is also generated. The concrete English module makes use the API for domain-specific vocabulary, and the resource grammars. Since GF supports separate compilation of modules, only these generated modules need to be recompiled whenever the UML model changes, not the whole grammar.

The heuristics is based on types, and on splitting an identifier into words based on capitalization. E.g., the identifier `OwnerPIN` is split into the strings "owner" and "PIN". Since we also know the type, i.e. in this case that `OwnerPIN` is the name of class, we build a noun "owner PIN" as described just above. Another simple rule is special handling of boolean properties that start with "is", e.g. `isValidated` becomes a sentence saying "... is validated".

The heuristics for grammar generation obviously depends on the natural language used for identifiers, in this case English. A good heuristics for German would be more complex, e.g. it would require access to a lexicon for determining the gender of nouns. The heuristics also requires a consistent convention for word boundaries in identifiers (e.g. is it `isValidated` or `is_validated`?).

Modifying the Grammar. The generated grammar does not always succeed in producing the best translation. Using the API it is possible to make hand-modifications to the generated grammar without too much difficulty. For example, the attributes `triesLeft` and `maxTries` are translated as "the tries left" and "the max tries" by default using the generated judgements we see below.

```
lin maxTries = mkSimpleProperty (adjCN "max" (strCN "tries"));
lin triesLeft = mkSimpleProperty (adjCN "tries" (strCN "left"));
```

Although these translations are quite adequate, we can improve them by making manual changes to the generated grammar using some of the operations provided in the API. Thus, using the judgements below, we can construct the text "the triesLeft attribute" and "the maximum number of tries", for these attributes.

```
lin maxTries = mkSimpleProperty (ofCN (adjCN "maximum"
                    (strCN "number")) (strCN "tries"));
lin triesLeft = mkSimpleProperty (attrCN "triesLeft");
```

5.3 Grammar-Based Formatting

The use of formatting in the translated text has a dramatic effect on the readability of the output. As we see in the motivating example in Fig. 3, the formatting includes e.g. breaking the text into paragraphs, using different fonts for headings and argument variables, and presenting various structures in the form of bullet lists.

Most of this formatting is done completely on the level of concrete syntax: An interface module has been defined that contains operations required to perform formatting tasks, without specifying an implementation. This interface can then be implemented in many different ways using different instances. Currently three instances exist, allowing the possibility to have no formatting, HTML formatting or LaTeX formatting. These instances do not define their own *pretty-printing* rules, instead they simply use formatting tags leaving the actual layout to be handled by the LaTeX and HTML rendering engines. The linearization rules of the concrete syntax then makes use of the operations specified by the formatting interface.

Using Lists for Aggregation. There is one exception to the rule that all formatting is done just in concrete syntax: formatting lists of conjunctions and disjunctions as bullet lists. This requires changes also in the abstract syntax, as well as support from the external program which transforms the result of the context free OCL parser into GF abstract syntax.

To treat lists of conjunctions (the machinery for disjunctions is just the same) in a special way we simply introduce a new category `AndList` in the abstract syntax, along with functions for creating such lists, and converting them into sentences:

```
fun oneAnd : Sent -> Sent -> AndList;
fun consAnd : Sent -> AndList -> AndList;
fun andList2Sent : AndList -> Sent;
```

The base case `oneAnd` takes two sentences and builds an `AndList`, containing just one conjunction. The function `consAnd` prepends a sentence to an existing `AndList`. Once the list is built, `andList2Sent` allows us to consider it as a sentence.

A list containing just one conjunction, i.e. a tree `oneAnd` x y would be linearized just as "x and y", while using `consAnd` should result in a bullet list, saying "the following conditions are true: ...". However, this is dealt with in the concrete syntax (which is omitted here), the abstract syntax just provides the required structure. In fact, we use the same kind of abstract syntax for sums and products, where we are not interested in formatting. In that case, the problem is to translate e.g. an OCL expression `2+3` as "2 plus 3", but `1+2+3` as "the sum of 1, 2 and 3".

When translating OCL to English, we must find OCL expressions where lists of conjunctions occur, and make sure that they are treated as `AndLists`. This can be seen as simple aggregation problem [6]. As mentioned above, this step is not performed inside the grammars, but in the transformation from context free OCL syntax trees (as returned by the external parser) to GF abstract syntax trees.

6 Related Work

Natural Language Generation (NLG) is described in [6] as producing understandable natural language text from a non-linguistic representation of information. This very general description also fits GF linearization: Linearization can be considered as a two-step procedure, where the linearization rules go from non-linguistic abstract syntax to linguistically motivated resource grammar constructions. The resource grammar implementation then takes the step to surface strings in natural language. However, while linearization is therefore clearly more than just linguistic realization (cf. the discussion in [6] on realization as the inverse of parsing), it is much simpler than a typical NLG system. Linearization rules (and the resource grammars) are written in GF concrete syntax, a restricted functional language, and linearization rules are always compositional. In contrast, [6] describes a typical NLG system architecture as a pipeline consisting of the phases text planning, sentence planning and linguistic realization, along with separate intermediate representation formats.

