An Authoring Tool for Informal and Formal Requirements Specifications

Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta

Chalmers University of Technology, Department of Computing Science
S-41296 Gothenburg, Sweden, {reiner,krijo,aarne}@cs.chalmers.se

Abstract We describe foundations and design principles of a tool that supports authoring of informal and formal software requirements specifications simultaneously and from a single source. The tool is an attempt to bridge the gap between completely informal requirements specifications (as found in practice) and formal ones (as needed in formal methods). The user is supported by an interactive syntax-directed editor, parsers and linearizers. As a formal specification language we realize the Object Constraint Language, a substandard of the UML, on the informal side a fragment of English. The implementation is based on the Grammatical Framework, a generic tool that combines linguistic and logical methods.

1 Introduction

The usage of formal and semi-formal languages for requirements specifications is becoming more widespread. Witness, for example, the Java Modeling Language (JML) [11], closely related to which is the ESC/Java specification language used in Extended Static Checking [12], the constraint language Alloy [9], and the Object Constraint Language (OCL) [15,21]. The OCL is not only used in meta-modeling to supply a precise semantics for UML diagrams, but also in requirements specification. A subset of the OCL is also used in iContract [10], the JAVA variant of design-by-contract, as an assertion language.

Although these languages make an effort to be more "user-friendly" than earlier formal notations that were based on set theory and predicate logic, it still takes a considerable effort to master them and use them effectively. Moreover, it should not be forgotten that the by far most popular language, wherein software specifications are still written today is natural language (NL).

None of the approaches mentioned above offers support for authoring, understanding, and maintaining formal specifications. We consider this deficiency to be a serious obstacle to routine usage and further development of formal and semi-formal methods. Specifically, the following problems have to be addressed, if formal and semi-formal notations are to become a standard item in the software engineer's toolbox:

Authoring. Support is needed for authoring well-written, well-formed formal specifications. A syntax-directed editor is of help along with specification templates.

Maintenance. Large and complex expressions in any formal language are not easy to read, even if, like OCL, this language was designed to enhance readability. In realistic scenarios, numerous and complex expressions have to be maintained and, therefore, understood by people who did not necessarily author them or are even familiar with formal languanges.

Mapping Different Levels of Formality. No specification language fits all needs. For different audiences and purposes it is important to have renderings in, say, NL, OCL, and first-order logic. For effective communication parts of these must be mappable into each other efficiently and with a clear semantics.

Synchronisation. If a system is specified in languages of differing level of precision, it is important to propagate changes consistently. For example, any change in an OCL constraint should be instantly reflected in the corresponding NL description. It will not do to perform these changes manually.

In this paper we suggest a solution to the problems just outlined. We show that a systematic connection between specification languages on differing levels of precision is possible. We concentrate on OCL and NL as specification languages, but the method is not limited to this configuration.

Our approach is based on the Grammatical Framework (GF) [18], a flexible mechanism that allows to combine linguistic and logical methods. The key idea is to specify (i) an abstract syntax for a specification language (in our case roughly corresponding to OCL) together with semantic conditions of well-formedness and type-correctness, and (ii) concrete syntaxes for all supported notations (in our case, concrete OCL expressions as well as a fragment of English). For each set of abstract/concrete syntaxes the GF system then implements algorithms for parsing, linearization, translation, type checking, and a graphical syntax editor. The abstract grammar is much richer than the usual context-free OCL grammar [15] and, together with the syntax editor, enables interactive editing of templates for frequently needed specifications. The result is an authoring system for requirements specifications that supports creation and maintenance of informal and formal specifications from a single source.

In Section 2 we walk through an example that serves as motivation and at the same time demonstrates what can be done with our system. In Section 3 we give some background on the GF formalism that is necessary to understand Section 4, where the implementation is discussed in detail. In Section 5 we evaluate our approach and we show how the problems outlined above are addressed in our system. The paper is rounded off with brief sections on related work, on future work, and by concluding remarks.

The latest prototype of our system can be downloaded from `http://www.cs.chalmers.se/~krijo/GF/specifications.html`.

2 Motivating Example

As a motivating example we will consider a standard queue data structure—a class `Queue`—and show how to use our system for developing specifications of this class in OCL and natural language.

