Towards the Generation of a Text-Based IDE from a Language Metamodel

Anneke Kleppe*

University Twente, Netherlands
a.kleppe@utwente.nl

Abstract. In the model driven world languages are usually specified by a (meta) model of their abstract syntax. For textual languages this is different from the traditional approach, where the language is specified by a (E)BNF grammar. Support for the designer of textual languages, e.g. a parser generator, is therefore normally based on grammars. This paper shows that similar support for language design based on metamodels is not only possible, but is even more powerful than the support based on grammars. In this paper we describe how an integrated development environment for a language can be generated from the language's abstract syntax metamodel, thus providing the language designer with the possibility to quickly, and with little effort, create not only a new language but also the tooling necessary for using this language.

Keywords: metamodeling, domain specific languages, text-based languages, parsing, compilers, IDE, generation.

1 Introduction

Currently, there is an increasing interest in the design of languages that are used somewhere in the software development process. First, *domain specific modeling languages* (DSMLs) are becoming more and more important. DSMLs are languages for modeling software, which are focused on describing a certain aspect or viewpoint of a software system. Second, there is a steady demand for occasional or little languages, i.e. languages that are used for a relatively small amount of time by a small group of people. For instance, in large, long-running projects often small (scripting) languages are being build that enable automation of specific, reoccurring tasks in that project. These languages are known under various names, amongst which *domain specific languages* [1]. Special to both types of DSLs is that they have a limited number of users, compared to general software languages like Java, C#, and UML.

Often these new languages are specified by a metamodel, which accounts for the popularity of metamodeling toolkits like the Eclipse Modeling Framework (EMF) [2] and Microsoft's DSL tools [3]. It is our view that metamodeling toolkits should support the creation of a language in full. Not only should they aid the language designer in his/her task of creating the metamodel, but they should also support the language designer in creating the tooling for the people that are going to use the

* The author is employed in the GRASLAND project funded by the Dutch NWO (project number 612.063.408).

language. Note that we use the term *language designer* for the person who creates the new language, and *language user* for the person who uses the newly created language and its supporting tools.

The current demands on tooling are high. For instance, if a dedicated text editor is provided, it should have syntax-highlighting and code-completion. Specially for modeling languages, tooling must include code generation software and should preferably include a debugger that is able to address the language user in terms of the domain specific model instead of the code language.

Languages targeting a limited number of users, do not warrant the effort in building such sofisticated tools, simply because the costs are too high. The only way that a language designer is able to create sofisticated tooling for such languages, is when most of it is generated by the metamodeling toolkit. In other words, the metamodeling toolkit needs to be able to generate an integrated development environment (IDE) for the language specified by the metamodel.

This paper describes the first steps towards the realisation of such a metamodeling toolkit, more specifically it describes the generation of a compiler front-end for a text-based concrete syntax of a language, based on the metamodel specification of that language. As this work is conducted within the Grasland project, our metamodeling toolkit is, for lack of a better one, named the Grasland toolkit. The Grasland toolkit is implemented in the form of a number of Eclipse plug-ins that build upon the functionality provided by the Octopus tool [4].

Section 2 of this paper outlines the process of language design as it is supported by the Grasland toolkit, and it establishes the terminology used. Sections 3 and 4 describe the two transformations that generate a grammar from a metamodel. Section 5 describes the generated static semantic analyzer. Finally, Section 6 describes future and related work.

2 Preliminaries

This section outlines the process of language design as it is supported by the Grasland toolkit, and it establishes the terminology used in this paper. Furthermore, the arguments for our approach are stated in the last subsection.

2.1 Terminology

In this paper we will use the following terms, which are formally defined to be special types of graphs.

- *Abstract Syntax Model* (ASM): a metamodel that specifies the abstract syntax of the language, which will be called *L*.
- *Abstract Syntax Graph* (ASG): an instance of the abstract syntax model.
- *Concrete Syntax Model* (CSM) or *Parse Model* (PM): a metamodel that specifies a concrete syntax of the language. (When talking about text-based syntaxes we will use *parse model*, when talking about graphical syntaxes we will use *concrete syntax model*.)

- *BNFset*: the set of (E)BNF rules that specifies a text-based concrete syntax of the language. Note that there is a correspondence between a BNFset and a parse model.
- *Parse Graph* (PG) or *Parse Tree* (PT): an instance of the parse model. (When talking about text-based syntaxes we will use *parse tree*, when talking about graphical syntaxes we will use *parse graph*.)
- *Navigations*: the set of outgoing associations and attributes of a metaclass.

Furthermore, we assume that a language can have multiple concrete syntaxes, and a concrete syntax can be either textual, graphical, or a hybrid one that combines textual parts with graphical ones, e.g. a table representation.

