

HUTN as a Bridge between ModelWare and GrammarWare – An Experience Report

Pierre-Alain Muller¹, Michel Hassenforder²

¹ IRISA/INRIA, Rennes, France, pierre-alain.muller@irisa.fr

² MIPS, Université de Haute-Alsace, France, michel.hassenforder@uha.fr

Abstract. In this paper we report on our experience using HUTN as a bridge between ModelWare and GrammarWare, to generate parsers and editors for DSLs defined under the shape of metamodels. We describe the problems that we have encountered with the ambiguities of the current HUTN specification and discuss how this specification may be fixed to be usable with grammar-driven tools.

1 Introduction

MDE (Model-Driven Engineering) is a form of generative engineering, by which all or parts of software applications are generated from models. MDE can be seen as an integrative approach combining existing techniques originally developed in different technical spaces.

In this paper we explore such combination, where we have been using HUTN¹ as a bridge between metamodels and parsers. Our goal is to build models which conform to given metamodels, in a textual way. In other words, we specify DSLs under the shape of metamodels, and we want to generate basic textual tools to work with these DSLs.

We are mainly interested in quick turn-around. We have been considering that for the purpose of prototyping DSLs, defining the details of a specific concrete syntax was a waste of time, and we have been looking for ways to derive a default concrete syntax from a metamodel. This is the textual equivalent of a previous work that we did in the context of an object editor generated from class diagrams in the Netsilon² tool.

The idea is to automate the generation of the grammar of a textual language, whose concepts are those specified in the metamodel, and whose concrete syntax (without much syntactic sugar) follows some generic and predefined schema. This generated grammar can then be fed into existing GrammarWare tools, to automatically generate parsers and editors for DSLs.

For the generic concrete syntax we have chosen HUTN which specifies a generic textual notation which can be customized for any metamodel conforming to the MOF. HUTN has been designed with human usability in mind, and this is achieved through consideration of the successes and failures of common programming languages.

HUTN offers three main benefits. (1.) It is a generic specification that can provide a concrete HUTN language for any MOF model; (2.) The HUTN languages can be fully automated for both production and parsing; and (3.) The HUTN languages are designed to conform to human-usability criteria.

The languages generated for each metamodel are all different, but are very similar in structure as they are generated from patterns parameterized with the specifics of each metamodel. HUTN is extremely useful for both rapid development and prototyping, as any change to the metamodel can automatically regenerate the metamodel-specific HUTN language (and generated tools).

Surprisingly we have not found open-source HUTN parsers (the DSTC has an early implementation, named TokTok³, but source files are not available). This is why we have decided to develop our own HUTN generator. The experience of implementing this generator has shown us where there exists ambiguity in the HUTN specification; we highlight and discuss these issues, with some suggested options for fixing the problems.

This paper is organized as follows. After this introduction, we present some related works, and then we present in detail 6 major problems that we have found in the HUTN specification, and which must be fixed so as to generate grammars suitable for parser generators. Finally the conclusion summarizes the paper, and outlines future directions.

2 Related works

Much of the concepts behind our work take their roots in the seminal work conducted in the late sixties on grammars and graphs and in the early eighties in the field of generic environment generators (such as Centaur⁴) that, when given the formal specification of a programming language (syntax and semantics), produce a language-specific environment.

The generic environment generators sub-category has recently received significant industrial interest; this includes approaches such as Xactium⁵, or Software Factories⁶. Among these efforts, it is Xactium which comes closer to our work. The major difference is that we use HUTN as a generic concrete syntax, while Xactium allows concrete syntax customization.

A similar experience⁷ with turning an OMG specification (OCL) into a grammar acceptable by a parser generator has been described by D. Akehurst and O. Patrascoiu.

An interesting discussion about mapping MOF metamodels and context-free grammars can be found in a technical report⁸ by M. Alanen and I. Porres.

At the meta-metalanguage level, there are several textual languages such as Emfatic⁹, KM3¹⁰ or Kermeta¹¹ which are used to build meta-models. These languages are not intended for model specification, but it is still interesting to study how they deal with modeling constructs and tooling. It is worth to note that our work is integrated in the Kermeta workbench, and is used to easily generate models which conform to the metamodels described in Kermeta, for instance for test purposes.

