Enhancing Xtext for General Purpose Languages

Adolfo Sánchez-Barbudo Herrera

Department of Computer Science, University of York, UK.
asbh500@york.ac.uk

Abstract. Xtext is a popular language workbench conceived to support development of tooling (e.g. parsers and editors) for textual languages. Although Xtext offers strong support for source code generation when building tooling for Domain Specific Languages (DSL), the amount of hand-written source code required to give support to complex General Purpose Languages (GPL) is still significant. This research investigates techniques for reducing the amount of hand-written source code for supporting GPLs, via the development of new DSLs from which source code can be automatically generated. In particular, these techniques will be researched in the context of the OCL and QVT Operational languages.

1 Context

This research is contextualized in an Engineering Doctorate (EngD) programme at the University of York. This programme differs from traditional PhD studies in that they are carried out in collaboration with industry. The main interests of this project's industrial sponsor is the Object Management Group (OMG)'s specifications, particularly *OCL* [1] and *QVT* [2]. The sponsor is involved in the official Eclipse [3] projects that provide implementations of these specifications.

2 Motivation

Since 2010, Eclipse OCL has evolved to better align with the OMG OCL standard, while at the same time providing enhancements in the form of high-quality textual editors. These editors have the particular feature that they are mostly automatically generated using the Xtext [4] language workbench. Eclipse QVTo (which implements the Operational QVT standard) is a mature project which relies on Eclipse OCL; the former has not evolved synchronously and in-step with the latter over the last 3 years. As a result, it is not aligned with the new Eclipse OCL implementation. Therefore, there is a need to make the Eclipse QVTo implementation evolve in the same direction as Eclipse OCL, which in turn evolves according to the OMG standard.

When undertaking that alignment, parsers and editors generated by Xtext are desired (e.g. reuse). However, in the Eclipse OCL implementation, there is a significant amount of hand written source code which should be avoided in favour of higher level of abstraction languages from which the source code can be generated. Those languages will ease the provision of the parsers and editors for Eclipse QVTo, which provides enough motivation for this EngD project.

3 Research project scope

The activities to be performed in this research project are conceived to ultimately provide support – in the form of Eclipse-based parsers and editors – for the OMG OCL and QVTo languages; in particular, this will be via the official Eclipse OCL and QVTo projects. As a consequence of technology decisions made earlier in the Eclipse projects, the research project is constrained to the use of the Xtext language workbench.

4 Problem

Xtext is a language workbench which can be used to automatically generate tooling (e.g. parsers and editors) for textual languages. However, whilst Xtext is suitable to fully generate tooling for textual DSLs, it cannot currently be used for a completely automated generation process to cope with more complex GPLs [5]. Instead, a semi-automated approach is used for producing GPL tools, in which Xtext provides automation in the first step, and generated code is complemented by hand-coding to support particular language requirements. A more complete automated generative process from language descriptions, with a higher level of abstraction that reduces the amount of hand-coding, is desirable. The following subsections will explain some of the issues arising when using Xtext for auto-generation of tools for GPLs such as OCL.

4.1 Fixed concrete and abstract syntax

The OMG OCL and QVT specifications separately describe the abstract syntax (AS) and the concrete syntax (CS) of the languages, exposing a gap between them. For example, Figure 1 depicts the *VariableExp* concept from the OCL AS. Figure 2 shows the corresponding CS grammar excerpt.

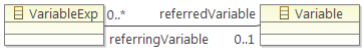

Fig. 1. An OCL VariableExp refers to a Variable (Figure 8.2 from [1]])

The typical approach [6] to bridge this gap is as follows. First, we obtain a syntax tree from a parser, and with further analysis algorithms such a syntax tree is refined. This refinement is declared in OCL by the grammar synthesized attributes showed in Figure 2. However, if this information was moved into an Xtext grammar, e.g., Listing 1.1, two problems can be seen:

```
1  VariableExp :
2    referredVariable=[Variable|simpleNameCS]
3    | referredVariable=[Variable|'self'];
```

Listing 1.1. Potential Xtext grammar rule to obtain a VariableExp

9.4.3 VariableExpCS

A variable expression is just a name that refers to a variable or self.