2.2 The Process of Language Design

Central to the process of language design as it is supported by the Grasland toolkit, is the ASM of the language. To create the tooling for the language user, the language designer needs to perform the tasks in Table 1, which are dependent on the type of concrete syntax used. Next to this, the language designer is likely to create an exchange format for abstract syntax graphs, for instance based on XML, as well as transformations from the ASM to various other metamodels, one of which will probably implement code generation.

Table 1. Tasks of a language designer for the two types of concrete syntax

Step	Text-based concrete syntax	Graphical concrete syntax
1	*Create the PM, which will include classes that represent references to other elements in the parse tree.*	*Create the CSM, which will include classes that represent graphical items like rectangles and lines.*
2	*Create the EBNF grammar, which will include keywords. Take an existing parser generator, (re)write the grammar for this generator, and generate a parser that will produce the parse tree from a text file.*	*No action needed. (Usually the CSM suffices to create a syntax-directed graphical editor, thus there is no need to create a parser.)*
3	*Create a model transformation from parse tree to abstract syntax graph (this is often called static analysis, it includes binding).*	*Create a model transformation from parse graph to abstract syntax graph.*
4	*Create a text editor dedicated to this concrete syntax, with syntax highlighting etc.*	*Create a graphical editor dedicated to this concrete syntax.*
5	*Create a tool chain such that an abstract syntax graph is created from a text file.*	*Create a tool chain such that an abstract syntax graph is created from a diagram.*

In this paper we will show how all of steps 1, 2, 3, and 5 for text-based syntaxes can be automated, i.e. none of the products are created by hand, they are all generated by the Grasland toolkit. Automation of step 4 is also possible, but not yet implemented in the Grasland toolkit.

2.3 Outline of Our Approach

Traditionally, when a new textual language is created, the main activity is to produce the BNFset. Next, a parser is created using some or other parser generator, e.g. [5, 6, 7]. The other parts of the language's compiler are implemented by hand, often by creating treewalkers that traverse the parse tree generated by the parser, as shown in Figure 1 (the shaded parts are created by the language designer). There is, in most cases, no explicit definition of the PM, nor of the ASM, although one can always extract the set of pure BNF rules, which might serve as a PM description, from the parser generator input.

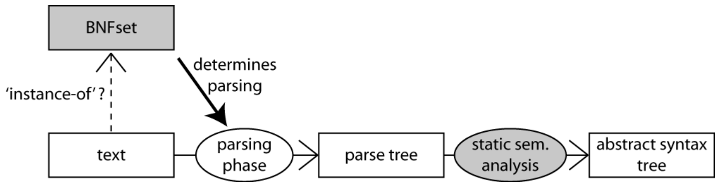

Fig. 1. The normal elements in a compiler.

In the Grasland approach, the only manual activity is to create the ASM, i.e. a metamodel and its invariants. From the ASM we generate a PM, which upholds certain requirements that will be explained in Section 3. This transformation is called *asm2pm*. From the PM we generate a BNFset, which - for practical purposes - can be generated in a format that is processable by the JavaCC parser generator [6]. This transformation is called *pm2bnf*. Next JavaCC generates a parser, which is able to produce a parse tree that is an instance of the PM in the sense that the nodes in the parse tree are instances of the Java classes that correspond to the classes in the PM. To implement the static semantic analysis, a tool is generated that transforms a parse tree into an ASG. This tool implements a model transformation from PM to ASM. Figure 2 shows the various elements in the Grasland approach; again the manually created elements are shaded.

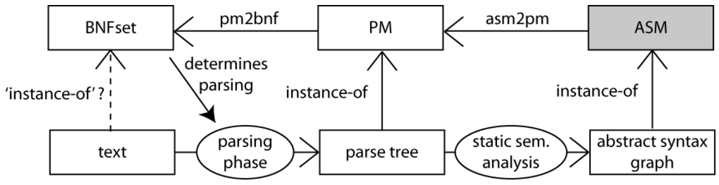

Fig. 2. The alternative process using metamodels

2.4 Rationale of the Approach

Specially for text-based languages, our approach is very different from the traditional process. Instead of focusing on BNF rules, the language designer will focus on the ASM. The PM and BNFset are automatically generated from the ASM. A number of arguments support this new design process.

The first argument is that at the start of the language creation process it need not be clear whether the new language is text based or graphical, and often the new language should support multiple syntaxes. So a specification of the concrete syntax cannot be a good starting point.

Second, although compiler construction is a formally defined area of expertise, it has one obvious omission, which is that the true ASM is not defined at all. What is usually called an abstract syntax tree in compiler construction, we call a parse tree. The abstract syntax tree is embellished with binding information and often reshuffled to produce what we call an abstract syntax graph. Note that in compiler construction the term abstract syntax tree is used for both formats. More importantly, it is the abstract syntax tree that is used for further handling, like code generation, which means that these phases lack a formal description. On this point metamodeling certainly has something to add to the area of compiler construction.

