TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering

Frédéric Jouault Jean Bézivin Ivan Kurtev
ATLAS team, INRIA and LINA
{frederic.jouault,jean.bezivin,ivan.kurtev}@univ-nantes.fr

Abstract

Domain modeling promotes the description of various facets of information systems by a coordinated set of domain-specific languages (DSL). Some of them have visual/graphical and other may have textual concrete syntaxes. Model Driven Engineering (MDE) helps defining the concepts and relations of the domain by the way of metamodel elements. For visual languages, it is necessary to establish links between these concepts and relations on one side and visual symbols on the other side. Similarly, with textual languages it is necessary to establish links between metamodel elements and syntactic structures of the textual DSL. To successfully apply MDE in a wide range of domains we need tools for fast implementation of the expected growing number of DSLs. Regarding the textual syntax of DSLs, we believe that most current proposals for bridging the world of models (MDE) and the world of grammars (Grammarware) are not completely adapted to this need. We propose a generative solution based on a DSL called TCS (Textual Concrete Syntax). Specifications expressed in TCS are used to automatically generate tools for model-to-text and text-to-model transformations. The proposed approach is illustrated by a case study in the definition of a telephony language.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Specialized application languages; D.3.4 [*Processors*]: Code Generation

General Terms Languages

Keywords Model Driven Engineering, DSL, Concrete Syntax

1. Introduction

Domain Specific Languages (DSLs) have some properties that General Purpose Languages (GPLs) like C++, Java, C#, and UML do not have. For instance, with DSLs, domain concepts are directly represented by syntactic constucts. This often enables more concise and precise specifications, which even non-programmer domain experts can understand. Moreover, a sentence expressed in a DSL usually makes use of higher-level constructs (e.g. rules) than an equivalent sentence in a GPL. A DSL may also be designed to enable reasonning about (e.g. proving properties) or optimizing sentences by restricting what the user can do. This is typically not possible with a GPL.

There are, however, issues limiting the usage of DSLs. A major one is the reduced availability of tools for DSLs compared to GPLs. This is emphasized by the fact that several DSLs are typically required where one GPL is enough. A single GPL may indeed be used to build even the most complex systems. But numerous DSLs are necessary to represent the different facets of most systems.

There are several ways to implement DSLs, for example using XML engineering, Model Driven Engineering (MDE), or Grammarware (i.e. grammar-based systems [1]). There is a growing interest in using MDE for this purpose [2, 3, 4]. The different aspects of a DSL are captured by different models: the domain concepts are represented in a metamodel that we call a Domain Definition MetaModel (DDMM); languages like OCL [5] enable the specification of additional well-formedness constraints [6]; model transformation is a possible solution for DSL-to-DSL and even DSL-to-GPL translations; etc. AMMA [7, 4] (ATLAS Model Management Architecture) is an MDE framework, which provides such possibilities in order to build tools for DSLs.

In this work, we consider the concrete syntax facet of DSLs, when it is textual. The objective is to enable translation from text-based DSL sentences to their equivalent model representation, and vice-versa. Such a feature is essential to the development of tools for text-based DSLs.

The text-to-model problem is classically solved by defining a grammar, and then using one of the many available parser generators (e.g. yacc, ANTLR [8]). Model-to-text is generally handled separately by implementing a visitor that serializes its source model into an equivalent textual representation. This requires two separate encodings of the same syntax: grammar and visitor. For model-based DSLs a third non-syntactic specification (i.e. the metamodel) is also required. However, there is a significant redundancy between these elements. For instance, information already available in the metamodel needs to be duplicated in the grammar (e.g. multiplicity of elements). Parse trees then need to be converted into models either by tree walkers (i.e. visitors) or using annotations in the grammar. These are not only tedious to specify but also depend on the chosen parser generator.

Implementing tools for a single GPL in this way is generally not problematic: many GPL tools do not even use parser generators but human-written parsers. It is, however, not always possible to spend that much resources on each DSL. To find a solution to these issues, we explore generative approaches.

We propose in this work to extend AMMA with support for the specification of textual concrete syntaxes. TCS (Textual Concrete Syntax) is a DSL designed for this purpose. It works by providing means to associate syntactic elements (e.g. keywords like `if`, special symbols like `+`) to metamodel elements with little redundancy. Both model-to-text and text-to-model translations can

GPCE '06 October 22-26, 2006, Portland, Oregon.

be performed using a single specification. A grammar can thus be generated from both the metamodel and the TCS model to perform text-to-model translation. Grammar annotations that build the model while parsing can be automatically generated. Model-to-text translation can also be performed with the same information. To this end, a generic interpreter has been defined to traverse the model following the syntactical path specified in TCS. Keywords and symbols are written alongside model information.

TCS contributes a significant capability to AMMA: bridging the modeling and syntax worlds. The concrete syntax of AMMA core languages like KM3 [9] (Kernel MetaMetaModel), ATL [10, 11] (ATLAS Transformation Language), and TCS itself can be implemented with TCS. The concrete syntax of other DSLs can also be specified with TCS. An example of such a DSL is SPL [12] (Session Processing Language), which we use as a case study in this work.

The paper is organized as follows. Section 2 details the problem domain of TCS. Section 3 presents the main concepts of the Textual Concrete Syntax DSL illustrated on SPL. Implementation issues are discussed in Section 4. Section 5 gives related work, and Section 6 concludes.

2. Background

Before presenting the details of the TCS language we give a short overview of the concepts required to understand the rationale behind it. TCS is a DSL that operates in the context of the AMMA framework. It facilitates the conversion between models defined in the AMMA space and their textual representations found in the Grammarware technical space. The concepts of DSL, technical space, and the AMMA architecture are explained below.