2.1 A Class for Queues

For the purpose of this example, we need to specify the interface of a class `Queue` for queues of integers (but we need not consider any implementation details). We use the standard OCL types in doing this:

Queue

`enqueue(i: Integer): Integer`
`dequeue(): Integer`
`getFirst(): Integer {query}`
`size(): Integer {query}`
`asSequence(): Sequence(Integer) {query}`

This class should be very straightforward. We have an operation `enqueue` for enqueueing an integer on the queue and an operation `dequeue` for removing the first integer of the queue. The return value of `enqueue` is simply the value of its argument. We also have an operation `getFirst` for inspecting the first element of the queue. The operation `asSequence` gives us a `Sequence` (standard OCL type) with all the elements from the queue, in their correct order. This operation is included for specification purposes; in an actual implementation of the class, `asSequence` is not required.

Note also that all operations which do not affect the state of the queue ("observer methods" or "queries") have been tagged with `{query}`, using standard UML notation.

2.2 Using the GF-based System

Our system is based on the GF system (described in Section 3) with grammars for OCL and English (Section 4). It features an interactive editor for formulating constraints in OCL and English:

Suppose that we want to author a postcondition for the method `enqueue` of the class `Queue` in the interactive editor. Figure 1 shows what the editor looks like after a few initial steps.

In this screen-shot we see the beginning of a postcondition for an operation. The main part of the window shows the postcondition in OCL ① and in English ②, and also an abstract (internal to GF) representation ③. To complete this postcondition, we select a subgoal (that is, metavariable or placeholder) of the form `[?...?]` and then select one of the possible refinements in the lower left subwindow ④, until there are no more subgoals to fill in. In the example, the next logical step is to specify the operation for which the current postcondition is intended, that is, `enqueue`. So we select the subgoal `[?Operation?]` and the refinement `enqueue`. Figure 2 shows the result.

Now a new subgoal `[?BindPPCond?]` is active, and new refinements appear in the lower left menu. The subgoal `[?Class?]` was automatically filled in with `Queue`, since this was the only correct refinement left after we chose the operation `enqueue`. The system is able to infer this automatically.

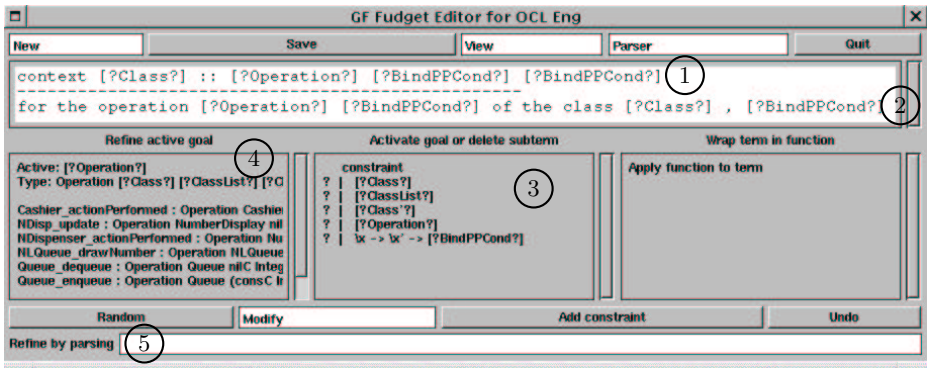

Figure 1. The Interactive Editor

Note that we edit the postcondition in OCL and in English in parallel. Every change is instantly reflected in both the OCL and the English version. What is actually going on is that we are editing the abstract representation, which is linearized to English and OCL. This means that the user of the editor can produce OCL constraints, even though he or she only understands the English form of the constraint.

There are also other ways to interact with the editor. Aside from choosing refinements from a menu to fill in a subgoal, we can simply enter a string (at ⑤ in Figure 1) in English or OCL which will be parsed by the GF editor. We can also wrap a term in a function, that is, perform bottom-up editing instead of top-down.

As will be seen in Section 3, the interactive editor is merely one part of GF: having grammars for OCL and English means that we also have a parser for OCL and for a fragment of English as well as a translator between OCL and this fragment of English.

2.3 More Examples

We present some more constraints for methods in the `Queue` class authored with our system and highlight some of the problems that had to be solved in order to obtain a smooth rendering in English. In the OCL versions of the constraints only line breaks and spaces were inserted by hand (this is a current limitation). The formatting of the English version of the constraints was achieved by including LaTeX commands in the English grammar.