3 Issues and problems with the HUTN specification

In this section we examine the reasons that make the HUTN specification ambiguous from a grammar point of view, and we propose and motivate solutions to make the HUTN grammar suitable for automatic processing by tools. The problems that we have encountered are mainly related to the optional or overlapping constructions, and collection delimitation. In this paper, we will use the same notation for the grammar rules as the one used in the HUTN specification.

3.1 Problem with Keyword Attributes

The rule which describes the notion of KeywordAttribute

```
KeywordAttribute ::= '~'? <AttributeName>?
```

contains two optional fields, which leads to the following 4 choices:

```
KeywordAttribute ::= ε
                  | '~'
                  | <AttributeName>
                  | '~' <AttributeName>
```

Semantically, the two last choices correspond respectively to the `True` and `False` values affected to `Boolean` attributes. Thus, using the model in Figure 1, it is possible to create an instance with the following expression:

```
A { B ~C };
```

A
-B:Boolean
-C:Boolean

Figure 1 : A simple model.

However, it is also possible to write `A { ~ }`, or even `A { }` (with the two first choices). In the first case we negate an unknown attribute, while in the second case we state something such as the existence of an unknown attribute. Additionally, the first empty choice introduces an ambiguity in the rule `ClassContent`. This rule describes the notion of collection (potentially empty) of `KeywordAttribute` (among others). Simplifying the writing we get:

```
ClassContents ::= KeywordAttribute *
```

Syntactically, the fact that `KeywordAttribute` may be empty and that the collection can be empty as well, leads to an ambiguity in the syntax tree. If we use the previous rule, with a `A` letter as `KeywordAttribute`, and an input text composed of two `A` letters, then the syntax tree can be one of the following (among many others).

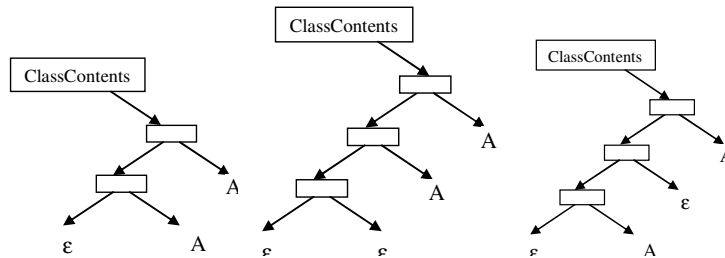


Figure 2: Potential syntax trees for the same ambiguous rule.

The leftmost tree is the one which best represent the rule. However, the two other trees are also possible, and make the grammar ambiguous. This problem is probably a typo, and we suspect that the authors of HUTN actually wanted to write the following rule, which removes all the previously described ambiguities.

```
KeywordAttribute ::= '~'? <AttributeName>
```

3.2 Problem with class contents

The description of the class content is given by the following rule:

```
ClassContents ::= ( ( AttributeInstance | ReferenceInstance |
ContainedObject) ';'? ) *
```

Semantically this rule is well formed; the different possibilities are enumerated as specified in the MOF specification. At first sight this rule is not ambiguous. However, when examining the definition of the three symbols which describe the content of a class, one finds the following rules:

```
AttributeInstance ::= NormalAttribute | KeywordAttribute
NormalAttribute  ::= <AttributeName> (':' | '=') AttributeValue
KeywordAttribute ::= '~'? <AttributeName>

ContainedObject ::=
  (<AssociationName>':')? (ClassInstance | ClassInstanceRef)

ReferenceInstance ::= ContainedReference | NonContReference
ContainedReference ::=
  (<ReferenceName> (':' | '='))? (ClassInstance | ClassInstanceRef)
NonContReference ::= <ReferenceName> (':' | '=') ClassInstanceRef
```

The semantic of the `ReferenceInstance` rule allows the merge of the two ways to define a reference of an instance: either contained, or not contained, in the referring class. The notion of content refers to the notion of entity which cannot exist externally to the referring class. A syntactic ambiguity will arise as described in the following example:

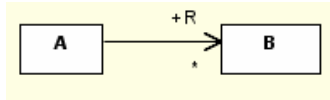


Figure 3: A simple example to illustrate a reference ambiguity.

