A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax

Adolfo Sánchez-Barbudo Herrera[1], Edward D. Willink[2], Richard F. Paige[1]

[1] Department of Computer Science, University of York, UK.
{asbh500, richard.paige}_at_york.ac.uk
[2] Willink Transformations Ltd. ed_at_willink.me.uk

Abstract. Existing language workbenches, such as Xtext, support bridging the gap between the concrete syntax (CS) and abstract syntax (AS) of textual languages. However, the specification artefacts – i.e. grammars – are not sufficiently expressive to completely model the required CS-to-AS mapping, when it requires complex name resolution or multi-way mappings. This paper proposes a new declarative domain specific transformation language (DSTL) which provides support for complex CS-to-AS mappings, including features for name resolution and CS disambiguation. We justify the value of and need for a DSTL, analyse the challenges for using it to support mappings for complex languages such as Object Constraint Language, and demonstrate how it addresses these challenges. We present a comparison between the new DSTL and the state-of-the-art Gra2Mol, including performance data showing a significant improvement in terms of execution time.

Keywords: Concrete Syntax · Abstract Syntax · Domain Specific Transformation Language · Xtext · OCL · Gra2Mol

1 Introduction

One of the challenges that Model-Driven Engineering (MDE) tool implementors face when creating modelling languages is how to effectively bridge the gap between the concrete syntax (CS) and the abstract syntax (AS) of a language: the CS must be designed so that end-users have a familiar and accessible syntax, whereas the AS must be provided behind-the-scenes to enable model management and manipulation – and the two artefacts must be related.

Although this is a general challenge addressed by many works in the field, there are still gaps, particularly for bridging the CS-to-AS (CS2AS) gap for non-trivial modelling languages like the Object Constraint Language (OCL). To understand the aims of this research, we introduce its scope and motivation in the remainder of this section. Section 2 goes deeper into the challenges that arise when specifying CS2AS bridges for languages like OCL. Section 3 introduces the proposed solution to overcome these challenges. Section 4 assesses related work, and we present a more extensive comparative study with Gra2Mol in Sect. 5. We give final remarks and future work in Sect. **??** and conclude in Sect. **??**.

1.1 Scope

Bridging the CS and the AS of a modelling language is a topic with significant related work (discussed in Sect. 4). We focus on the problem for a subset of languages:

- Those whose AS is given in the form of an established (possibly standardised) meta-model. In other words, the end user is interested in editing models conforming to an already existent meta-model.
- Those whose CS is textual and given in the form of a grammar. Although we are aware of previous work [1–3] that supports for textual concrete syntaxes without any grammar provision, they are out of this paper scope.

We use OCL [4] to illustrate the ideas of our approach. OCL has a textual CS and managing instances of it consists of editing models conforming to the language AS (meta-model). The grammar and meta-model come from the specification defined by the Object Management Group (OMG).

1.2 Motivation

To clarify the motivation for our approach, we expose a problem with a specific language workbench: Xtext [5]. Then, we briefly introduce our solution.

Problem. Xtext grammars provide the means to specify bridges between the CS and the AS. However, this can only be done easily for simple languages. Consider the following example of an OCL expression:

```
1  x.y
```

Fig. 1 shows a plausible CS definition. It uses a very simplified OCL grammar and CS with just navigation expressions for ease of presentation.

```
1   NameExpCS:
2    sName=SimpleNameCS
3   ( navOp=('.' | '->')
4     navExp=NameExpCS)?;
5      SimpleNameCS:
6     name=(ID | 'self');
```

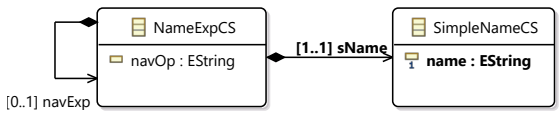

Fig. 1. Example CS definition

In terms of AS (Clause 8 of [4]), we can be sure that *'y'* must be a *PropertyCallExp*. This means, in terms of evaluation (dynamic semantics), that the *'y'* property must be navigated from the object evaluated from the *PropertyCallExp* source (i.e. *'x'*). *'x'* could be a *VariableExp*, whose evaluation uses the value of the *'x'* variable (perhaps defined in an outer *LetExp*). However, in OCL, *'x'* could also be another property navigation using the value of the implicit *'self'* variable. In other words, the original expression could be shorthand for *'self.x.y'*.

```
1     NameExpCS:
2     sName=SimpleNameCS
3     ( navOp=('.' | '->')
4     navExp=NameExpCS)?;
5     SimpleNameCS:
6     name=(ID | 'self');
```