Operation `getFirst`

OCL:
```
context Queue::getFirst() : Integer
    pre:  self.size() > 0
    post: result = self.asSequence()->first
```
English: for the operation getFirst() : Integer of the class Queue, the following precondition should hold:

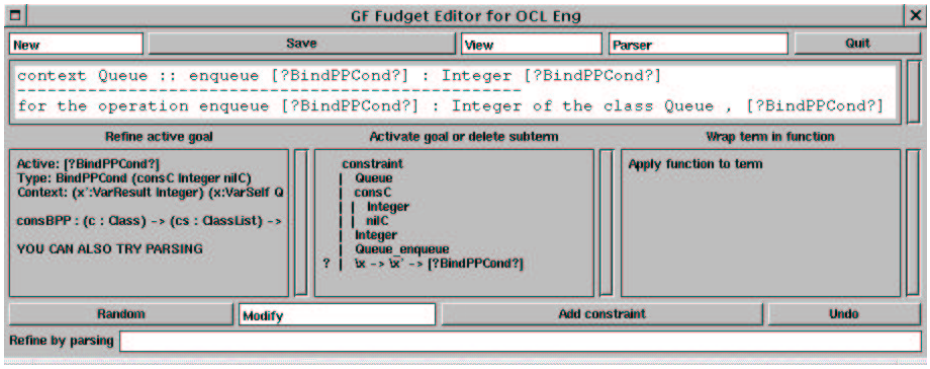

Figure 2. The Interactive Editor—one editing step later

the size of the queue is greater than zero
and the following postcondition should hold:
the result is equal to the first element of the queue

The meaning of the OCL constraint `self` depends on the context. In these examples, `self` refers to an instance of `Queue`, since we are formulating constraints for an operation of the class `Queue`. In English, `self` corresponds to an anaphoric expression, which in this particular case is "the queue".

The operation `asSequence` can be seen as a way of converting a `Queue` to an OCL `Sequence`. While this type cast is necessary in OCL, it is not that interesting in English. It is therefore omitted, so the OCL expression `self.asSequence()` simply corresponds to "the queue" in English.

Operation `dequeue`

OCL:
```
context Queue::dequeue() : Integer
    pre: self.size() > 0
    post: (self.size() > 0 implies self.asSequence() =
        self.asSequence@pre() -> subSequence(2, self.size() + 1))
      and result = self.getFirst@pre()
```
English: for the operation dequeue() : Integer of the class Queue, the following precondition should hold:
the size of the queue is greater than zero
and the following postconditions should hold:
- if the size of the queue is greater than zero, then the queue is equal to the subsequence of the queue at the start of the operation which starts at index 2 and ends at the index equal to the size of the queue plus one
- the result is equal to the first element of the queue at the start of the operation

Here we see that a sequence of conjuncts in OCL (such as `x and y and ...`) can be displayed as an itemized list in English. This implies that we need to have the word "postconditions" in plural form (in contrast to the `getFirst` example,

where we have "postcondition"). We can also note that `@pre` in OCL simply corresponds to "at the start of the operation" in English.

The OCL operation `subSequence` requires its second argument to be greater than or equal to its first argument – it never returns an empty sequence. This explains why we use the condition that the size of the queue is greater than zero in the first postcondition.

3 Grammatical Framework

The Grammatical Framework (GF) is a framework for defining grammars and working with them [18]. It is used for defining special-purpose grammars on top of a semantic model, which is expressed in type theory [13]. Type theory is a part of GF, the abstract syntax part. The concrete syntax part tells how type-theoretical formulas are translated into a natural language or a formal notation.

The first application of GF was in a project on Multilingual Document Authoring at Xerox Research Centre Europe [4]. The idea in multilingual authoring is to build an editor whose user can write a document in a language she does not know (for example, French), while at the same time seeing how it develops in a language she knows (for example, English). From the system's point of view, the object constructed by the user is a type-theoretical formula, of which the French and English texts are just alternative views.