[A] VariableExpCS ::= simpleNameCS

[B] VariableExpCS ::= 'self'

Abstract syntax mapping

VariableExpCS.ast : VariableExpression

Synthesized attributes

[A] VariableExpCS.ast.referredVariable =
env.lookup(simpleNameCS.ast).referredElement.oclAsType(VariableDeclaration)

[B] VariableExpCS.ast.referredVariable =
env.lookup('self').referredElement.oclAsType(VariableDeclaration)

Inherited attributes

-- none

Disambiguating rules

[91][1][A] simpleNameCS must be a name of a visible VariableDeclaration in the current environment

env.lookup (simpleNameCS.ast).referredElement.oclIsKindOf (VariableDeclaration)

Fig. 2. VariableCS attribute grammar excerpt

Firstly, concepts representing both abstract syntax (e.g. VariableExp, line 1) and concrete syntax (e.g. simpleNameCS, line 2) need to be mixed. The essence of separating the abstract syntax from concrete syntax as it is done in the OCL specification would get lost with an Xtext grammar. Secondly, syntax tree refinements (described by the right-hand side of the OCL grammar's synthesised attributes) can't be represented in Xtext. Thus the OCL CS2AS bridge needs to be hand-coded by customizing the generated parser.

4.2 Visibility rules for name analysis

Another challenge when refining syntax trees is name resolution, based on qualified accesses, nested scopes, and inheritance; these constructs are common in OO-derived GPLs. For instance, listing 1.2 depicts a typical name resolution scenario, with respect to the method which can be referred to by a method call expression.

```
1  class A {
2    public void methodA() { // do something }
3  }
4  class B extends A{
5    public void methodB() { // do something }
6    public static void main(String [ ] args) {
7      B b = new B();
8      b.methodA();
9    }
10 }
```

Listing 1.2. Simple name resolution Java example

When calling a class method for a given object in Java, one could choose from a set of methods based on visibility rules, e.g., all public methods defined by the class plus all public methods of every ancestor. Xtext grammars don't provide means to define this kind of rule, which has to be manually encoded.

4.3 Syntax tree rewrites based on semantic analysis.

In languages like OCL, we might find situations in which the final abstract syntax tree can't be obtained with simple context-free grammar syntax rules. Consider the OCL expression *self.aProperty->size()*. The *size()* library operation is normally applied on collections in order to obtain its number of elements. In this case, it is a applied on the value of *aProperty* of a model element. The issue is that it isn't known in advance if the value of *aProperty* is a collection until some semantic analysis is done. In the OCL language, if *aProperty* turned out to be defined as a single-value, there is a syntactic rewrite – an *implicit collection conversion*. This kind of syntax rewrite is described in the OMG OCL specification by Listing 1.3; it can't be expressed in Xtext.

```
1  OperationCallExpCS.ast.source =
2    if OclExpressionCS.ast.type.oclIsKindOf(CollectionType)
3      then OclExpressionCS.ast
4      else OclExpressionCS.ast.withAsSet()
5    endif
```

Listing 1.3. Syntax rewrite example

5 Proposed solutions and expected contributions

Given the issues mentioned previously, we propose improvements to Xtext to provide support for generating tooling for more complex GPLs. Currently, crucial activities such as name resolution and syntax rewrites have to be hand-written by customizing code generated by Xtext[1]. Our idea is to use DSLs which capture the variability of these activities, in order to further automatically generate the corresponding source code. The overall approach is depicted in Figure 3. The following subsections describe the expected contributions.