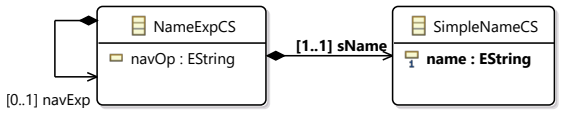

Fig. 2. Here are two figures side-by-side.

This kind of situation cannot be handled by Xtext grammars. Syntactically, it is unknown whether the name *'x'* that precedes the *'.'* operator is a *VariableExp*, or a *PropertyCallExp*. Additional semantic information (static semantics) is required. Despite enhancing EBNF notation [6] to map the AS from the CS, Xtext grammars are insufficient to cope with all the required mappings.

Proposed approach. Given such problematic scenarios, we advocate a clear distinction between the CS specification (i.e a grammar), from which a CS meta-model can be straightforwardly derived (as Xtext does), and the AS specification (i.e a meta-model). Transition from the CS to the AS is then a matter of exercising a model-to-model (M2M) transformation. In particular, we propose a domain specific transformation language (DSTL); our solution entirely operates in the modelware technological space [7].

The reader may note that the approach itself is not novel. The convenience of a CS meta-model has been previously published [8], and, as discussed in our previous work [9], an OCL based informal description is proposed by the own OCL specification. Gra2Mol [10] demonstrates the same idea of a DSTL to map grammars to arbitrary AS meta-models. However, we have identified limitations that have pushed us to come up with a new DSTL, which combines novel features from DSLs like NaBL [11], while offering both declarative capabilities and significant performance improvement (see Sect. 5).

2 Problem Analysis

In this section, we analyse challenges to be addressed when specifying CS2AS bridges for languages like OCL that require non-trivial CS2AS mappings.

2.1 Challenge 1: Significant gap between CS and AS.

Previous work [12, 5] has proposed how meta-models can be mapped from grammars specification. In OCL, there is an AS meta-model which has been established *a priori*; there are substantial differences between the CS and AS. When the mappings between CS and AS elements (e.g. between a grammar non-terminal and a meta-class) are not direct (1-1 mapping), existing approaches

cannot easily establish the desired CS2AS bridges. In general, the possibility to create many AS elements from many CS elements (M-N mappings) is required.

In our introductory example we required either a 2-1 or 2-2 mapping. A *NameExpCS* and a *SimpleNameCS* corresponding to the *'x'* expression, maps either to a *VariableExp* for the *'x'* variable or to a *VariableExp* for the implicit *'self'* variable and a *PropertyCallExp* for the *'x'* property.

2.2 Challenge 2: Cross-references resolution.

When creating AS models, graphs are produced rather than trees. This requires a mechanism to set cross-references at the AS level. For instance, in OCL, the AS elements reference their type. We must therefore specify the computation of these types that may involve identification of a common specialization of template types.

2.3 Challenge 3: Name resolution.

Name resolution is a particular form of cross-referencing where we use CS information such as a name to locate one AS named element in the context of another AS element to resolve a cross-reference between the AS elements. For instance, in our introductory example, *'y'* is used in the context of the *PropertyCallExp* to resolve the reference to the *Property*.

2.4 Challenge 4: Disambiguation resolution.

In the introductory example, we explained how *'x'* might map to either a *VariableExp* for *'x'* or a *VariableExp* and *PropertyCallExp* for *'self.x'*. Syntactically, we cannot determine which AS should be created. Disambiguation rules are therefore required whenever a CS element is ambiguous. CS2AS bridges can specify these CS disambiguation rules as computations involving the CS and/or AS model elements.

3 Solution

We now propose our solution to the aforementioned challenges.