Furthermore, the power of metamodelling is larger than the power of BNF. One can express more in a metamodel. Therefore, starting with a BNF grammar and creating a metamodel from the grammar, as for instance described in [8], will result in a restricted metamodel. Most certainly, this metamodel will not be the one that the language designer wants to use as ASM.

A fourth argument is that although the syntax of the majority of programming languages can be classified as context-free, the languages themselves are often context-sensitive. That is, the static analysis phase of the compiler adds context sensitive information. For instance, variable binding may be considered context-sensitive information because a variable 'a' is not always bound to the same variable declaration, the binding depends on the context in which ‘a’ is found. So, to support the language designer in creating a complete toolset for a text-based concrete syntax, we need not only consider parsing but also static analysis. Currently, there are many parser generators, but as far as we know there are no generators for static semantic analysers.

From one argument comes another. Now that we have established that we have a need for a static semantic analyser, it is a good choice to generate the parse model from the abstract syntax model. In this way we have full control over the differences between the two models and therefore we will be able to automatically generate the static semantic analyser that bridges the two.

Another consideration for our choice of design process, is that the field of parsing and compiler construction is very well established. The parser generators that result from this research are tried and tested and can be used without further ado.

A final argument is a reduced ‘time to market’. In the Grasland approach the language designer is able to 'play' with the abstract syntax model and for each change in this model he will be able to generate a working IDE with a single push of a button.

This means that testing the changes takes as least effort as possible. Although, as the title of this paper tells, we are still working towards a toolkit that is able to generate a complete IDE, our experiments with the generation of parts of this IDE are promising.

The next sections describe the how the steps in Table 1 are implemented in the Grasland toolkit.

3 The ASM to PM Transformation

This section describes the algorithm for the *asm2pm* transformation. This algorithm implements the creation of the parse model (or CSM), which includes classes that represent references to other elements in the parse tree. Note that this algorithm actually is defined on the meta meta level, i.e. it is not a transformation of model to model, but of metamodel to metamodel.

The algorithm, which is outlined in List 1, makes use of the composite - reference distinction in associations in the metamodel. We use a formal definition of metamodel that ensures that in any instance of the metamodel the composites form a subgraph that is really a tree. The composite relationships are subsequently used in the *pm2bnf* transformation to construct the BNF grammar. In the case that the subgraph formed by the composite associations is not a tree, but a set of unrelated trees (a forest), the algorithm will produce a set of unrelated sets of grammar rules. It is up to the language designer to decide whether this is (un)desired. Figure 4 shows an example of an ASM, Figure 3 shows the PM that is automatically generated from this ASM. The differences are marked by the colour of the classes and the font of the role names.

Note that for each of the classes for which a reference class is created (step 3), the language designer must indicate which attribute of String type is used as identifier. This knowledge is used in the static semantic analyser to implement the binding. Implementations of the Java counterparts of the classes in the ASM are automatically generated using the functionality of the Octopus tool, and the same is done for the PM.

3.1 Possibilities to Tune the *asm2pm* Transformation

The algorithm in List 1 is fully automatic and produces a parse model without any extra user effort. However, if the algorithm for the *asm2pm* transformation is executed as is, then the differences between the ASM and PM are minimal. Often the language designer wants a larger difference between the two, therefore there are options to tune the *asm2pm* transformation. Note that these differences are taken into account in the generation of the static semantic analyser as well.

The first option is to indicate that certain metaclasses in the ASM should not appear at all in the PM. Examples are the classes *PrimitiveType* and *NullType* in Figure 3. These types are only present in the ASM to provide for a number of basic elements in the language, but the language user is not meant to create new instances of these metaclasses. The language designer can indicate that these classes are hidden to the concrete syntax. Currently this is done by means of a properties file. We are investigating the possibility of indicating hidden elements using Eclipse project properties.