2.1 Domain Specific Languages

A DSL is a language designed to solve a delimited set of problems. This contrasts with GPLs that are supposed to be useful for much more generic tasks, crossing multiple application domains. A given DSL provides means for expressing concepts derived from a well-defined and well-scoped domain of interest.

Similarly to GPLs, DSLs have the following properties:

- They usually have a concrete syntax;
- They may also have an abstract syntax;
- They have a semantics, implicitly or explicitly defined.

There are several ways to define these syntaxes and semantics. The most commonly used way for defining the syntax is via grammar-based systems. In contrast, there are multiple semantic specification frameworks but none has been widely established as a standard yet. In the context of MDE we consider a DSL as a set of coordinated models. This is aligned to one of the main principles of MDE: to consider models as unification concept. In the following paragraphs we elaborate on this vision by describing the types of models found in a DSL and their purpose.

Domain Definition Metamodel. As we mentioned, the basic distinction between DSLs and GPLs is based on the relation to a given domain. Programs (sentences) in a DSL represent concrete states of affairs in this domain, i.e. they are models. A conceptualization of the domain is an abstract entity that captures the commonalities among the possible state of affairs. It introduces the basic abstractions of the domain and their mutual relations. Once such an abstract entity is explicitly represented as a model it becomes a metamodel for the models expressed in the DSL. We refer to this metamodel as a Domain Definition MetaModel (DDMM). Since the DDMM is a specification of the domain's conceptualization we may regard it as an ontology [13]. This base ontology plays a central role in the definition of the DSL. For example, a DSL for directed graph manipulation will contain the concepts of nodes and edges, and state that an edge may connect a source node to a target node. Such a DDMM plays the role of the abstract syntax for a DSL.

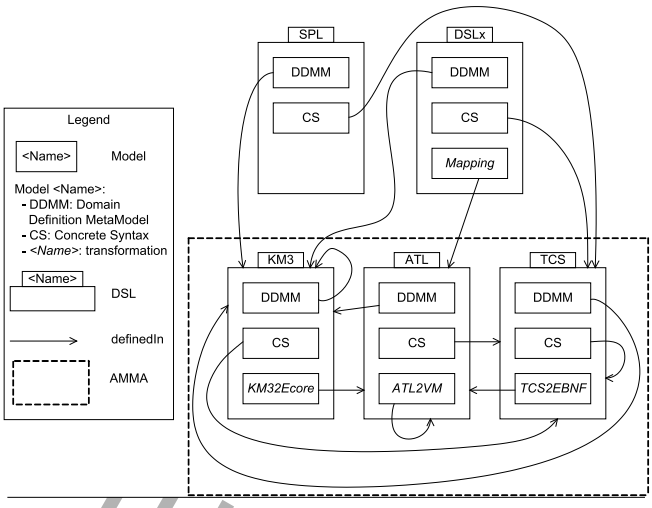

Figure 1. AMMA core DSLs

Concrete Syntax. A DSL may have different concrete syntaxes. A concrete syntax may be defined by a transformation model that maps the DDMM onto a "display surface" metamodel. Examples of display surface metamodels may be SVG [14] or GraphViz [15], but also XML. An example of such a transformation for a Petri net DSL is the mapping from places into circles, from transitions into rectangles, and from place to transition or transition to place relations into arrows. The display surface metamodel will then have the concepts of Circle, Rectangle and Arrow.

Semantics. A DSL may have an execution semantics definition. This semantics definition may also be defined by a transformation model that maps the DDMM onto another DSL having by itself a precise execution semantics or even to a GPL. The firing rules of a Petri net may, for example, be mapped into a Java code model.

In addition to canonical execution, there are plenty of other possible operations on programs based on a given DSL. Each may be defined by a mapping represented as a transformation model. For example, if one wishes to query DSL programs, a standard mapping of the DDMM onto Prolog may be useful.

In the context of MDE there is a need for efficient tools for specification of DSLs. In this paper we use and extend the AMMA modeling architecture that provides tools for defining DSLs. The next section briefly describes the main components of AMMA.

2.2 The AMMA Framework

Similarly to the vision explained in the previous section, DSLs in AMMA are perceived as sets of models. AMMA provides several DSLs that are used to define the components of other DSLs. They form the core of the framework. This core includes a language for describing metamodels called KM3 and a model transformation language called ATL. In this work, we extend the already proposed AMMA structure with TCS in order to specify the textual concrete syntax of DSLs. Figure 1 shows the components of AMMA (including TCS) and how they may be used to define DSLs.

It can be seen that these three DSLs contain models that are expressed in some other DSL from the core. For example, the DDMM of KM3 is defined in KM3. The concrete syntax of KM3 is defined in TCS. Furthermore, KM3 is mapped to the elements of Ecore [16] by using an ATL transformation (the box *KM32Ecore*). The semantics of ATL is defined as a transformation to the language

of the ATL virtual machine (*ATL2VM*) described in [11]. This transformation is itself expressed in ATL.

We can define other DSLs by using the core DSLs of AMMA. For example, the SPL language contains two models. Its DDMM is defined in KM3 and its concrete syntax in TCS. The semantics of the language is not defined since we assumed that it is implemented by already existing tools.

An arbitrary language (denoted as DSLx in Figure 1) can be defined in a similar manner. In the context of DSLx, the box *Mapping* denotes a possible mapping to another DSL or a GPL such as Java.

Currently, AMMA does not provide means for defining semantics of DSLs. The problems of semantics definition of DSLs go beyond the scope of this paper.