3.1 Distinct CS and AS Meta-Models

The overall approach is depicted in Fig. 3. We advocate introducing distinct CS and AS meta-models. The AS is the established target meta-model ❸. The CS can be an intermediate meta-model ❷ automatically derived from a grammar definition ❶. A potentially complex bridge ❹ between the CS and AS of a language defines mappings between the concepts of the CS and AS meta-models, i.e. defining a model-to-model (M2M) transformation. Existing tools can generate a CS meta-model and the parser ❺ capable of producing the conforming

CS models from a given textual input. In this paper we are concerned with the CS2AS bridge from which we synthesize the M2M transformation solution ❻ that is responsible for consuming CS models in order to produce the final AS ones.

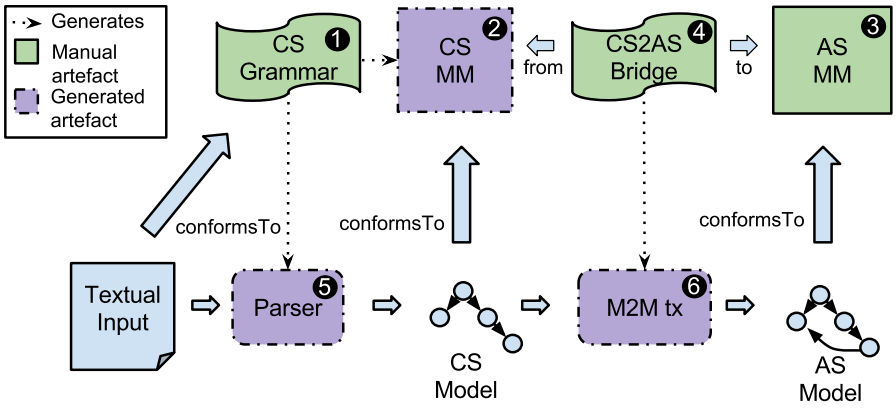

Fig. 3. Overall approach

With the proposed approach we operate in the modelware technological space. The significant parsing concerns do not affect us and so we are not dependent on a particular parser and/or language workbench technology. For example, Xtext (and ANTLR [13] based parsers) are suitable for this approach. More generally, any underlying parser produces CS models conforming to a meta-model could be used. We could therefore use IMP [14] (and LPG [15] based parsers).

3.2 CS2AS External DSTL

We propose a new external DSTL for the CS2AS definition ❹ in Fig. 3. We use a new DSTL rather than an existing M2M transformation language, to provide a more concise declarative language in this domain. The DSTL reuses Essential OCL as the expressions language. The following characteristics led us to define it as a DSTL:

One input and output domain. The model transformations involves just one source input domain and one target output domain. Each domain which may comprise several meta-models. There is no need to support in-place transformations.

Specific name resolution related constructs. We add specific constructs to define name resolution in a declarative manner.

Specific disambiguation rules. The CS disambiguation concern is separated by providing a dedicated declarative section to specify the disambiguation rules that drive AS element construction.

The DSTL consists of four different sections: *helpers*, *mappings*, *name resolution* and *disambiguation*. Each addresses a particular concern of the process of describing CS2AS bridges, and they are introduced below.

```
1  mappings {
2   map PropertyCallExp from NameExpCS
3    when nameRefersAProperty {
4     ownedSource := let parent = self.parentAsNameExp()
5                 in if parent = null
6                  then VariableExp {
7                   referredVariable = trace.lookup(Variable, 'self'); }
8                  else parent.trace;
9                  endif
10    property := trace.lookupFrom(Property, sName, trace.ownedSource.type)
11    type := trace.property.type }}
```

Listing 2. Mappings section excerpt

Helpers. The helpers section provides reusable functionality in the form of helper operations. For instance, Listing 1 depicts a declaration of a helper operation that retrieves the parent element of a *NameExpCS* as another *NameExpCS*. When the parent element is either *null* or a non-*NameExpCS*, *null* is returned.

```
1  helpers {
2   NameExpCS::parentAsNameExp() : NameExpCS[?] :=
3     let container = self.oclContainer()
4     in if container.oclIsKindOf(NameExpCS)
5       then container.oclAsType(NameExpCS)
6       else null endif }
```

Listing 1. Helpers section excerpt

Mappings. The *mappings* section is the main part of the DSTL. The mappings declare how AS outputs are created and initialized from CS inputs. The DSTL includes the basics of declarative M2M transformation languages[16].