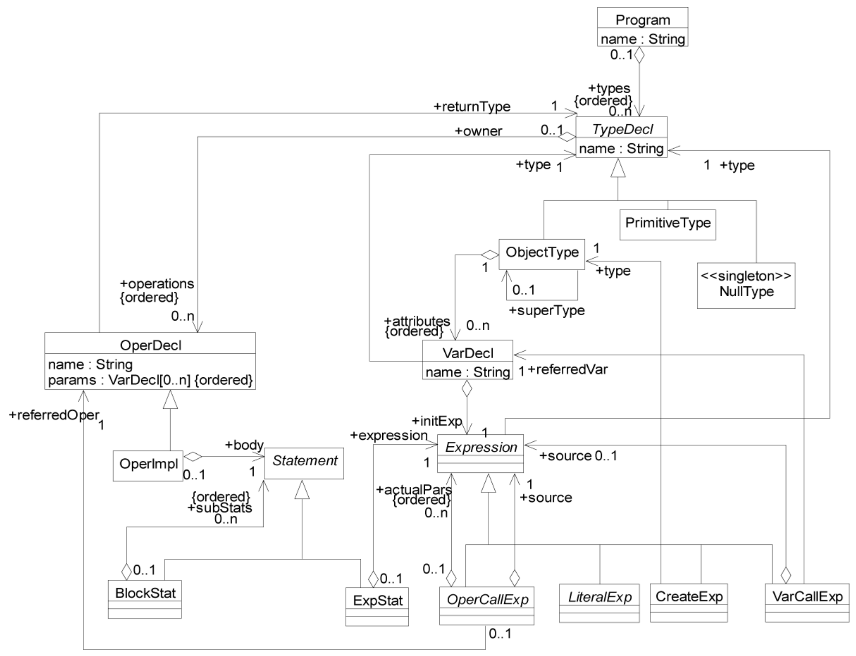

Fig. 3. Example ASM

1. Every class in the ASM becomes a class in the PM. The language designer may indicate prefix and postfix strings that are used to name the classes in the PM, in order to distinguish them from the classes in the ASM. E.g. the ASM class named *Variable-Declaration* becomes the PM class named *prefixVariableDeclarationpostfix*.
2. Every composite association is retained.
3. For every non-composite association from class A to class B a new class is introduced that represents a reference to an instance of class B. A new composite association is added from class A to this new reference class. The role name of the old association is copied to the new one, as well as the multiplicities.
4. Every attribute with non-primitive type, i.e. whose type is another class in the meta-model, is transformed into a composite association from the owner of the attribute to the class that is the attribute type. The name of the attribute becomes the role name. Any multiplicities are copied.
5. Enumerations and datatypes are retained.
6. Additionally, three attributes are added to every PM class. They hold the line number, column number, and filename of the parsed instance of the class.

List. 1. The algorithm for asm2pm

The second option is to indicate that certain attributes and outgoing associations of a metaclass need not be present in the input text file, instead their value will be determined based on the values of other elements that are present. In fact these elements

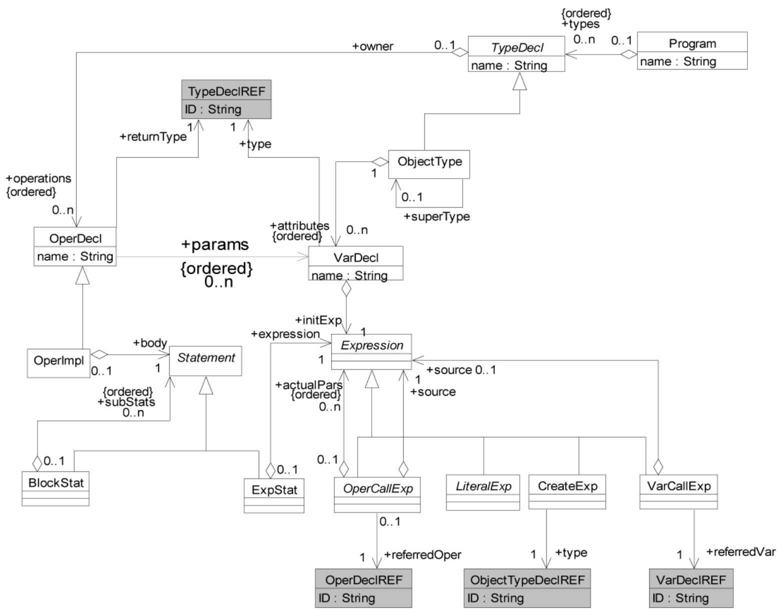

Fig. 4. Example PM

are what is known in OCL [9, 10] as *derived* elements. The language designer may indicate that a certain element need not be taken into account in the parse model, if an OCL derivation rule for this element in the ASM is provided. An example of a derived element in Figure 3 is the *type* of an *Expression*.

4 The PM to BNF Grammar Algorithm

This section describes the algorithm for the *pm2bnf* transformation, which implements the creation of the BNF rules that are used by a parser generator to produce a parser. Note that like the *asm2pm* algorithm, this algorithm too resides on the meta meta level, i.e. it is not a transformation of model to model, but of metamodel to metamodel. Alanen and Porres [11] present algorithms for the relation between PM and BNFset, which we have used and extended.