We can clearly identify that there already exist technologies that provide the required functionality for specifying various forms of concrete syntaxes. For example, Grammarware provides means for definition of grammars and tools for language manipulation such as parsers and parser generators. Another form of concrete syntax may be based on XML and therefore the tools available in the XML technology should be used.

It is generally more efficient to reuse existing tools for syntax definition instead of inventing/reinventing new ones. This reuse is an example of integration between various technologies: MDE and Grammarware, MDE and XML, etc. A global vision on treating various technologies and their integration in a uniform way is based on the concepts of Technical Space (TS) and projectors between spaces [17]. Before describing the role of TCS as a bridge between MDE and EBNF/Grammarware technical spaces we briefly present the notions of technical space and projector in the next section.

2.3 Technical Spaces and Projections

Technical spaces were introduced in [18], in the discussion on problems of bridging different technologies. This concept was further elaborated in [17] where technical spaces are defined as model management frameworks. The notion of technical space is another important unification concept along with the concept of model. The intention behind it is to denote technologies at a more abstract level in order to allow reasoning about their similarities and differences and possibilities for integration. In this paper we consider two technical spaces: the MDE technical space that allows creation and manipulation of models and the Grammarware technical space that allows definition of language grammars.

An important benefit of treating technical spaces as explicit entities is the recognition of the various capabilities offered by technical spaces and their combination aimed to solve a given problem. To achieve an effective integration towards a certain goal, however, various technologies should interact with each other. An important requirement for such an interaction is the possibility for transferring an artifact from one space to another space and vice versa. This inter-space transfer is called bridging.

Bridging is implemented by transformation utilities called technical projectors. The responsibility to build projectors lies in one reference space. The rationale to define them is quite simple: when one facility is already available in a given space and building it in another space is economically too costly, then the decision may be taken to build a projector that enables the reuse of the facility. There are two kinds of projectors according to the direction of the transformation relative to the chosen reference space: injectors transfer artefacts to the reference space and extractors in the opposite direction.

2.4 Basic KM3 Concepts

TCS works by associating syntactical elements to metamodel elements. All the metamodel examples given in Section 3 are expressed in KM3. TCS semantics is also defined in relation to KM3. We give a brief description of KM3 here that should help understanding the rest of the paper. A more detailed description including formal semantics is given in [9].

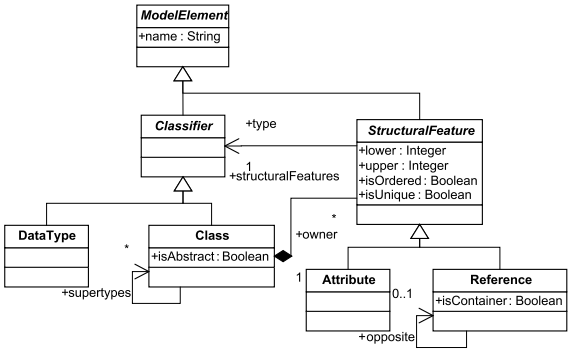

Figure 2. Simplified class diagram of KM3

KM3 is a metametamodel that has concepts similar to those found in MOF [19] but is simpler than MOF. A simplified class diagram illustrating the basic KM3 constructs is shown in Figure 2.

The class *Classifier* denotes concepts that may have instances. It is specialized into *DataType* and *Class*. *Datatypes* have instances that are literal values. *Class* instances have structure that consists of a set of *StructuralFeatures*. By instances of a *Class* we mean here model elements conforming to this class (see [9]). There are two kinds of structural features: attribute and reference. Structural features are typed and have multiplicity. The multiplicity of a feature is encoded by a pair of values called *lower* and *upper*. Classes may extend zero or more other classes and may be abstract. An abstract class cannot have direct instances.

3. TCS: Bridging Metamodels and Grammars

Many of the problems related to textual concrete syntaxes are already solved in the Grammarware technical space. There is no reason to rebuild such facilities in the MDE technical space. What we need is a projector between these spaces. TCS is a language that allows specification and automatic generation of projectors between the Grammarware TS and the MDE TS per given DSL.

This section presents the syntactical constructs of TCS and their semantics based on examples. We start with an overview of the usage of the language and gradually present the syntax going from simpler to more complex features.

3.1 Overview

The overview of the usage of the TCS language is shown in Figure 3. Assume we want to build a DSL called L. In MDE TS we provide a metamodel of L named MM_L expressed in KM3. The definition of the concrete syntax is expressed in TCS and is denoted as CS_L. The required bridge between the two technical spaces consists of an *injector* and an *extractor*. The injector takes a model in L expressed in the textual concrete syntax of L and generates a model conforming to MM_L in the MDE TS. An example model is denoted as SM_L and it conforms to the grammar of L denoted as G_L. G_L is expressed in ANTLR. The extractor generates textual representation of models in the MDE TS conforming to MM_L. Figure 3 shows an example in which a model M_L is extracted to SM_L.