Listing 2 depicts an excerpt for our example; we highlight the relevant features. Line 3 refers to a disambiguation rule that is specified in the *disambiguation* section (explained later). Lines 7, 8, 10 and 11 make use of *trace* expressions, which let us access the AS domain from CS elements. Lines 7 and 10 make use of *lookup* expressions to compute name resolution based cross-references (more details later).

The *mappings* section addresses complex CS2AS mappings like that required by our example. The use of OCL supports complex computation and full navigation of the CS and AS models.

Name resolution. The third section of the DSTL specifies how names are resolved. Explaining the full capabilities of the language would merit its own paper. We therefore focus on what is required to explain name resolution in our example: in particular, how a *Property* might be located to resolve the *PropertyCallExp::referredProperty* cross-reference.

```
1  nameresolution {
2   Property {
3    named-element name-property name; }
4   Class {
5    for all-children -- scopes can be configured for all-children elements
6     nested-scope ownedProperties;
7    exports ownedProperties; }}
```

Listing 3. Basic name resolution declaration for *Property* elements lookup.

Listing 3 shows the solution for the simple case. We firstly identify *Property* as a named element, the target of name-based lookups (lines 2-3). Basic unqualified named element lookups are based on the concept of lookup environments (scopes) propagation (Clause 9.4 of [4]). They are detailed in our previous work [9]. In our example, we declare how *Properties* are contributed to lookup scopes. In this case, it is done by the owning *Class* (Lines 5-6). Since a property name might occlude others defined in outer scopes, we use the ***nested-scope*** keyword.

Named elements might be the target of lookups out of the scope of the element that performs the lookup. For instance, a *PropertyCallExp* may refer to a *Property* of a *Class* that is not the *Class* defining the expression's scope. Thus, we also declare that a *Class* **exports** its owned *Properties* (line 7).

Finally, we explain how name-based lookups are linked with the mappings section. In Listing 2, we remarked on two new expressions that enhance OCL: **lookup** expressions (line 7) are used to declare a named element lookup in the current scope. They require the target element type and additional input information (the string *'self'*, in that example); **lookupFrom** (line 10) expressions are used to look up **exported** elements. They require another parameter indicating from which element the lookup is performed (the *type* of the *ownedSource*, in that example).

Disambiguation. The *disambiguation* section of the DSTL declares CS disambiguation rules which can be referred to by mappings declared in the *mappings* section. These disambiguation rules act as a guard for the referring mapping. Listing 4 shows an example of disambiguation rules required by our introductory example.

```
1  disambiguation {
2    NameExpCS {
3      nameRefersAVariable :=
4        let asParent = oclContainer().trace
5        in asParent.lookup(Variable, sName) <> null;
6      nameRefersAProperty :=
7        let csParent = parentAsNameExp(),
8           asParent = oclContainer().trace
9        in if parentNameExpCS = null
10         then asParent.lookup(Property, sName) <> null
11         else asParent.lookupFrom(Property, csParent.trace.type, sName) <> null
12         endif; }}
```

Listing 4. CS disambiguation rules

Our DSTL separates the disambiguation rules from the mappings section. This lets us solve a typical issue in declarative transformation languages where mappings applied to the same input type contain non-exclusive guards (two guards might evaluate to true). For instance, in our example, *'x'* might be both a variable to refer in that particular expression scope, *and* a property of the *'self'* variable. In order to address this issue and keep the mappings section declarative, we enhance the semantics of the disambiguation section so that the order in which the disambiguation rules are defined is significant: the first disambiguation rule that applies for a particular CS element is used. In our example, and providing the mentioned conflict, *'x'* disambiguates to a *VariableExp*, rather than a *PropertyCallExp*, since the *nameRefersAVariable* disambiguation rule is defined first.

3.3 Implementation

The DSTL has been prototyped using Xtext. The corresponding Eclipse plugins are publicly available[3]. The implementation does not include an M2M transformation engine capable to execute instances of the DSTL, rather it contains an Xtend-based [17] code generator[4] that generates a set of Complete OCL files conforming to the OCL-based internal DSL described in our previous work [9]. As explained in [9], the actual CS2AS transformation execution is performed by a generated Java class that uses the Eclipse Modeling Framework and Ecore meta-models to transform CS models to AS models.