The generation of the BNFset from the PM is implemented in a single algorithm. Yet, the language designer may choose between two different output formats; either BNF, or a grammar that can directly be used as input to the JavaCC parser generator [6]. The BNF grammar that is produced is actually an extension of EBNF that uses labelling of non-terminals in the right hand side of a grammar rule. (Not to be confused with Labelled BNF [12], which uses labels on the non-terminals at the left hand side of each rule.) The labels correspond with the names of the attributes or association roles in the PM. An example in which the labels are highlighted, can be found in List 3.

1. Every class in the PM becomes a non-terminal in the grammar. The rules for these non-terminals are formed according to the following rules.
2. If a class has subclasses then the BNF rule becomes a choice between the rules for the subclasses. All attributes and navigations of the superclass are handled in the subclass rules.
3. For every composite association from A to B, B will appear in the right hand side of the grammar rule for A. The multiplicity is the same as in the association (for 0..1, 1, 0..*, 1..*; multiplicities of the form 3..7 are considered to be specified using invariants). Using an extension of BNF, we associate the rolename with the non-terminal in the right hand side of the rule.
4. Every attribute, all of which have a primitive type, is transformed into an occurrence of a predefined non-terminal for that primitive type in the right hand side of the rule for its owner. (We support the primitive types String, Integer, Real.)
5. Every attribute that has Boolean type, is transformed into an optional keyword. If present, the attribute has value true, if not the attribute's value is false.

List. 2. The algorithm for pm2bnf

The input for the JavaCC parser generator is such that the generated parser produces instances of the Java implementations of the classes in the PM. The algorithm that implements *pm2bnf* is given in List 2. An example can be found in List 3, which shows the BNF rules generated from the parse model in Figure 4. Note that tokens in the right hand side of the grammar rules are surrounded by angled brackets ('<' and '>').

4.1 Possibilities to Tune the *pm2bnf* Transformation

The algorithm in List 2 is fully automatic and produces a grammar without any extra user effort. However, there are a number of differences between the metamodel formalism used for the parse model and the BNF formalism and the language designer is able to influence how these differences appear in the generated grammar, thus tuning the *pm2bnf* generation.

The most apparent difference is the lack of ordering in navigations from a metaclass, versus the ordering of the elements in the right hand side of a BNF rule for a non-terminal. To indicate a certain ordering in the BNF rules the language designer can associate an index to all navigations This is done in a so-called properties file. An example can be found in List 4, where the order of the navigations from the metaclass *ObjectType* in Figure 3 is given. The first element to be included in the right hand side of the corresponding BNF rule is the attribute called *name*, the second is the optional reference to a super type, etc. Without directions from the language designer the Grasland toolkit will randomly assign an ordering.

Another difference between a metamodel and a grammar is that most grammar rules contain one or more keywords, whereas the metamodel does not. These keywords are relevant in the parser because they enable the parser to differentiate between language elements (rules). Therefore the Grasland toolkit provides the option

```
************ The grammar rules **************
1. BlockStat ::= <CURLY_OPEN> ( subStats:Statement )* <CURLY_CLOSE>
2. CreateExp::= <CREATEEXP_BEGIN> type:ObjectTypeREF <CREATEEXP_END>
3. ExpStat ::= expression:Expression
4. Expression ::= (LiteralExp
        | OperCallExp
        | VarCallExp
        | CreateExp)
5. ObjectTypeREF ::= ID:<IDENTIFIER>
6. ObjectType ::= <OBJECTTYPE_BEGIN> name:<IDENTIFIER> [
   <OBJECTTYPE_SUPERTYPE_BEGIN> superType:ObjectTypeREF] [
   attributes:VarDecl( <SEMICOLON> attributes:VarDecl )* <SEMICOLON> ] (
   operations:OperDecl )* <OBJECTTYPE_END>
7. OperCallExp ::= referredOper:OperDeclREF <BRACKET_OPEN> [ actual-
   Pars:Expression( <COMMA> actualPars:Expression )* ] <BRACKET_CLOSE> [
   <OPERCALLEXP_SOURCE_BEGIN> source:Expression ]
8. OperDeclREF ::= ID:<IDENTIFIER>
9. OperDecl ::= (OperImpl)
10. OperImpl ::= name:<IDENTIFIER> <BRACKET_OPEN> [ params:VarDecl(
   <COMMA> params:VarDecl )* ] <BRACKET_CLOSE> <COLON> return-
   Type:TypeREF ( locals:VarDecl )* body:BlockStat
11. Program ::= <PROGRAM_BEGIN> name:<IDENTIFIER> startExp:ExpStat (
   types:Type )* <PROGRAM_END>
12. Statement ::= (BlockStat
        | ExpStat) <SEMICOLON>
13. TypeREF ::= ID:<IDENTIFIER>
14. Type ::= (ObjectType)
15. VarCallExp ::= referredVar:VarDeclREF [ <VARCALLEXP_SOURCE_BEGIN>
   source:Expression]
16. VarDeclREF ::= ID:<IDENTIFIER>
17. VarDecl ::= name:<IDENTIFIER> <COLON> type:TypeREF [
   <VARDECL_INITEXP_BEGIN> initExp:Expression]
************ The token definitions **************
        CREATEEXP_BEGIN ::= "new"
        CREATEEXP_END ::= "()"
        NULLLITEXP_BEGIN ::= "null"
        OBJECTTYPE_BEGIN ::= "class"
        OBJECTTYPE_END ::= "end_class"
        OBJECTTYPE_SUPERTYPE_BEGIN ::= "extends"
        OPERCALLEXP_SOURCE_BEGIN    ::= "on"
        OPERDECL_LOCALS_BEGIN       ::= "locals"
        PROGRAM_BEGIN ::= "program"
        PROGRAM_END ::= "end_program"
        VARCALLEXP_SOURCE_BEGIN     ::= "on"
        VARDECL_INITEXP_BEGIN       ::= "="
        IDENTIFIER ::= ["a"-"z", "A"-"Z", "_"] ( ["a"-"z", "A"-"Z", "0"-"9", "_" ] )*
```