The approach we take starts with the metamodel and the concrete textual syntax description of a given language L. Our goal is to obtain three entities for L: its annotated grammar G_L expressed

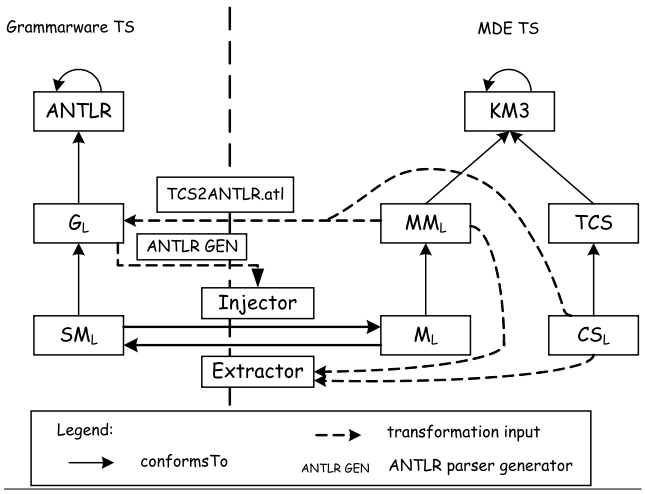

Figure 3. Overview of TCS usage

in ANTLR, and the couple of injector and extractor. G_L is generated by an ATL transformation named *TCS2ANTLR.atl*. It takes MM_L and CS_L as input (shown with dashed lines) and generates the rules and the annotations in G_L. This grammar is used to generate the injector. The injector is a parser generated by the tools provided by the ANTLR technology. The generation is done by the ANTLR parser generator (denoted as *ANTLR GEN*).

The extractor works on the internal representation of models expressed in L and creates their textual representation. It is possible to generate an extractor per every language L. However, we take another approach in which a single extractor is implemented as an interpreter that works for every language. The extractor takes a model M_L written in L, its metamodel MM_L, and its TCS syntax description CS_L and generates the textual representation SM_L of M_L.

Using TCS is typically simpler than developing ad-hoc injectors and extractors. One specification is enough for both directions. Moreover, redundancy between a TCS model and its corresponding metamodel is reduced (e.g. property multiplicity and type are omitted in TCS). With an ideal tool, both the abstract and concrete syntaxes should be specified separately without impacting each other's structure. However, TCS simplification power comes at a certain price: the structural gap between a metamodel and a TCS model is limited. This means that compromises have to be made: either the syntax is adapted to be within TCS possibilities, or the metamodel is simplified.

An important constraint imposed by TCS on metamodels is that they must have a root element. This is roughly equivalent to a start symbol in the corresponding grammar. Other limitations will be presented in Section 4.

3.2 Running Example: SPL

SPL is used as a running example throughout this paper. We start by showing how SPL concrete syntax looks like. Listing 1 shows a simple SPL program that forwards incoming calls to address `sip:phoenix@barbade.enseirb.fr`. The `SimpleForward` service (lines 1-11) declares the target address (line 3) and a registration session (lines 6-10). This session contains an `INVITE` method (lines 6-8) which forwards incoming calls to the declared address (line 7).

Listing 1. Simple SPL program

```
1  service SimpleForward {
2    processing {
3      uri us = 'sip:phoenix@barbade.enseirb.fr';
4
5      registration {
6        response incoming INVITE() {
7          return forward us;
8        }
9      }
10   }
11 }
```

Explanations of how TCS works are illustrated by showing how it can be used to specify the SPL concrete syntax. We give excerpts from the SPL metamodel in KM3, and the corresponding excerpts from the concrete syntax specification in TCS. The metamodel excerpts are necessary because TCS works by annotating this abstract syntax. Only a subset of SPL metamodel and syntax will be given here. The full SPL metamodel and TCS model can be found on the GMT website [20] in the *CPL2SPL* example, which is described in [21].

Let us consider the first metamodel excerpt given in Listing 2. It starts with the declaration of the String data type. Then it specifies that an SPL *Program* (lines 3-5) contains (line 4) exactly one *Service* (lines 7-11). The latter has a name of type *String* (line 8), declarations of type *Declaration* (line 9), and sessions of type *Session* (line 10).

Listing 2. SPL metamodel excerpt in KM3: *Program* and *Service*

```
1  datatype String;
2
3  class Program extends LocatedElement {
4    reference service container : Service;
5  }
6
7  class Service extends LocatedElement {
8    attribute name : String;
9    reference declarations[*] ordered container :
       ↪Declaration;
10   reference sessions[*] ordered container : Session;
11 }
```

Listing 3 gives a TCS model excerpt specifying the concrete syntax of these elements according to Listing 1. Here is an informal description:

- **String.** Data type *String* is represented as an identifier corresponding to lexer non-terminal `NAME` (line 1).
- **Program.** Class *Program* is represented as its contained service (lines 3-5).
- **Service.** Class *Service* is represented as: keyword `service`, the `name` of the service, symbol `{`, keyword `processing`, symbol `{`, the `declarations` of the service, its `sessions`, and two symbols `}` (lines 7-14).

TCS elements are associated to their corresponding metamodel elements by their names. For instance, TCS template *Program* corresponds to KM3 class *Program* and TCS property *service* to KM3 feature *service*. This example shows that it is straightforward to encode such a simple syntax in TCS: syntactic elements are specified in syntax order.

Listing 3. SPL TCS model excerpt: *Program* and *Service*

```
1  primitiveTemplate identifier for String default using
     ↪NAME;
2
3  template Program main
4    : service
5    ;
6
7  template Service  -- context: put this here?
8    : "service" name "{"
9      "processing" "{"
10       declarations
```

```
11          sessions
12        "}"
13      "}"
14    ;
```

A detailed description of the basic TCS constructs used here and of their semantics is given in Section 3.3. Section 3.4 details how a grammar can be derived from these basic constructs. Sections 3.5, 3.6, and 3.7 present more complex TCS constructs.

3.3 Basic Constructs

This section presents the basic TCS constructs. Most of them are illustrated in Listing 3. By default, line number references given in this section refer to this listing.