The GF programming language, as well as the tools supporting multilingual authoring, are designed to be generic over both the subject matter and the target language. While prototypes have been built for documents such as tourist information and business letters, the most substantial application so far has been natural-language rendering of formalized proofs [5]. Software specifications are another natural GF application, since it is usually clear how specifications are expressed in type theory. Most uses of type theory as a specification language have been based on the Curry-Howard isomorphism, but we will here use it for OCL specifications.

3.1 Abstract Syntax

GF, like other logical frameworks in the LF [6] tradition, uses a higher-order type theory with dependent types. In this type theory, it is possible to define logical calculi, as well as mathematical theories, simply by type signatures. The type-theoretical part of a GF grammar is called the abstract syntax of a language.

To take an example, we first define the types of propositions and proofs, where the type of proofs depends on proposition.

```
cat Prop ; Proof Prop ;
```

We then define implication as a two-place function on propositions, and the implication introduction rule is a function whose argument is a function from proofs of the antecedent to proofs of the succedent:

```
fun Imp  : Prop -> Prop -> Prop ;
fun ImpI : (A,B:Prop) -> (Proof A -> Proof B) -> Proof (Imp A B)
```

As usual in functional languages, GF expresses function application by juxtaposition, as in `Proof A`, and uses parentheses only for grouping purposes.

3.2 Concrete Syntax

On top of an abstract syntax, a concrete syntax can be built, as a set of linearization rules that translate type-theoretical terms into strings of some language. For instance, English linearization rules for the two functions above could be

```
lin Imp A B = {s = "if" ++ A.s ++ "then" ++ B.s} ;
lin ImpI A B c = {s = "assume" ++ A.s ++ "." ++ c.s ++ "." ++
                      "Hence" ++ "if" ++ A.s ++ "then" ++ B.s} ;
```

As shown by these examples, linearization is not just a string, but a record of concrete-syntax objects, such as strings and parameters (genders, modes, etc.), and parameter-dependent strings. Notice that linearization rules can generate not only sentences and their parts, but also texts. For instance, a proof of the implication $A\&B \to A$ as generated by the rules above, together with a rule for conjunction elimination, is a term that linearizes to the text:

> Assume A and B. By the assumption, A and B. *A fortiori*, A. Hence if A and B then A.

Different languages generally have different types of concrete-syntax objects. For instance, in French a proposition depends on the parameter of mode, which we express by introducing a parameter type of modes and defining the linearization type of `Prop` accordingly:

```
param  Mode = Ind | Subj ;
lincat Prop = {s : Mode => Str} ;
```

The French linearization rule for the implication is

```
lin Imp A B =
     {s = table {m => si (A.s ! ind) ++ "alors" ++ B.s ! m}} ;
```

which tells that the antecedent is always in the indicative mode and that the main mode of the sentence is received by the succedent. One may also notice that *si* is not a constant string, but depends (in a way defined elsewhere in the grammar) on the word following it (as in *s'il vous plaît*).

Finally, in formal logical notation, linearization depends on a precedence parameter:

```
lincat Prop = {s : Prec => Str} ;
lin Imp A B = {s = mkPrec p0 (A.s ! p1 ++ "->" ++ A.s ! p0)} ;
```

where the function `mkPrec` (defined elsewhere in the concrete syntax) controls the usage of parentheses around formulas.

The examples above illustrate what is needed to achieve genericity in GF. In the abstract syntax, we need a powerful type theory in order to express dependencies among parts of texts, such as in inference rules. In the concrete syntax, we need to define language-dependent parameter systems and complex structures of grammatical objects using them.

3.3 Functionalities

GF helps the programmer of grammar applications by providing framework-level functionalities that apply to any GF grammar. The main functionalities are **linearization** (translation from abstract to concrete syntax), **parsing** (translation from concrete to abstract syntax), **type-checking** of abstract-syntax objects, **syntax editing**, and **user interfaces** (both line-based and graphical). Although these functionalities apply generically to all grammars, it is often useful to customize them for the task at hand. For this end, the GF source code (written in Haskell) provides an API for easy access to the main functionalities.

4 Implementation

4.1 Classes and Objects

In this section we give a general idea of how we have implemented GF grammars for OCL and natural language (at present English). We begin by defining categories and functions for handling standard object-oriented concepts such as classes, objects, attributes and operations:

```
cat Class;
cat Instance (c : Class);
```

There is a category (type) `Class` of classes, and a dependent type `Instance`, hence, for every class `c` there is a type `Instance c` of the instances of this class. OCL expressions are represented as instances of classes (and we can of course see an instance of a class as an object).