Say for instance that we build an object instance of A, which refers to a B instance, via the relation with the R role. In HUTN, class A would be defined as follows:

```
A { R : B "b" }
```

When a HUTN parser encounters the text describing the relation R : B "b" it will have to determine the correct representation: is it a ContainedReference or a NonContReference? Unfortunately there is nothing in the syntax to help the parser. Actually, the NonContReference rule happens to be already included in the ContainedReference rule and this is what generates the ambiguity. Therefore the NonContReference can be safely removed (from a syntactic point of view).

The semantic of the ClassContents rule allows merging the two kinds of description for associations and references. Several options are possible to specify by name (or not) the ClassInstance or ClassInstanceRef. When the names are provided there is no ambiguity. However, when either name is not specified, a parser cannot distinguish between a ClassInstance or a ClassInstanceRef. Actually this is because the two unnamed associations or references follow the same syntactic rule. Figure 4 illustrates the problem. The parser can analyze the descriptions for the roles R and A, but not the last (unnamed) which is ambiguous.

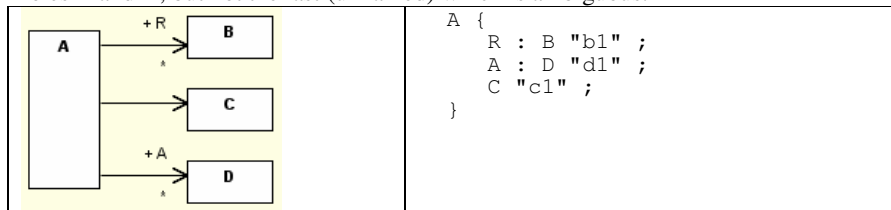


Figure 4: Example of ambiguity when the relation bears no name (or role)

We propose to extract the conflicting (redundant) parts of the two relations, and to factorize it out in the ClassContents rule. Thus the rules ContainedObject and ReferenceInstance are dedicated to the description of instances seen through an association or a reference, explicitly named. After those syntactic transformations, the rewritten grammar rules are:

```
ClassContents ::= ( (AttributeInstance | ReferenceNamedInstance |
ContainedNamedObject | ClassInstance | ClassInstanceRef) ';' '?' ) *

ContainedNamedObject ::= <AssociationName> ':'
(ClassInstance | ClassInstanceRef)

ReferenceNamedInstance ::= <ReferenceName> (':' | '=')
(ClassInstance | ClassInstanceRef)
```

3.3 Problem with class references

A class reference is defined as follows:

```
ClassRefString ::= PackageRootRef | DocumentRootRef
PackageRootRef ::=
  ClassRefSeparator TextualValue (ClassRefSeparator TextualValue) *
DocumentRootRef ::= ClassRefSeparator
  ClassRefSeparator TextualValue (ClassRefSeparator TextualValue) *
ClassRefSeparator ::= ':' | '.' | '/'
```

This formulation is difficult to decrypt, as it is based on the merge of two concepts (`PackageRootRef` and `DocumentRootRef`) which differ only by an initial `ClassRefSeparator`. Having separated these two descriptions complicates a lot the analysis. A simpler, more compact writing may be:

```
ClassRefString ::= ClassRefSeparator?
  ClassRefSeparator TextualValue (ClassRefSeparator TextualValue) *
```

Now, the analysis of the rule is straightforward with EBNF parsers, and requires a simple transformation with those accepting only BNF.

3.4 Problem with class instance references

A class instance reference is defined as follows:

```
ClassInstanceRef ::= <ClassName> ? ClassRefString | ExternalObjectRef
```

with

```
ClassRefString ::= ClassRefSeparator? ClassRefSeparator TextualValue
  (ClassRefSeparator TextualValue) *
```

`ClassInstanceRef` introduces an optional operator, by which an instance reference may follow its class name. This operator generates two ambiguities because `ClassInstanceRef` can be used in collections such as `ClassContents` or `MultiValueList`. A class content may be terminated by an optional `;` symbol, while multi value list may be separated by an optional `,` symbol.

As `ClassInstanceRef` starts with `ClassRefSeparator` and always terminates with a pair `ClassRefSeparator TextualValue`. However, there is an ambiguity related to the optional symbols discussed in the previous paragraph. When a `ClassRefSeparator` is parsed it is not possible to distinguish between the current `ClassRefString` and a new `ClassInstanceRef`.