List. 3. The resulting BNF rules

```
BLOCKSTAT_BEGIN=CURLY_OPEN
BLOCKSTAT_END=CURLY_CLOSE
CREATEEXP_BEGIN=new
CREATEEXP_END=()
NULLLITEXP_BEGIN=null
OBJECTTYPE_ATTRIBUTES_END=SEMICOLON
OBJECTTYPE_ATTRIBUTES_SEPARATOR=SEMICOLON
OBJECTTYPE_BEGIN=class
OBJECTTYPE_END=end_class
OBJECTTYPE_SUPERTYPE_BEGIN=extends
OBJECTTYPE_ORDER_1=name
OBJECTTYPE_ORDER_2=superType
OBJECTTYPE_ORDER_3=attributes
OBJECTTYPE_ORDER_4=operations
OPERCALLEXP_ACTUALPARS_BEGIN=BRACKET_OPEN <MANDATORY>
OPERCALLEXP_ACTUALPARS_END=BRACKET_CLOSE <MANDATORY>
OPERCALLEXP_ACTUALPARS_SEPARATOR=COMMA
```

List. 4. Part of the properties file for pm2bnf

for the language designer to indicate which keywords should be used in the grammar rule corresponding to a metaclass instance. Without keyword directions the Grasland toolkit will generate keywords based on the class and association role names.

For each metaclass there are two options to use a keyword: (1) at the start of the right hand side, (2) at the end of the right hand side. An example is the keyword 'new', indicated by `CREATEEXP_BEGIN`, that should appear at the start of a CreateExp instance. For each navigation there are three possibilities: (1) a keyword that should appear before the navigated element, (2) a keyword that should appear after the element, and (3) a keyword that separates the elements in a list. The last is sensible only when the multiplicity of the association is larger than one. In case that the element is optional (i.e. lower bound of multiplicity is zero), the language designer is able to indicate whether the keyword should still appear even if the element is not present. This is useful, for instance to indicate that the opening and closing brackets of a parameter list should be present even if there are no parameters. An example can be found in List 4, where the brackets are mandatory for the navigation `OPERCALLEXP_ACTUALPARS`. Note that a keyword in this approach can be any string, including brackets etc.

A third difference between a metamodel and a grammar is that the parsing algorithm used poses a number of requirements on the rules. For instance, the JavaCC parser generator creates LL(n) parsers, and its input should be an LL(n) grammar, where n indicates the number of lookahead tokens used. If the language designer decides to create a grammar with too few keywords, then the parser generator will produce errors and/or warnings. As the Grasland toolkit is a prototype we regard resolving these to be the responsibility of the language designer for now. By adding more keywords or by adding (by hand) lookaheads to the generated grammar the language designer will always be able to generate a grammar that is correct. Even so, the Grasland toolkit provides a minimal support in the form of the generation of lookaheads in the rule for a class with subclasses, where choice conflicts are likely because the attributes and navigations of the superclass appear in the rules for each subclass.

5 The Static Semantic Analyser

The two most important aspects of static semantic analysis are binding and type checking. This section describes how the Grasland toolkit implements these issues.

5.1 Binding

Binding is the general term for the binding of names to their definitions. These names may refer to types, for instance in a variable declaration, or to variables or operation/ functions, for instance in assignments or operation calls. Binding is often context sensitive in the sense that not all occurrences of the same name are bound to the same definition, depending on the context of the name it may be bound to a different definition, sometimes even to a definition of a different kind of element. For instance, in one context “message” may be bound to a variable, in another to a type or operation. Such a context is usually called a *namespace*.