Classes are introduced by judgements like the following:

```
fun Bool : Class; Integer : Class; Real : Class;
```

This means that we have type checking in the abstract grammar: for example, where a term of type `Instance Bool` is expected, we cannot use a term of type `Instance Integer`.

The linearizations of these functions to OCL is easy: for `Integer` we simply take `lin Integer = {s = "Integer"}`, and so on. In English, a class can be linearized either as a noun (which can be in singular or plural form, say, "integer" or "integers") or as an identifier of a class ("Integer"). In GF we handle this by using parameters, as explained in Section 3.

Subtyping (inheritance) between classes is handled by defining a subtype relation and a coercion function:

```
cat Subtype (sub, super : Class);
fun coerce : (sub,super:Class) -> Subtype sub super ->
      Instance sub -> Instance super;
```

The function `coerce` is used for converting an instance of a class `c` into an instance of any superclass of `c`. The arguments to this function are the classes in question, a proof that the subtyping relation holds between the classes, and finally an instance of the subclass. The result is an instance of the superclass. For every pair of classes in OCL's subtyping relation we introduce a term (a proof) of the type `Subtype`, e.g.:

```
fun intConformsToReal : Subtype Integer Real;
```

The linearization of `coerce` is interesting: since the whole point is to change the type (but not the meaning) of a term, the linearization rule will leave everything as it is. For both OCL and English we have:

```
lin coerce _ _ _ obj = obj;
```

GF converts to context free grammars to realize parsing, and this makes this rule circular (it has the form `Instance -> Instance`). This means that we cannot use our grammars to parse OCL or English with the GF system as it is now. We will have to implement custom modifications for coercion rules.

4.2 Attributes, Operations and Queries

For operations and attributes we have three categories:

```
cat Attribute (c,a : Class);
    Operation (c:Class) (args:ClassList) (returns:Class);
    OperationQ (c:Class) (args:ClassList) (returns:Class);
```

Attributes are simple enough: the two arguments to `Attribute` give the class to which the attribute belongs, and the type (class) of the attribute itself, respectively. For operations, we need to know if they have side-effects, i.e. whether they are marked with `{query}` in the underlying UML model or not. This explains why there are two categories for operations. The first argument of these categories is, again, the class to which they belong. The second argument is a (possibly empty) list of the types of the arguments to the operation, the third argument is the return type (possibly void) of the operation. The use of lists makes these categories general (they can handle operations with any number of arguments), but this generality also makes the grammar a bit more complex at places.

Here is how we use an UML attribute or query method (a term of type `OperationQ`) within an OCL expression:

```
fun valueOf : (c, result:Class) -> (Instance c) ->
      Attribute c result -> Instance result;
    query : (c:Class) -> (args:ClassList) -> (ret:Class) ->
      Instance c -> OperationQ c args ret  -> InstList args ->
      Instance ret;
```

The arguments to `query` are, in turn: the class of the object we want to query, a list of the classes of the arguments to the query, the return type of the query, the object we want to query, the query itself, and finally a list of the arguments of the query. The result is an instance (an object) of the return type.

The linearization to OCL is fairly simple:

```
lin query _ _ ret obj op argsI =
      dot1 obj (mkConstI (op.s ++ argsI.s ! brackets));
```

What happens here is that the list of arguments is linearized as a comma-separated list enclosed in parentheses (`argsI.s ! brackets`), then we put the name of the query (`op.s`) in front, and finally add the object we query and a dot (`dot1` ensures correct handling of precedence), so we end up with something like `obj.query(arg1, arg2, ...)`.

For the English linearization, we have the problem of having one category for all queries, regardless of the number of arguments they depend on. Our solution is to give a custom "natural" linearization of queries having up to three arguments (this applies to all query operations in `Queue`). For instance, the linearization of the query `getFirst` produces "the first element of the queue". For `asSequence` we take, as could be observed in Section 2, simply "the queue". The implementation is based on the following:

```
param Prep = To | At | Of | NoPrep;
lincat OperationQ = {s : QueryForm => Str;
         preps : {pr1 : Prep; pr2 : Prep; pr3 : Prep}};
```

The idea is that the linearization of a query includes up to three prepositions which can be put between the first three arguments. If there are more than three arguments, these prepositions are ignored and we choose a more formal notation like "query(arg1, arg2, ...) of the queue".