4 Related Work

We now discuss how our approach relates to previous work. Space constraints prevent a detailed comparison with the very many tools that provide partial support to the problems identified in this paper, including TEF[18], Spoofax[19] and Monticore[20]. The state-of-the-art related to this research is Gra2mol [10] for which we include a detailed comparative study (Sect. 5). Here, we discuss two particular language workbenches in more detail: Xtext, because it has motivated this research and we aim to integrate with it; and Spoofax, whose NaBL[11] sublanguage has been a source of inspiration of a part of our DSTL.

4.1 Xtext

The introduction mentioned some of the limitations of Xtext; we now relate the challenges from Sect. 2 to Xtext's capabilities.

Challenge 1. Although Xtext grammars provide mechanisms to bridge the CS and AS of a language, as soon as we move away from simple DSLs to those that require M-N mappings, Xtext is insufficient.

Challenge 2. Xtext grammars support name resolution for cross-references in the AS models. They do not support derived resolution such as the types of OCL expressions.

Challenge 3. Xtext grammars resolve names using simple implicit scoping rules. More complex scoping scenarios requires customized code.

Challenge 4. Xtext provides no way to declare CS disambiguation rules.

4.2 Spoofax

Spoofax is a language workbench to give support – e.g parsers, editors – to textual languages. Although it was not originally intended to create models, there is work [21] showing that Spoofax can be used for this purpose. We now relate the challenges from Sect. 2 to Spoofax capabilities.

[3] `https://github.com/adolfosbh/cs2as`

[4] Implementation details about the generator are not included in this paper

Challenge 1. Past Spoofax work [21] to generate meta-models from grammars suffers from the same limitations as Xtext (above). However, Stratego/XT [22] can be used within Spoofax to address this challenge. Building on its foundations, we can define transformations from AST elements (i.e., the CS model) produced by a parser into an AS model.

Challenge 2. Stratego/XT can resolve cross-references in the AS model.

Challenge 3. Spoofax offers a declarative name resolution language (NaBL [11]). However, the name resolution descriptions are only aware of the grammar descriptions (SDF [23]). Cross-references are set when producing the initial AST obtained from the parser. This inhibits cross-references to external AS models – e.g. an AS model with no CS. In the case of OCL, many of the external (meta-)models on which OCL queries operate do not necessarily relate to any textual CS at all.

Challenge 4. Stratego/XT specifies disambiguation rules using *strategy expressions*. There is no convenient way to declare CS disambiguation rules relying on name resolution.

5 Gra2Mol: Comparative Study

We consider Gra2Mol as the-state-of-the-art related to this work. Although it was originally intended as a text-to-model tool for software modernization, their DSTL fits in the same scope and objective we present in this paper. To better assess how our proposed DSTL contributes to the field, we present a comparative study with Gra2Mol. The study consists of a qualitative evaluation in terms of features/capabilities and a quantitative evaluation in terms of performance.

5.1 Qualitative Study

In this section we compare Gra2Mol and our DSTL in terms of their features and capabilities. Due to restricted space, we focus on relevant differences.

Query language. Gra2Mol is based on a tailored structure-shy (like XPath) query language, and our DSTL is based on the statically typed OCL. The Gra2Mol query language is less verbose and more concise than OCL; thus, Gra2Mol instances tend to be smaller. However, Gra2Mol navigation operators are based on accessing children elements. This forces[5] the declaration of deep navigations from the root element, whenever the information is not contained by a given CS element. This leads to performance penalties, because the operators are not as fast as a simple oclContainer() call. Also, the Gra2Mol query language is designed to work strictly on CS models. This has some advantages (e.g., conciseness) compared with our DSTL, because the latter requires usage of *trace* expressions to access the AS domain. However, navigating the AS domain (graphs) from the CS one (trees) provides more concise and/or less expensive

[5] Gra2Mol has a language extension mechanism to introduce new operators, which could be used to improve the default built-in functionality.

navigations to retrieve some particular AS information (e.g. querying the type – a cross-reference – of a particular expression). More importantly, focusing on CS navigations prevents CS2AS transformations from working with external AS models (e.g. a library model with no CS).

Name resolution. Name-based cross-references are declared in Gra2Mol as another model query. These queries are described as direct searches that consider where the target element is located in the model. Model queries get significantly complicated when simulating lookup scopes. In complex languages like OCL, the declarative nature of our *nameresolution* section makes name-based cross-reference declarations concise.