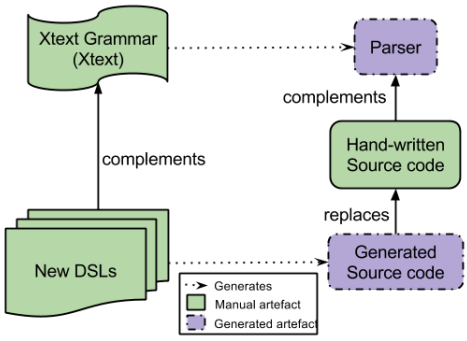

Fig. 3. Generating currently hand-written code from DSLs instances

[1] Xtext has a generated, but naive name resolution strategy

5.1 DSLs to reduce the amount of hand-written artefacts

We will propose a set of DSLs, complementing Xtext grammars, which will allow language engineers to reduce the amount of hand-written artefacts needed to give support to GPLs. These DSLs will be accompanied with the required tooling (e.g. editors, code generators) to automate generation of source code.

5.2 Efficient scheduling of activities

Our proposed enhancements to Xtext are in terms of *bridging CS and AS*, *name resolution* and *syntax rewrites*. These different activities are closely related: the AS graphs will not be completed until name resolution is performed, whereas some names resolutions can't be undertaken until some AS elements are available. Some syntax rewrites are not possible without performing name resolution. So, there is a complex chain of activity dependency.

One of the goals of this research is analysing the dependencies among these activities with the aim of generating an efficient implementation capable of exploiting them for faster AS model retrieval. This area of research will produce a contribution to the field, because related work either addresses these activities in an isolated way, or does not consider the overall dependency issue at all.

6 Related Work

To the best of our knowledge, there is no related work that aims to provide abstract descriptions to complement Xtext grammars so as to generate additional source code for GPLs. There exists some work related to the detailed research activities that may provide good ideas or inspiration to individual problems (though none target Xtext tool generation or grammars).

6.1 NaBL and Stratego

The most relevant related work on name resolution is by Konat et al [7], which introduces NaBL to specify name bindings for a language in terms of namespaces, scopes, site definitions and use definitions. This declarative language comprises a DSL to support name resolution when building AS trees, however it's only conceived to be used by Spoofax [8] language workbench. Syntax rewrites can be specified in Spoofax by using a transformation language called Stratego [8].

6.2 JastAdd and JastEMF

JastAdd [9] is another approach related to our research topics; it provides a language (a flavour of attribute grammar) in which not only are there syntactic rules to initially build AS trees but also language constructs to define inherited and synthesised attributions; these can be used to address name resolution concerns, as well as context-dependent syntax rewrites.

From the name resolution point of view, NaBL provides a more convenient language with a higher level of abstraction. However, JastAdd introduces constructs for syntax rewrites, though we know of no work that analyses dependencies between syntax rewrites and name resolution. JastEMF [10] demonstrates the suitability of using JastAdd to create parsers which produce EMF-based models from textual inputs. Both JastAdd and JastEMF do not yet produce the high-quality textual editors required for this project.

6.3 Gra2Mol

Cánovas et al. [11] propose Gra2Mol to obtain models from source code. Their tool provides the means to facilitate the creation of AS models generated from CS models produced by a parser – in this case by the means of a domain specific transformation language (DSTL). Despite the accepted convenience of DSTLs for this task, a bespoke query language is used which is fragile in scenarios where the structure to be queried changes. For instance, the CS metamodel (the source of a transformation) frequently changes when incrementally building support for a language. Gra2Mol also doesn't generate a textual editor for the target language.

7 Plan for evaluation and validation

The following metrics and evaluation methods are presented which will be used to validate the contributions of the research.

To validate the contribution of the DSLs (and source generators), the **lines of code** will be measured and compared: the Java sources which currently need to be hand written, with respect to the new descriptions based on those DSLs.

To validate the contribution that a dependency analysis might provide, the same test suites will be executed using both implementations and relevant hypothesis testing will be performed to verify that there is a significant improvement in the measured **execution time** results.

8 Preliminary work and current status

8.1 Complete OCL documents

Instead of pursuing the aforementioned DSLs, a more general language will be used as part of the first prototype. In this case, the descriptions are given in the form of Complete OCL documents [1]. The reasons behind this decision are as follows.

- It is not yet clear what expressiveness is needed of a language to represent complex scenarios for name resolution or syntax rewrites. A GPL like OCL, we hypothesise, may provide enough flexibility.
- The source code generated by Xtext is Java. The Eclipse OCL project includes an *OCL2Java* code generator so that complex scenarios described with OCL can be translated to Java.