4.3 Constraints

For handling OCL constraints (invariants, pre- and postconditions) we introduce a category `Constraint` and various ways of constructing terms of this type. The simplest form of constraint is an invariant for a class:

```
cat Constraint;
fun invariant : (c:Class) -> (VarSelf c -> Instance Bool) ->
      Constraint;
```

To construct an invariant we supply the class for which the invariant should hold: the first argument of `invariant`. We require a boolean expression (a term of type `Instance Bool`) which represents the actual invariant property. An additional complication is that we want to be able to refer to (the current instance of) the class `c` in this boolean expression—in terms of OCL this means to use the variable `self`. This accounts for the type of the second argument, `VarSelf c -> Instance Bool`, which can be thought of as a term of type

`Instance Bool` where we have access to a bound variable of type `VarSelf c`. This bound variable can only be used for one purpose: to form an expression `self` of the correct type:

```
fun self : (c:Class) -> VarSelf c -> Instance c;
```

The linearization of `invariant` is simple, and we show the linearizations for both OCL and English:

```
lin invariant c e = {s = "context"++c.s++"inv:"++e.s};
lin invariant c e = {s = ["the following invariant holds
      for all"] ++ (c.s ! cn pl) ++ ":" ++ e.s} ;
```

Notice the choice of the plural form of a class: `c.s ! cn pl` produces, for example, "queues", for `Queue`.

For formulating pre- and postconditions of an operation, we use the same technique employing bound variables. In this case one bound variable for each argument of the operation is required, besides the ones for `self` and `result`.

4.4 The OCL Library and User Defined Classes

The grammar has to include all standard types (and their properties) of OCL. Just as an example, we show the `Sequence` type and some of its properties:

```
fun Sequence : Class -> Class;
    subSequence : (c:Class) -> Instance (Sequence c) ->
      (a,b : Instance Integer) -> Instance (Sequence c);
    seqConforms2coll : (c:Class) ->
      Subtype (Sequence c) (Collection c);
```

The operations of `Sequence` (or any standard OCL type) are not terms of type `OperationQ`, they are simply modelled as functions in GF. This is very convenient, but it also means that the grammar does not allow to express constraints for the standard OCL operations. User defined operations, however, must permit constraints, so they are defined using `Operation` and `OperationQ`. Here are some operations of the class `Queue` from Section 2:

```
fun Queue : Class;
    Queue_size : OperationQ Queue nilC Integer;
    Queue_enqueue : Operation Queue (consC Integer nilC) Integer;
```

Note the use of the constructors `nilC` and `consC` to build lists of the types of the arguments to the operations.

5 Evaluation

5.1 Advantages

Our approach to building an authoring tool has a number of advantages for the development of requirements specifications:

Single Source Technology. Each element of a specification is kept only in one version: the annotated syntax tree of the abstract grammar. Concrete expressions are generated from this on demand. In addition, edits made in one concrete representation, are reflected instantly in all others. This provides a solution to the maintenance and synchronization problems discussed in Section 1. The following two items address the mapping problem:

Semantics. The rules of abstract and concrete GF grammars can be seen as a formal semantics for the languages they implement: for each pair of concrete languages, they induce a function that gives to each expression its "meaning" in the other language. Working with the syntax directed editor, which displays abstract and concrete expressions simultaneously, makes it easy for users to develop an intuition for expressing requirements in different specification languages.

Extensibility. GF grammars constitute a declarative and fairly modular formalism to describe languages and the relationships among them. This makes it relatively easy to adapt and extend our implementation.

These positive features rest mainly on the design principles of GF. From an implementor's point of view, the GF base provides a number of additional advantages. The fact that GF is designed as a *framework* is crucial:

Tools. GF provides a number of functionalities for each set of abstract and concrete grammars as detailed in Section 3.3 and an interactive syntax directed editor coming with a GUI. In particular, we have a parser for full OCL incorporating extensive semantic checks.

Development Style. The declarative way, in which knowledge about specific grammars is stored in GF, permits a modern, incremental style of development: rapid design-implementation-test cycles, addition of new features on demand, and availability of a working prototype almost from the start, are a big asset.