In the HUTN example given below, the `A` class instance seems to be composing two instances `b` and `c`, defined respectively in the root of the document (instance `b` in package `p`) and in the current package (`c` in the current package). However, a parser might also have considered that `A` is constituted of only one `c` instance, itself defined in the `b` instance in the `p` package.

```
A {
  //p/b
  /c
}
```

When parsing the `\c'`, it can be decided to consider `\'` either as the continuation of a list describing the first instance, or as a symbol starting a second instance. It is not possible to resolve this ambiguity (which looks like the well-known 'dangling else' problem in programming languages) without changing the grammar, unless trusting the heuristics followed by the parser which would favor the continuation of the current description. Changing the grammar can be done by requiring a mandatory `<ClassName>` in the `ClassInstanceRef` description.

3.5 Problem of the simple values

The following rules describe the set of simple values, and call for several comments:

```
DataValue      ::= SingleValueData | MultiValueData
SingleValueData ::= TextualValue
                | NumericValue
                | EnumValue
                | BooleanValue
                | TypeCodeValue
                | StructValue
                | UnionValue
                | ClassInstance
                | ClassInstanceRef
```

`TextualValue` (Characters, strings, wide characters and wide strings) and `TypeCodeValue` are represented both as literal strings. As these two symbols share the same lexical definition there is no reason to introduce two distinct syntactic symbols.

`EnumValue` is represented by a name, and can therefore be considered as being an attribute, thus an adjective describing a class. In that case, `EnumValue` will be identified in the end as a `TextualValue`. Consequently, it will never be seen as a specific symbol of the language.

`StructValue` and `UnionValue` can be represented by a collection of values, syntactically organized like a `MultiValueData`. Thus this rule is not useful as already described by the higher level `DataValue` rule. The rule is rewritten as follows:

```
SingleValueData ::= TextualValue
                | NumericValue
                | BooleanValue
                | ClassInstance
                | ClassInstanceRef
```

The syntactic separation between `TextualValue`, `NumericValue` and `BooleanValue` does not hurt, but is probably not necessary neither, as there is no other HUTN rule which uses them explicitly. Moreover, this distinction probably belongs to the semantic field of the language and is therefore out of the scope of a grammar specification.

3.6 Problem of the class instances

The following rule describes the class instances:

```

ClassInstance
  ::= ClassHeader ParametricAttrs '{' ClassContents '}' ';'?
ClassHeader ::= ClassAdjectives <ClassName> (ClassIdentifier)?
ClassAdjectives ::= ( '~'? <AttributeName> | DataValue ) *
ClassIdentifier ::= TextualValue
ParametricAttrs ::= '(' ValueList ')'
```

This rule generates several different problems. At the syntax level, a `ClassAdjectives` is a collection which can contain `DataValue`. A `DataValue` can contain a `ClassInstance`. A `ClassInstance` can start by the `<ClassName>` keyword. As a consequence, when a `<ClassName>` is parsed, it can be used either as an adjective for a class (yet to come) or as a new class instance definition. In the following HUTN example, the `B` symbol can be either the name of the future `b` instance, or the name of an adjective that would be associated to a `b` instance.

```
A { B b; }
```

This conflict may be resolved if the parser can look ahead after the `B` symbol: as there is no opening curly brace, it may then reject the adjective option. Such resolution however requires parsing capabilities generally not available in simple LL(1) or LALR parsers.

At the semantic level, the `ClassAdjectives` rule allows the qualification of a (yet unknown) class, and thus it is not possible to know the allowed adjectives at the time of parsing. This situation is ambiguous, as within a single model, the same adjective name may correspond to many other features in many different classes. Figure 5 shows an example where `orange` denotes both a `Fruit` and a `Color`.

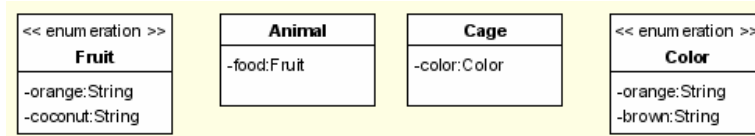


Figure 5: Potential for confusion with class adjectives.