– One of the goals proposed by the sponsor is producing some parts of the OMG OCL specification [1], which currently uses OCL to specify *CS2AS* descriptions. By using OCL to drive part of the Eclipse OCL implementation, it could be reused to drive part of the OMG OCL specification.

The aforementioned DSLs will need to be investigated. However, instead of directly producing Java code, the source code generators will produce Complete OCL documents. Figure 4 depicts the whole generative process.

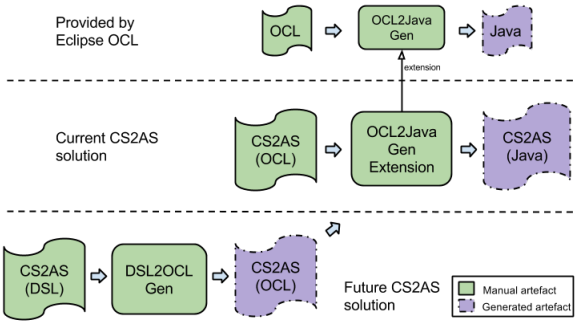

Fig. 4. Leveraging source code generators from a high-level DSL to Java

8.2 CS2AS bridge description

One of the mentioned activities consists of specifying the CS2AS description and generate source code[2] from them. Assuming that Complete OCL documents will be used to describe how CS and AS could be bridged, Listing 1.4 shows an example related to an OCL language concept:

```
1  context VariableExpCS
2  def : ast(env : env::Environment) : ocl::VariableExp =
3    let refVariable : ocl::Variable = env.lookupVariable(self.name)
4    in ocl::VariableExp {
5      referredVariable = refVariable,
6      type = refVariable.type
7    }
```

Listing 1.4. Complete OCL based CS2AS description

8.3 Name resolution description

Some progress has currently been done with respect to how name resolution descriptions will be expressed and Complete OCL documents will also be used

[2] Due to space limitations details about source code generator will be omitted

for this task. They will describe, for instance, how AS elements, with the corresponding name, are contributed to an *environment* so that those elements can be found by name later on. Listing 1.5 shows an example of how a Package contributes named elements (nested packages and types) to the environment.

```
1  context Package
2  def : _env(env : env::Environment) : env::Environment =
3   env.nestedEnv()
4     .addElements(self.nestedPackage)
5     .addElements(self.ownedType)
6  }
```

Listing 1.5. Complete OCL based Name Resolution description

9 Future work

- Finish source code generators from CS2AS and name resolution descriptions.
- Design a solution for syntax rewrites including source code generator.
- Improve generated source code implementation based on dependency analysis of CS2AS, name resolution and syntax rewrites activities.
- Design high level of abstraction DSLs and implement the corresponding generators to produce Complete OCL documents.
- Evaluate the contribution of final solutions as described in this paper.

Acknowledgement. We gratefully acknowledge the support of the EPSRC via the LSCITS initiative, and the sponsor Willink Transformations Ltd.

References

1. OMG. OCL, V2.4. `http://www.omg.org/spec/OCL/2.4`, January 2013.
2. OMG. QVT, V1.2. `http://www.omg.org/spec/QVT/1.2`, May 2014.
3. Eclipse Platform. On-Line: `http://www.eclipse.org/`, 2004.
4. M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *OOPSLA*, 2010.
5. Edward D. Willink. Re-engineering Eclipse MDT/OCL for Xtext. *Electronic Communications of the EASST*, 36:1, 2010.
6. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, & tools*. Pearson Education Inc., 2007.
7. Gabriel Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, volume 7745. Springer Berlin Heidelberg, 2013.
8. Lennart Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*, volume 45, 2010.
9. Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in SE*. Springer, 2006.
10. Christoff Bürger, Sven Karol, and Christian Wende. Applying attribute grammars for metamodel semantics. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, FML '10, pages 1:1–1:5. ACM, 2010.
11. Javier Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Software & Systems Modeling*, 13:1–22, 2012.