Each metamodel *Classifier* is associated to a TCS *Template*, which specifies how to textually represent model elements typed by this *Classifier*. There are two main kinds of TCS *Templates*:

- **PrimitiveTemplates** specify the lexer token corresponding to a given metamodel *DataType*, identified by its name. More than one primitive template may be defined for a single data type. This is typically the case for strings: one template represents them as identifiers, whereas a second one represents them as string literals. Exactly one primitive template may be declared as `default` for each data type. Line 1 specifies `default` primitive template `identifier` for data type *String*, which corresponds to lexer token `NAME`.
- **ClassTemplates** specify how classes are represented. This specification consists of a sequence of syntactic elements that are: keywords, special symbols, etc. More information on syntactic elements is given below. A *ClassTemplate* has the same name as its corresponding *Class*. Exactly one class template must be declared as `main` (e.g. line 3 for template *Program*). It corresponds to the root of the model. In contrast to primitive templates, only one class template can be defined for each class in the metamodel. This design choice is aimed at simplifying the TCS specifications. Our experiments have not shown that it is too restrictive.

Syntactic elements are used to represent the contents of a *Class*. They can be of the following kinds:

- **Keywords.** A keyword is a reserved word with specific meaning. In SPL, `service` (line 8) and `processing` (line 9) are keywords. A keyword is specified between double quotes.
- **Special Symbols.** A special symbol is a sequence of characters used as separator or operator (e.g. `{` line 8 and 9). It is specified between double quotes. Each symbol must additionally be listed in the symbols section of the TCS model (not shown here due to space limitations).
- **Properties.** A property corresponds to a metamodel structural feature (i.e. attribute or reference) of the class associated to the contextual template or one of its super classes. It is specified as an identifier, which value is the name of its associated feature. The textual representation of a property depends on its associated feature, especially its type and multiplicity. For simplification we will later directly refer to these as a property's type and multiplicity. Optional property arguments can be specified between curly braces (`{` and `}`). This is detailed below. Identifier `service` at line 4 is a property corresponding to reference `service` of class *Program* (line 2, Listing 2).

As mentioned above, the textual representation of a property depends on its type T. There are two possibilities corresponding to the two main kinds of templates presented above:

- **DataType.** When T is a *DataType*, a primitive template is used. This primitive template is chosen among those associated to T. A specific template may be specified by its name using the `as = <name>` property argument. If no explicit primitive template is specified a default primitive template must be defined for the type and will be used. Property `name` at line 8 is associated to the *String DataType*. Primitive template `identifier` specified at line 1 is therefore used to represent its value.
- **Class.** When T is a *Class*, the class template corresponding to class T is used. Class template `Service` defined at lines 7-14 is thus used to represent property `service` at line 4.

The multiplicity of the property is used to know the number of times the template must be used. A separator to be placed between each use of the template may be specified using the `separator = <separator>` property argument.

3.4 Grammar Generation from Basic Constructs

Following the informal semantics of each TCS construct presented above, a grammar can be generated from a KM3 metamodel and a TCS model. We implemented this translation as the *TCS2ANTLR.atl* ATL transformation, which will be made available on the GMT website [20]. Listing 4 gives the grammar excerpt corresponding to KM3 and TCS excerpts of Listings 2 and 3. It is written using ANTLR [8] version 2 (ANTLRv2) syntax and stripped of the auto-generated annotations. These annotations build the model while parsing but make the grammar less readable. A lexical analyzer (or lexer) is also required. However, we only focus on the parser, and therefore on the grammar here.

Listing 4. Annotation-free SPL grammar excerpt: *Program* and *Service* (ANTLRv2 syntax with highlighted terminals)

```
1  identifier
2    : NAME
3    ;
4
5  program
6    : service
7    ;
8
9  service
10   : "service" identifier LCURLY
11       "processing" LCURLY
12         (declaration (declaration)*)?
13         (session (session)*)?
14       RCURLY
15     RCURLY
16   ;
```

The *TCS2ANTLR.atl* transformation implements a set of declarative translation rules. A full description of these rules is out of the scope of this work. Here is a brief description of the rules used for the generation of Listing 4:

- **PrimitiveTemplate to ProductionRule.** Each primitive template is translated into a production rule containing the corresponding terminal (e.g. lines 1-3). This indirection is used to ease annotation generation. Value conversions (e.g. string to integer) can this way be centralized.
- **ClassTemplate to ProductionRule.** A production rule is created for each class template (e.g. lines 5-7 and 9-16). The name of the rule is the name of the template with a lowercase first letter (ANTLR requirement for non-terminals). The content of the rule is derived from the content of the template: translations of syntactic elements (see the rules below) appear in the same order.
- **Keyword and Special Symbol to Terminal.** Keywords are translated into literal terminals (e.g. `"service"` line 10) and special symbols into non-literal terminals (e.g. `LCURLY` line 10). The non-literal terminals must be defined in the lexer (e.g. `LCURLY: "{";`).

Property kind	Multiplicity		Handling
	lower	upper	
Mono-valued	0	1	Exactly one
	1	1	occurrence
Multi-valued	$0 \leq n \leq m$	$m > 1$	As if $m = *$ (see below)
	0	*	Zero or more
	1	*	One or more
	$1 < n$	*	As if $n = 1$

Table 1. Handling of multiplicities by TCS

Properties are handled differently depending on their multiplicity. Table 1 summarizes how multiplicities are handled by TCS. When the upper bound equals to one we call the property *mono-valued*. When it is greater than one we call the property *multi-valued*. Here are the rules corresponding to both cases:

- **Mono-valued Property to NonTerminal.** A mono-valued property is simply translated into a non-terminal, even when it is optional (i.e. lower bound equals zero) in the metamodel. This design choice is motivated by our belief that optionality should be explicit. Therefore, a mono-valued property can only be made optional by placing it within a conditional constructs (see next section). The non-terminal symbol derived from the property has the name of the non-terminal used for the property type (either clas or a data type).
- **Multi-valued Property to NonTerminals.** Each multi-valued property is translated into a sequence consisting of two non-terminals with the same name. The name of the non-terminals is the same as the non-terminal derived from the property type. The second non-terminal is followed by a repetition construct (i.e. * in ANTLRv2). Properties with fixed upper bound (denoted as m in the table, $m > 1$) are handled as unbounded. This is a simplification, which could be tediously eliminated by using m-times repetition of the same non-terminal plus appropriate grammar constructs. On the base of the experiments we did this simplification does not lead to drawbacks. Separators between elements, when any, are placed just before the non-terminal and inside the repeated block. When the lower bound (denoted as n in the table) is one nothing more is necessary. When it is greater than one, we handle it as if it was one. This second simplification could also be eliminated by expanding as many non-terminals as necessary before the repeated block. When the lower bound is zero an additional optionality construct is appended (i.e. ? in ANTLRv2). This is the case for `declaration` and `session` at lines 12 and 13. Special cases such as in Listing 4 could be simplified (e.g. `(declaration)*` instead of `(declaration (declaration)*)?`) because no separator is specified. However, this is not necessary in practice so we decided not to introduce additional complexity in the transformation rule.

3.5 Additional Constructs

In the previous sections we saw how basic TCS constructs can be used to specify a simple syntax. These basic constructs are, however, not always powerful or convenient enough to handle more complex syntaxes. We describe here some relatively simple TCS constructs, which help overcoming some of basic constructs limitations. Their semantics is briefly outlined. The rules to generate grammar from these constructs are not detailed here because of space constraints:

- **Abstract ClassTemplates** enable the navigation of inheritance hierarchy. For each abstract class template a production rule is generated. It has the form of an alternative of non-terminals corresponding to the subclasses of its associated class. This feature is typically used with abstract classes.
- **Conditionals** are used when the presence of a sequence of syntactic elements in the concrete syntax depends on a condition. A conditional construct specifies a condition, a sequence S_1 of syntactic elements to use when the condition is true, and an optional sequence S_2 to use otherwise. It is always possible to evaluate the condition while serializing a model to text. The condition is moreover specified so that it is reversible: it can be used to set appropriate values in properties while parsing. The condition is a conjunction of simple expressions. These expressions can be:
  - A boolean property, which is set to true if S_1 is recognized, and to false if it is S_2.
  - A comparison between an integer property and a literal value, which can be used to set the property to this value if S_1 is recognized. If S_2 is recognized, it must specify a value for the property.
  - Non-emptiness test for a multi-valued property (syntax: `isDefined(<property>)`, which must be initialized in S_1 and not used in S_2.

  A conditional construct is used in Listing 10 at line 2. A variable declaration is represented by its `type`, followed by its `name`, then an optional `initExp` after an *equals* symbol (i.e. `=`), and ends with a semi colon. The `initExp` is optional and the *equals* symbol should only be there if there is an `initExp`. A conditional construct is used to test if there is an `initExp` and only represent the *equals* symbol and the `initExp` if it is the case. We can see that the design decision described in Section 3.4 to require explicit optionality of properties does not change anything here. Because `initExp` is preceded by an *equals* symbol it must be in a conditional.
- **Operators** can be specified with their priority, associativity (left or right), symbol (e.g. "+"), etc. *OperatorTemplates* may then refer to these operators. An appropriate structure is created in the target grammar. For instance, one rule is created per priority using the rule of higher priority. This works for LL(k) and LALR(1) grammar generators. For LALR(1) grammar generators, operators may also be simply defined with their priorities. The LALR(1) generated parser will then use this information upon shift-reduce conflicts. It is not possible to give more details on this rather complex feature here. *OperatorTemplates* are used in the SPL syntax for arithmetic expressions.

There are other constructs in TCS that are not essential. For instance, there is a construct that enables reusing portions of a TCS specification.

3.6 Symbol Table

The TCS syntactical constructs presented so far enable relatively complex syntax specifications. For instance, the concrete syntax of SPL, KM3, and TCS could mostly be specified in TCS with these constructs only. There is, however, one major limitation: we have only seen how composition references can be represented. With composition references only, models are limited to trees. By using references that cross the nesting/aggregation hierarchy models become graphs. In the remaining part of this section we call this type of references cross-references.

A TCS construct called symbol table makes the usage of cross-references in models possible. The term "symbol table" is borrowed

from the similar concept of symbol table in compilation theory. This feature will be illustrated on SPL variable declaration and usage. We first describe a problem related to cross-references in Section 3.6.1. Then we give two different solutions. We show in Section 3.6.2 how to overcome the problem by making a compromise in the metamodel. Finally, we show in Section 3.6.3 how TCS symbol table handling can be used to provide a better alternative.

3.6.1 Description of the Problem

Let us first consider the SPL metamodel excerpt given in Listing 5. It corresponds to SPL variable declaration and usage. A *Declaration* (lines 1-3) has a `name` (line 2). A *VariableDeclaration* (lines 5-8) is a *Declaration* with a `type` (line 6) and an initialization expression (`initExp`, line 7). A *Variable* (lines 10-12) refers to its *VariableDeclaration* via reference `source` (line 11). This reference is not a composition: its definition in KM3 does not include the `container` keyword.