5.2 Limitations

GF gives a number of functionalities for free, so that applications can be built simply by writing GF grammars. The result, however, is not always what one would expect from a production-quality system. Software built with GF is more like a prototype that needs to be optimized (more accurately: the grammars can be retained, but the framework-level algorithms must be extended). In the case of specification authoring, we encountered the following limitations:

Parsing. The generic GF parsers are not optimized for parsing formal languages like OCL, for which more efficient algorithms exist. More seriously, the parser has to be customized to avoid the circularity problem due to instance coercions (Section 4.1).

Compositionality. Texts generated by GF have necessarily the same structure as the corresponding code. One would like to have methods to rephrase and summarize specifications.

The need for grammars. All new, user-defined concepts (classes and their features) have to be defined in a GF grammar. It would be better to have the

possibility to create grammars dynamically from UML class diagrams given in some suitable format (for example, in the UML standard exchange format XMI [20]); this can be done in the same way as GF rules are generated from Alfa declarations [5].

A general limitation, which is a problem for *any* natural-language interface, is:

Closedness. Only those expressions that are defined in the grammar are recognized by the parser.

This means a gap persists between formal specifications and informal legacy specifications. One could imagine heuristic natural language processing methods to rescue some of this material, but GF does not have such methods at present.

Finally, an obstacle to the applicability of syntax-directed editors for programming languages, for which special techniques are required [19], is the phenomenon that top-down development as enforced by stepwise refinement is usually incompatible with the direction of the control flow. The latter, however, is more natural from an implementor's point of view. This problem does not arise in the context of specifications due to their declarative nature.

6 Related Work

We know of no natural-language interfaces to OCL, but there are some earlier efforts for specifications more generally: Holt and Klein [7] have a system for translating English hardware specifications to the temporal logic CTL; Coscoy, Kahn and Théry [3] have a translator from Coq code (which can express programs, specifications, and proofs) into English and French. Both of these systems function in batch mode, and they are unidirectional, whereas GF is interactive and bidirectional. Power and Scott [16] have an interactive editor for multilingual software manuals, which functions much like GF (by linearizations from an underlying abstract representation), but does not have a parser. In all these systems, the grammar is coded directly in the program, whereas GF has a separate grammar formalism. The mentioned systems are fine-tuned for the purposes that they are used for, and hence produce or recognize more elegant and idiomatic language. But they cannot be dynamically extended by new forms of expression. An idea from [3] that would fit nicely to GF is optimization by factorization: For example, *x is even or odd* is an optimization of *x is even or x is odd*.

The context free grammar of OCL 1.4 [15] is a concrete grammar, which is not suitable as a basis for an abstract grammar for both OCL and English. Furthermore, it provides no notion of type correctness. A proposal for OCL 2.0 [14] addresses these problems: both an abstract and a concrete grammar are included, as well as a mechanism for type correctness. However, these grammars are partly specified by metamodelling, in the sense that UML and OCL themselves are used in the formal description of syntax and semantics. It is, therefore, not obvious how to construct a GF grammar directly based on the OCL 2.0 proposal.

A general architecture for UML/OCL toolsets including a parser and type checker is suggested in [8], but informal specifications are not discussed there.

7 Future Work

Besides overcoming the limitations expressed in Section 5.2, we will concentrate on the following issues:

Integration. For our authoring tool to be practically useful, it must be tightly integrated with mainstream software development tools. In the KeY project [1] a design methodology plus CASE-tool is developed that allows seamless integration of object-oriented modeling (OOM) with program development, generation of formal specifications, as well as formal verification of code and specifications. The KeY development system is based on a commercial UML-CASE tool for OOM. Following, for example, [8] we will integrate our tool into KeY and, hence, into the CASE tool underlying KeY. Users of the CASE tool will be able to use our authoring tool regardless of whether they want to do formal reasoning.

Stylistic Improvements. To improve the style of texts, we plan to use techniques like factorization [3] and pronominalization [17], which can be justified by type-checked definitions inside GF grammars. To some extent, such improvements can be even automatized. However, one should not underestimate the difficulty of this problem: it is essentially the same problem as taking a piece of low-level code and restructuring it into high-level code.

More and Larger Case Studies. We started to author a combined natural language/OCL requirements specification of the API of the Java Collections Framework based on textual specifications found there.