The following HUTN example creates an animal which eats oranges, and an orange cage.

```
{ orange Animal; orange Cage; }
```

However, it is also possible to parse the following HUTN text which creates an animal which eats brown (which makes no sense) and a coconut colored cage (which makes no sense neither), as in the following example:

```
{ brown Animal; coconut Cage; }
```

To detect such errors (at the syntax level) the lexical analyzer should know the classes to which these adjectives apply. A way to achieve this would be to reorganize the `ClassHeader` rule, so that adjectives appear after the class name and the optional identifier. The rule may be rewritten as:

```
ClassHeader ::= <ClassName> (ClassIdentifier)? ClassAdjectives
```

In that case, the HUTN text would be:

```
{ Animal orange; Cage orange; }
```

With this change, the look-ahead syntactic problem described previously would be solved as well. Unfortunately the solution cannot be used as it triggers two new problems. One is related to the `ClassInstance` rule and the other to the optional `ClassIdentifier`. The following HUTN text illustrates the problems:

```
Rectangle rect {
  Point (10, 10);
  Point (100, 100);
}
```

When the parser read the `rect` word it is unable to recognize if it is a `ClassIdentifier` or an adjective as the two symbols are represented by the same textual value. The `ClassInstance` rule below states that a `ClassAdjectives` is followed by `ParametricAttrs`. `ParametricAttrs` is a list of values between parenthesis, while class adjectives can be (among others) bracketed lists delimited by square brackets `[]`, round brackets `()` or angle brackets `< >`. An ambiguity arises as parenthesis may denote either attributes or parameters.

```
ClassInstance
  ::= ClassHeader ParametricAttrs '{' ClassContents '}' ';'?
ParametricAttrs ::= '(' ValueList ')'
```

The `rect` rectangle in the previous example is built by associating two points, respectively (10, 10) and (100, 100). A human would probably understand the parametric description of the two points coordinates; unfortunately a parser would not be able to choose between adjectives or parameters. There is no simple solution to remove these newly generated ambiguities. It will be necessary to rely either on the conflict strategy adopted by a given parser, or to modify the grammar, for instance by forbidding the multi-value attributes to remove the `ClassInstance` conflict. Such a rewriting could be:

```
ClassAdjectives ::= ( ( '~' <AttributeName> ) | SingleValueData) *
```

4 Conclusion

In this paper we have reported on our experience using HUTN as a bridge between ModelWare and GrammarWare. We have motivated the choice of HUTN as a generic concrete syntax for all kinds of languages defined in terms of metamodels. We have presented several ambiguities that we have found in the current version of the HUTN

standard and proposed solutions to remove these ambiguities, so as to reuse existing parsers.

Currently our HUTN parser generator is being integrated with the Kermeta workbench, to produce easily instances of the Kermeta meta-classes, for instance for test purposes of model transformations.

In the future, we intend to add support for the specification of customized concrete syntaxes, much as Xactium is doing.

References

- ¹ HUTN final adopted specification, available from <http://www.omg.org/cgi-bin/doc?formal/2004-08-01>
- ² Muller, P.-A., Studer, P., Fondement, F. and Bezivin, J. Platform independent Web Application Modeling and Development with Netsilon. *Accepted for publication in the Journal on Software and Systems Modelling (SoSym)*. Available from : http://www.sciences.univ-nantes.fr/lina/atl/www/papers/netsilon_sosym.pdf.
- ³ TokTok, the DSTC's implementation of the OMG's HUTN standard, available from <http://www.dstc.edu.au/Research/Projects/Pegamento/TokTok/>
- ⁴ Borrás, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. Centaur: the system. *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments*, 13 (5). 14-24.
- ⁵ Clark, T., Evans, A., Sammut, P. and Willans, J. Applied Metamodelling: A Foundation for Language Driven Development, <http://albini.xactium.com>, 2004.
- ⁶ Greenfield, J., Short, K., Cook, S., Kent, S. and Crupi, J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
- ⁷ D. Akehurst, O. Patrascoiu, *OCL 2.0 - Implementing the Standard for Multiple Metamodels*, OCL 2.0 - Industry standard or scientific playground? Workshop, UML 2003, San Francisco.
- ⁸ Alanen, M., Porres I., *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*. Technical Report 606, TUCS, Mar 2004.
- ⁹ Emfatic, available from : <http://www.alphaworks.ibm.com/tech/emfatic>
- ¹⁰ ATLAS group, INRIA & LINA, KM3: Kernel MetaMetaModel Manual. 2004
- ¹¹ Muller, P.-A., Fleurey F., Jézéquel J.-M., *Weaving Executability into Object-Oriented Meta-Languages*, MODELS 05, Montego Bay.