When using our system for translating OCL to English (as opposed to syntax editing), there is also the external OCL parser / typechecker and grammar generation, and the architecture is more similar to that of a compiler than a NLG system. We also do some transformations to the GF syntax trees in an external program. Some of these transformations could be described in terms of NLG concepts, e.g. aggregation (Sect. 5.3). They are also similar to some of the ideas in [21], which describes generation of natural language text from formal proofs as a process resembling code generation in a compiler.

7 The Case Study in Numbers

The Java Card API case study consists of OCL specifications of 37 classes, the word count of the English translation is close to 17000. The generated grammar modules of domain-specific vocabulary contain about 1100 concepts, i.e. 1100 abstract syntax functions, each one with a corresponding linearization rule. 361 of

these concepts are actually being used in the translated specifications. By-hand modifications were made to the linearization rules of 73 of these 361 concepts, i.e. 20% of the used domain-specific concepts needed modifications. 18 of these modifications are of a trivial nature and could probably be automated if one introduces domain-specific heuristics for grammar generation, which leaves 15% of the used domain-specific concepts that require non-trivial modifications.

8 Conclusion

We have presented a tool for translating formal OCL specifications into natural language based on GF grammars. By adding a domain-specific vocabulary API, formatting, and other stylistic improvements, we achieve a translation of a non-trivial case study of OCL specifications which is acceptable to a human reader. Relatively few by-hand modifications using the API were necessary; the modifications are made on the grammar level, but do not require linguistic or GF expertise. Although we add external programs, the compositional and declarative GF formalism remains the centre of our work: the external programs are used either to generate GF grammar modules, or to manipulate GF abstract syntax trees. The tool and the Java Card API case study are available on the web [16].

8.1 Future Work

Important lines of future work include: (1) Further improvements to the natural language, e.g. by more sophisticated use of aggregation and referring expressions generation. (2) A more formal evaluation of the quality of the generated natural language. (3) Integration into the KeY system, most importantly providing a user interface for manipulating domain-specific vocabulary, based on the API we have defined.

Acknowledgements

We thank Reiner Hähnle, Aarne Ranta and the anonymous referees for valuable suggestions on how to improve the paper.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling **4** (2005) 32–54
2. Hähnle, R., Johannisson, K., Ranta, A.: An authoring tool for informal and formal requirements specifications. In Kutsche, R.D., Weber, H., eds.: Fundamental Approaches to Software Engineering. Number 2306 in LNCS (2002)
3. Ranta, A.: Grammatical Framework: A Type-theoretical Grammar Formalism. The Journal of Functional Programming **14** (2004) 145–189

4. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis, Napoli (1984)
5. Sun Microsystems: Java card homepage (2004) http://java.sun.com/products/javacard/.
6. Reiter, E., Dale, R.: Building applied natural language generation systems. Journal of Natural Language Engineering **3** (1997) 57–87
7. Ranta, A.: Grammatical Framework homepage (2005) www.cs.chalmers.se/~aarne/GF.
8. Khegai, J., Nordström, B., Ranta, A.: Multilingual syntax editing in GF. In Gelbukh, A., ed.: CICLing-2003, Mexico City, Mexico. LNCS, Springer (2003)
9. Ranta, A.: The GF resource grammar library (2004) http://www.cs.chalmers.se/~aarne/GF/lib/resource/.
10. Sommerville, I.: Software Engineering. Seventh edn. Addison Wesley (2004)
11. The Object Management Group: Object constraint language specification (2004) http://www.omg.org/docs/formal/03-03-13.pdf.
12. The Object Management Group: Unified modelling language homepage (2004) http://www.uml.org.
13. Larsson, D., Mostowski, W.: Specifying Java Card API in OCL. In Schmitt, P.H., ed.: OCL 2.0 Workshop at UML 2003. Volume 102C of ENTCS., Elsevier (2004) 3–19
14. Meijer, H., Poll, E.: Towards a full formal specification of the Java Card API. In Attali, I., Jensen, T., eds.: Smart Card Programming and Security. Number 2140 in LNCS, Springer (2001) 165–178
15. Burke, D.A.: Improving the natural language translation of formal software specifications. Master's thesis, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2004)
16. Johannisson, K.: OCL to natural language tool homepage (2004) http://www.cs.chalmers.se/~krijo/gfspec/.
17. Daniels, H.J.: Eine deutsche Grammatik für OCL. Studienarbeit (2003) http://www.cs.chalmers.se/~krijo/gfspec/.
18. Hallgren, T., Ranta, A.: An extensible proof text editor. In Parigot, M., Voronkov, A., eds.: Logic for Programming and Automated Reasoning, LPAR. LNAI 1955, Springer (2000) 70–84
19. Johannisson, K.: Disambiguating implicit constructions in OCL (2004) Online proceedings of OCL and Model Driven Engineering Workshop at UML 2004, http://www.cs.kent.ac.uk/projects/ocl/oclmdewsuml04/description.htm.
20. Ljunglöf, P.: Expressivity and complexity of the Grammatical Framework. PhD thesis, Chalmers University of Technology, Göteborg University, SE-412 96 Göteborg, Sweden (2004)
21. Coscoy, Y., Kahn, G., Thery, L.: Extracting text from proofs. In Dezani-Ciancaglini, M., Plotkin, G., eds.: Proc. Second Int. Conf. on Typed Lambda Calculi and Applications. Volume 902 of LNCS. (1995) 109–123