Listing 5. SPL metamodel excerpt: variable declaration and usage

```
1  abstract class Declaration {
2    attribute name : String;
3  }
4
5  class VariableDeclaration extends Declaration {
6    reference type container : TypeExpression;
7    reference initExp[0-1] container : Expression;
8  }
9
10 class Variable {
11   reference source : Declaration;
12 }
```

Listing 6 gives a first naive encoding of the corresponding concrete syntax. Property `source` (line 6) is specified as if it was a composition reference (i.e. like `type` and `initExp` at line 2). This does not work. The generated grammar illustrates the problem.

Listing 6. Erroneous SPL TCS model excerpt: variable declaration and usage

```
1 template VariableDeclaration
2   : type name (isDefined(initExp) ? "=" initExp) ";"
3   ;
4
5 template Variable
6   : source
7   ;
```

Listing 7 gives the excerpt of the SPL grammar generated from the erroneous specification given in Listing 6. Property `source` has been transformed into a non-terminal corresponding to *VariableDeclaration*. With such a grammar, line 7 of Listing 1 would look like: `return forward uri us = 'sip:phoenix@barbade.enseirb.fr';`. This is incorrect since it should look like: `return forward us;`.

Listing 7. Erroneous ANTLR grammar excerpt for *Variable*

```
1 variable
2   : variableDeclaration
3   ;
```

Actually, the textual representation of a variable is not its declaration but simply an identifier. Listing 8 gives an excerpt of the correct grammar. Property `source` is now represented by the `identifier` non-terminal. This new grammar is a correct representation of the syntax used in Listing 1.

Listing 8. Correct ANTLR grammar excerpt for *Variable*

```
1 variable
2   : identifier
3   ;
```

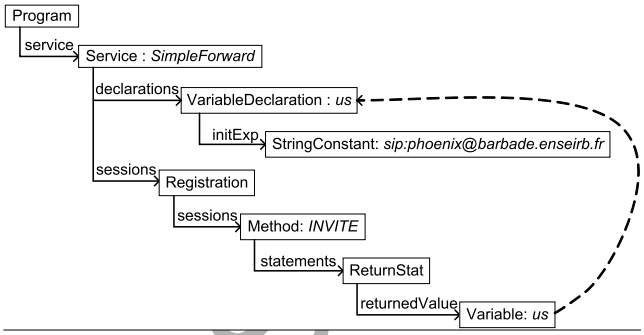

Figure 4. Simplified model corresponding to SPL example

There are two main possibilities to get this result. The first one is to make a compromise in the metamodel by forbidding the use of cross-references. The second one is to use TCS symbol table. We consider both approaches below.

3.6.2 Making a Compromise on the Metamodel

Since cross-references are a problem, let us first try to not use them. Therefore, we replace the `source` cross-reference from *Variable* to *VariableDeclaration* by a simpler name-based reference. Listing 9 gives the corresponding excerpt of SPL metamodel. The only change with respect to Listing 5 is that a *Variable* now simply refer to its *Declaration* by name (attribute `referredVariableName` line 11). This kind of reference does not directly attach a variable to its declaration and is typical in Abstract Syntax Trees (ASTs).

Listing 9. SPL metamodel excerpt: *VariableDeclaration*, tree version

```
1  abstract class Declaration {
2    attribute name : String;
3  }
4
5  class VariableDeclaration extends Declaration {
6    reference type container : TypeExpression;
7    reference initExp[0-1] container : Expression;
8  }
9
10 class Variable {
11   attribute referredVariableName : String;
12 }
```

The corresponding TCS excerpt is given in Listing 10. It only uses constructs presented in previous sections. The corresponding grammar is the correct one, which was given in Listing 8. Property `referredVariableName` is indeed transformed into non-terminal `identifier` because its type is data type `String`.

Listing 10. SPL TCS model excerpt: *VariableDeclaration*, tree version

```
1 template VariableDeclaration
2   : type name (isDefined(initExp) ? "=" initExp) ";"
3   ;
4
5 template Variable
6   : referredVariableName
7   ;
```

Figure 4 gives a simplified representation of the model corresponding to Listing 1. We do not specify the type of the *VariableDeclaration* and of the *Method*, the direction of the *Method*, and other details not relevant here. However, with the solution that we have just considered the dashed arrow does not exist. The model is limited to a tree.

3.6.3 Using TCS Symbol Table Handling

We show how TCS symbol table handling can be used to represent cross-references. The objective is, on one hand, to generate the same grammar as with the previous solution (i.e. Listing 8). On the other hand, the model should be a graph with cross-references instead of simply a tree. The dashed arrow of Figure 4 should be directly represented in the model.

We are going to use the metamodel given in Listing 5, in which a variable points to its declaration via cross-reference `source` (line 11). Listing 11 gives the corresponding TCS model excerpt. Firstly, *VariableDeclaration* has to be put in the current symbol table (`addToContext` keyword on line 1). This means that each time a variable declaration is encountered it is added to the symbol table. Secondly, the representation of property `source` is changed from the naive approach by adding property argument `refersTo = name` (line 6). This means that each time a variable is encountered, its `source` property will be set to the *VariableDeclaration* (type known from the metamodel) having the corresponding `name`. This *VariableDeclaration* will be looked up in the symbol table. The target property of `refersTo` (e.g. `name` here) must be of type *String*.

Listing 11. SPL TCS model excerpt: improved *VariableDeclaration*, graph version

```
1 template VariableDeclaration addToContext
2   : type name (isDefined(initExp) ? "=" initExp) ";"
3   ;
4
5 template Variable
6   : source{refersTo = name}
7   ;
```

The grammars generated from Listings 10 and 11 are identical with the exception of their annotations. This is expected because in both case we have the correct SPL grammar. The difference is in the structure defined by the metamodel: a tree for Listing 10, and a graph for Listing 11. Appropriate annotations get generated from the TCS model of Listing 11 for:

- **VariableDeclaration template** (line 1): a piece of code putting the declaration in the symbol table is added to the generated production rule.
- **Source property** (line 6): a piece of code looking up a declaration is added after the non-terminal corresponding to property `source`. Look up is performed by searching for a declaration, which name corresponds to the identifier of the variable.