In a Grasland generated PM all elements that need to be bound are instances of reference metaclasses (see List 1, rule 3). For each reference metaclass we know the metaclass from which it is derived. We call this metaclass the *target metaclass*.

Simple Binding. The most primitive way of binding these elements is by searching the parse tree for all instances of the target metaclass and comparing their names with the name of the element to be bound. This is the default implementation of binding.

However, it is possible for the language designer to indicate that certain metaclasses in the ASM act as namespaces. For instance in our example, the classes *Type*, *OperDecl*, and *Program* all act as namespaces. If there is a class labelled as namespace, then the *asm2pm* algorithm will produce a metamodel in which every class has the operation *findNamespace,* which will return the element’s surrounding namespace. An *INamespace* interface is added to the metaclass(es) that act as namespaces for this purpose. The implementation of each of the *findNamespace* operations is specified by an OCL body expression.

The binding algorithm is in this case implemented as follows. First, find the surrounding namespace of the instance of the reference metaclass, then search this namespace for occurrences of the target metaclass and compare their names with the name of the reference element. If a match is found then the reference is bound to the found instance of the target metaclass. If no match is found, then the surrounding namespace of the searched namespace is searched in the same manner, and so on and so forth, until the outmost namespace has been searched. If no match was found, an error message is given. The search of a namespace goes down the parse tree to the leaves of the tree, unless one of the nodes is itself a namespace, then the search stops at this node.

Complex Binding. A more complex way of binding is based not only on the name of the reference element but also on the occurrence of surrounding elements. For instance, the binding of an operation call is usually determined not only by the name of the operation but also by the number and types of the parameters. In our example, the link called *referredOper* between an *OperCallExp* instance and an instance of the reference class *OperDeclREF* is an example of such a complex binding.

The language designer may indicate the use of a complex binding by stating an invariant in the ASM that must hold after the reference element is bound. For instance, for the example in Figure 3, the following entry in the properties file indicates the use of complex binding.

```
OperCallExp.referredOper=paramsCheck
```

In this case, the invariant called *paramsCheck* must be present for the class *OperCallExp*. It is specified by the following OCL expression. Note that the use of names for invariants is a standard OCL feature.

```
context OperCallExp
inv paramsCheck: referredOper.params.type = actualPars.type
```

Having this in place the Grasland toolkit implements complex binding more or less in the same manner as simple binding. First a list of possible matches is found based on the name only, then for each element in this list the invariant is checked. If no correct element is found then the search continues in the next namespace, etc.

An advantage of this approach is that normally these invariants need to be part of the ASM anyhow, so there is no extra effort needed from the language designer. Another advantage is that all the information that the language designer must provide is based on the ASM. The ASM is truly the focus of the language design process, even though a text-based language is being specified. This leaves room for the creation of multiple views each based on a different concrete syntax, with the possibility of combining textual and graphical views all working together on the same ASG.

Please note that this algorithm implements static semantic checking. This means that dynamic binding and dynamic scoping are by definition not covered.

5.2 Static Checking

An important observation with regard to static checking is that the rules that are checked during this phase are easily specified by OCL invariants on the ASM. These are the so called well-formedness rules. For instance, in our (simple) example the following rule provides enough information to perform type checking.

```
context VariableDecl
inv: self.type = initExp.type
```

Static checking is therefore implemented in the generated static semantic checker as the checking of invariants on the abstract syntax graph. Whenever an invariant is broken, an error message is given to the language user.

Even more complex forms of type checking involving type conformance can be handled in this manner. For instance, given the existence of an operation in the *Type* class that implements the type conformance rules, the following invariant allows for type checking with type conformance. The type conformance operation itself can also be specified using OCL.

```
context VariableDecl
inv: self.type.conformsTo(initExp.type)

context Type::conformsTo( actualType: Type) : Boolean
body: if ( actualType = self)
   then true
   else if not actualType.superType.oclIsUndefined()
        then self.conformsTo( actualType.superType)
        else false
        endif
   endif
```

The advantage of this approach is that the invariants can be used for all concrete syntaxes that may be defined for the ASM. Thus static checking becomes a common functionality instead of a functionality that needs to be implemented for each of the different concrete syntaxes.

6 Conclusion and Related Work

In this paper we have shown that it is possible to generate (parts of) an IDE, more specifically the front-end of a text-based compiler, from a metamodel. Given the tuning possibilities offered in both the *asm2pm* and *pm2bnf* transformations, the language designer can influence the resulting grammar considerably, with minimal effort from his part. Not yet mentioned is the fact that the Grasland toolkit is able to produce a deparser for the textual syntax, as well as a parser and deparser for an XML based interchange format for ASGs, and that all the generated tools described in this paper are combined to create an integrated language user environment. Because we do not foresee large difficulties in generating a language-specific editor, we conclude that it is indeed feasible to generate a text-based IDE from a metamodel, as was our initial ambition.