Further Languages. It is well-known how to develop concrete grammars for other natural languages than English. Support for further formal specification langages besides OCL might require changes in the abstract grammar or could even imply to shift information from the abstract to the concrete grammars. It will be interesting to see how one can accomodate languages such as Alloy or JML.

Improve Usability. The usability of the current tool can be improved in various ways: the first are obvious improvements of the GUI such as context sensitive pop-up menus, powerful pretty-printing, active expression highlighting, context-sensitive help, etc. A conceptually more sophisticated idea is to enrich the abstract grammar with rules that provide further templates for frequently required kinds of constraints. For example, a non-terminal `memberDeleted` could guide the user in writing a proper postcondition specifying that a member was deleted from a collection object. This amounts to encoding *pragmatics* into the grammar.

Increase Portability. The GUI of GF's syntax editor is written with the Haskell Fudgets library [2]. We plan to port it to Java. This is compatible with the KeY system, which is written entirely in Java.

8 Conclusion

We described theoretical foundations, design principles, and implementation of a tool that supports authoring of informal and formal software requirements

specifications. Our research is motivated by the gap between completely informal specifications and formal ones, while usage of the latter is becoming more widespread. Our tool supports development of formal specifications in OCL: it features (i) a syntax-directed editor with (ii) templates for frequently needed elements of specifications and (iii) a single source for formal/informal documents; in addition, (iv) parsers, (v) linearizers for OCL and a fragment of English, and (vi) a translator between them are obtained.

The implementation is based on a logico-linguistic framework anchored in type theory. This yields a formal semantics, separation of concrete and abstract syntax, separation of declarative knowledge and algorithms. It makes the system easy to extend and to modify.

In summary, we think that our approach is a good basis to meet the challenges in creating formal specifications outlined in the introduction.

Acknowledgements

We would like to thank Wojciech Mostowski and Bengt Nordström for the careful reading of a draft of this paper, for pointing out inaccuracies, and for suggestions to improve the paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. JELIA*, LNAI 1919, pages 21–36. Springer, 2000.
2. M. Carlsson and T. Hallgren. *Fudgets—Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 1998.
3. Y. Coscoy, G. Kahn, and L. Thery. Extracting text from proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Second Int. Conf. on Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 109–123, 1995.
4. M. Dymetman, V. Lux, and A. Ranta. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249, 2000.
5. T. Hallgren and A. Ranta. An extensible proof text editor. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning, LPAR*, LNAI 1955, pages 70–84. Springer, 2000.
6. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, 1993.
7. A. Holt and E. Klein. A semantically-derived subset of English for hardware verification. In *Proc. Ann. Meeting Ass. for Comp. Ling.*, pages 451–456, 1999.
8. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *Proc. 3rd Int. Conf. on the Unified Modeling Language*, LNCS 1939, pages 278–293. Springer, 2000.
9. D. Jackson. Alloy: A lightweight object modelling notation. `sdg.lcs.mit.edu/~dnj/pubs/alloy-journal.pdf`, July 2000.
10. R. Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of OO Languages and Systems, TOOLS 26*. IEEE CS Press, Los Alamitos, 1998.

11. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State Univ., Dept. of Computer Science, Feb. 2000.
12. K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Note #2000-002, Compaq Systems Research Center, Palo Alto, USA, May 2000.
13. B. Nordström, K. Petersson, and J. M. Smith. Martin-löf's type theory. In S. Abramasky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5. Oxford University Press, 2000.
14. Object Modeling Group. *Response to the UML 2.0 OCL RfP*, Aug. 2001. `cgi.omg.org/cgi-bin/doc?ad/01-08-01`.
15. Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, Sept. 2001. `www.omg.org/cgi-bin/doc?formal/01-09-67`.
16. R. Power and D. Scott. Multilingual authoring using feedback texts. In *COLING-ACL 98*, Montreal, Canada, 1998.
17. A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.
18. A. Ranta. Grammatical framework homepage, 2000. `www.cs.chalmers.se/~aarne/GF/index.html`.
19. T. Teitelbaum and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. *CACM*, 24(9):563–573, 1981.
20. Unisys Corp. et al. *XML Metadata Interchange (XMI)*, Oct. 1998. `ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf`.
21. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley, 1999.