Disambiguation rules. Separating the disambiguation rules away from the mapping declarations provides additional semantics and overcomes a Gra2Mol limitation [10]: "If two or more conforming rules exist, their filter conditions must be exclusive, since only one of them can be applied". This limitation prevents a simple Gra2Mol solution to our introductory 'x.y' example.

Front-end coupling. Our DSTL is not coupled to a parser technology or language workbench. The Gra2Mol transformation interpreter is coupled to a homogeneous CS meta-model they provide, which is incompatible with Xtext grammars; more generally, integrating Gra2Mol with a language workbench like Xtext is impractical.

5.2 Quantitative Study

The quantitative study consists of an experiment based on obtaining execution time measurements for both Gra2Mol and our prototype when executing CS2AS transformations. We focus on execution time because we aim to integrate these CS2AS transformations in textual editors, where too-slow execution time is unacceptable.

Gra2Mol is publicly available with different ready-to-go examples. Our experiment replicates one of them with our prototype and performs a benchmark involving models of different size and/or topology.

Example. We have picked one of the published Gra2Mol examples that is simple enough to fit within our space constraints, that requires cross-references resolution, and provides models of varied topologies.

References

1. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proc. of the 5th International conference*, pages 249–254, New York, NY, USA, 2006. ACM.
2. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-based language engineering with emftext. In *Generative and Transformational Techniques in Software Engineering IV*, pages 322–345. Springer, 2011.
3. Markus Voelter. Language and IDE Modularization and Composition with MPS. In *Generative and transformational techniques in software engineering IV*, pages 383–430. Springer, 2011.

4. Object Management Group. Object Constraint Language (OCL), V2.4. formal/2014-02-03 (http://www.omg.org/spec/OCL/2.4), February 2014.
5. Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference SPLASH '10*, pages 307–309, New York, NY, USA, 2010. ACM.
6. Niklaus Wirth. Extended backus-naur form (ebnf). *ISO/IEC*, 14977:2996, 1996.
7. Jean Bézivin. Model driven engineering: An emerging technical space. In *Generative and Transformational Techniques in Software Enegineering*, pages 36–64. Springer, 2006.
8. Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *Model Driven Engineering Languages and Systems*, pages 98–110. Springer, 2006.
9. Adolfo Sánchez-Barbudo, Edward Willink, and Richard F. Paige. An OCL-based Bridge from Concrete to Abstract Syntax. In Frédéric Tuong et al., editor, *Proceedings of the 15th OCL Workshop*, volume 1512, pages 19–34. CEUR, 2015.
10. Javier Luis Cánovas and Jesús García-Molina. Extracting models from source code in software modernization. *Software & Systems Modeling*, 13:1–22, 2012.
11. Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative Name Binding and Scope Rules. In *SLE'13*, pages 311–331. Springer, 2013.
12. Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In *Satellite Events at the MoDELS 2005 Conference*, pages 159–168. Springer, 2006.
13. Terence Parr. ANTLR. On-Line: http://www.antlr.org/.
14. Philippe Charles, Robert M Fuhrer, Stanley M Sutton Jr, Evelyn Duesterwald, and Jurgen Vinju. Accelerating the creation of customized, language-specific ides in eclipse. In *ACM SIGPLAN Notices*, volume 44, pages 191–206. ACM, 2009.
15. LALR Parser Generator. http://sourceforge.net/projects/lpg/.
16. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.
17. Xtend. On-Line: https://www.eclipse.org/xtend/.
18. Markus Scheidgen. Textual modelling embedded into graphical modelling. In *Model Driven Architecture–Foundations and Applications*, pages 153–168. Springer, 2008.
19. Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.
20. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.
21. Oskar van Rest, Guido Wachsmuth, Jim RH Steel, Jörn Guy Süß, and Eelco Visser. Robust real-time synchronization between textual and graphical editors. In *Proceedings of ICMT'13*, volume 7909, pages 92–107. Springer, 2013.
22. Eelco Visser. Program transformation with stratego/xt. In Christian et. al Lengauer, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer Berlin Heidelberg, 2004.
23. Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of the ACM International Conference OOPSLA '10*, pages 918–932, New York, NY, USA, 2010. ACM.