The idea of generating an IDE from a language specification is not new. In fact a number of metacase tools exist that perform this task, e.g. [13, 14]. What is new in our approach is that the focus of the language designer is on the metamodel, not on the BNF grammar. Keeping the focus on the ASM, instead of the grammar, is much more in line with the model driven process in which instances of the ASM are being transformed.

The process described by Wimmer and Kramler [8] starts with a grammar, from which a (raw) metamodel is built. Because this metamodel is (as they call it) “not user friendly”, it is transformed into an ASM. The Eclipse plug-in set xText [15] also starts with a grammar and produces a metamodel. Hearnden et. al. describe the use of Anti-Yacc [16], which also forces the language designer to create a grammar. This grammar and a metamodel are fed to Anti-Yacc, which generates the bridging between the PM and the ASM. However, no evidence is given of how binding is handled. Finally, HUTN [17] uses an abstract base syntax that is applied to all models, which is customized to exploit specific properties of particular models. Again, our approach offers more flexibility to the language designer.

The graph grammar community has also been working on generating IDEs, see for instance [18, 19, 20]. However, their focus is on visual concrete syntaxes. Likewise, Fondement and Baar [21] describe a way to specify a visual syntax. Here too, a completely different metamodel is defined for the concrete syntax. Their approach is complementary to the one described here, as we focus on textual syntax.

The only other reference that focuses on the ASM instead of the grammar, is Jouault et al. [22]. They define a template language in which the language designer may specify the textual syntax. This syntax specification is very similar to BNF rules, thus this approach does not relieve the language designer from writing a grammar(-like) specification. Furthermore, they do not deal with complex references, nor do they handle type checking.

Concluding we can state that the Grasland toolkit produces a good, workable IDE from a metamodel. As is always the case with the generation of software, the creation of an IDE by hand could produce a better and more efficient IDE. However, it is important to compare the time and effort needed to create a reasonable well IDE using the Grasland toolkit with the time and effort needed to create a perfect IDE manually. We are confident that the comparison will favour the Grasland approach.

References

[1] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain- specific languages. ACM Comput. Surv. 37(4), 316–344 (2005)

[2] The Eclipse Modeling Framework (2007), http://www.eclipse.org/emf

[3] Microsoft DSL tools. (2007), http://msdn.microsoft.com/vstudio/DSLTools/

[4] Octopus: OCL Tool for Precise UML Specifications (2007) http://www.klasse.nl/octopus

[5] Antlr (2007), http://www.antlr.org/

[6] JavaCC (2007), https://javacc.dev.java.net/

[7] Johnson, S.C.: Yacc – yet another compiler compiler. Technical Report CSTR 32, Bell Telephone Labs (July 1974)

[8] Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: WiSME 2005 4th Workshop in Software Model Engineering (2005)

[9] OCL 2.0 specification. Technical Report ptc/2005-06-06, OMG (2005)

[10] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2003)

[11] Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS, mar (2004)

[12] Forsberg, M., Ranta, A.: Labelled BNF: a highlevel formalism for defining well-behaved programming languages. In: Proceedings of the Estonian Academy of Sciences: Physics and Mathematics, number 52, pp. 356

[13] Reps, T., Teitelbaum, T.: The synthesizer generator. In: SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, pp. 42–48. ACM Press, New York, NY, USA (1984)

[14] MetaEdit+ (2007), http://www.metacase.com/

[15] xText (2007), http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf

[16] Hearnden, D., Raymond, K., Steel, J.: MOF-to-text. In EDOC, pp. 200–211. IEEE Computer Society, Los Alamitos (2002)

[17] Human-usable textual notation (HUTN) specification. Technical Report formal/04-08-01, OMG (2004)

[18] Bardohl, R.: GenGEd: Visual Definition of Visual Languages based on Algebraic graph Transformation. PhD thesis, TU Berlin, Berlin, Germany (1999)

[19] Minas, M.: Generating meta-model-based freehand editors. In: Proceedings of the third International workshop on graph based tools, 2006, EASST, pp. 1–11 (September 2006)

[20] de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) ETAPS 2002 and FASE 2002. LNCS, vol. 2306, Springer, Heidelberg (2002)

[21] Fondement, F., Baar, T.: Making metamodels aware of concrete syntax. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA. LNCS, vol. 3748, pp. 190–204. Springer, Berlin Heidelberg (2005)

[22] Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE ’06. Proceedings of the 5th international conference on Generative programming and component engineering, pp. 249–254. ACM Press, New York, NY, USA (2006)