The generated parser resolves symbol table references only after having parsed the whole source string. This means that forward references are allowed. References that cannot be resolved (e.g. usage of an undefined variable) or can be resolved to multiple targets (e.g. duplicate declaration of a variable) are reported as errors. Some DSLs may require forward references to be reported as errors too. In this case, an appropriate check should be performed on the model after injecting it.

Actual symbol table handling in TCS is actually a bit more complex but space limitation prevents us to fully describe it here. We only mention an additional feature: there may be several nested symbol tables. Each class template can specify the creation of a new symbol table. This is declared using the `context` keyword in the declaration of a template. Such a feature is used, for instance, to prevent a variable declared in a given method from being used in another. To this aim, template *Method* is declared with the `context` keyword.

3.7 Specific Constructs for Model to Text

A TCS model specifies a concrete syntax for a DSL that can be applied in both text-to-model and model-to-text directions. There are, however, concerns that are specific to the model-to-text direction: coding style concerns and indentations. They also need to be taken into account by TCS models. Coding style does not impact the grammar, only the serialization of blanks (or any other ignored tokens). Additional syntactic elements are provided for serialization support:

- **Block.** TCS blocks provide indentation information. They are delimited by square brackets (i.e. `[` and `]`). By default, each element contained in a block is on a separate line with proper indentation. Each block may additionally have specific arguments. Here are some of them:
  - `nbNL` is used to specify the number of new lines between each element (`nbNL = 1` by default).
  - `indentIncr` is used to specify the number of indentation level that are added to the current level by the block (`indentIncr = 1` by default).

  Listing 12 shows how indentation information can be added to the *Service* class template (originally defined in Listing 3). The block around `declarations` and `sessions` at lines 3-6 specifies that the content of `"processing""{""}"` must be indented. Moreover, two new lines should be inserted between each element. The outer block at lines 2-7 specifies that the content of `"service"name"{""}"` should be indented. The inner block at these same lines specifies that its content should be handled as a single element (i.e. no new line between each of them and no indentation increment). This is to make sure `processing` and `{` are not serialized on two separate lines. With this additional information, proper indentation like in Listing 1 is achieved.
- **Special Symbol Spacing** Each special symbol definition can declare how spaces should be written around it. By default, symbols are neither prefixed nor suffixed with spaces because it is usually not necessary to disambiguate the grammar. `leftSpace` (resp. `rightSpace`) declares that the symbol must be prefixed (resp. suffixed) with a whitespace. `leftNone` (resp. `rightNone`) declares that the symbol must not be prefixed (resp. suffixed) with a whitespace even if the previous (resp. following) symbol declared `rightSpace` (resp. `leftSpace`).
- **Custom Separator.** When none of the above constructs is enough, custom separators may be used. For instance: `<space>` to force the serialization of a space, and `<newline>` to force a line feed.

Listing 12. SPL TCS model excerpt: *Service* with indentation

```
1 template Service context
2   : "service" name "{" [ [
3       "processing" "{" [
4         declarations
5         sessions
6       ] {nbNL = 2} "}"
7     ] {nbNL = 0, indentIncr = 0} ] "}"
8   ;
```

Although no experiment has been conducted in this direction yet, we believe that indentation information specified in TCS could also be used by a text editor to provide automatic indentation.

4. Implementation Issues

First, we briefly mention two features of TCS that are not directly related to the TCS language constructs:

- **Traceability.** The current implementation of TCS provides text-to-model traceability by keeping line and column information in models.
- **Generic Editor.** Textual Generic Editor (TGE) is a tool that partly builds on TCS services. It is available as part of the AM3 project [20]. TGE provides a text editor which is parameterized by information gathered from TCS models. An outline (i.e. tree representation of a program) is generated using TCS text-to-model ability. Hyperlinks and hovers (i.e. automatic display of the target of a link) are provided using text-to-model traceability.

Second, although the TCS tools already enable complex syntax specification, they still have some limitations. We list here some of them and try to provide some hints towards solutions:

- **Error reporting** ranges over two levels. Firstly, errors in TCS and KM3 source models may prevent the correct generation of the target grammar. These errors can typically be expressed as OCL constraints over these source models. Consequently, error checking is implemented in ATL using the solution presented in [6]. Secondly, even when the target grammar is syntactically correct, it may be ambiguous. Non-determinisms reported by the parser generator (ANTLRv2 in our case) are not traced back to corresponding TCS elements. A possible solution to this problem would be to implement traceability between TCS and KM3 models on one hand and the grammar on the other hand. The discussion about grammar class below presents a complementary solution: reducing the number of ambiguities by using a more powerful parser generator.
- **Grammar class** depends on the parser generator that is used. For instance, with ANTLRv2 it is a linear approximation of LL(k). The new version of ANTLR (version 3, or ANTLRv3) is LL(*) [22]. Porting TCS to ANTLRv3 requires to adapt the generated grammar to ANTLRv3 syntax and API, which is used by the generated annotations. This would provide a more powerful tool: fewer grammars are ambiguous in LL(*) than in LL(k). Similarly, TCS could also be ported to other parser generators such as yacc, which is LALR(1).
- **Lexical Analysis** issues are not detailed in this work. Although TCS provides some preliminary support to specify lexers, they still need to be partially specified in ANTLRv2 syntax. Studying common lexer usage should help extend TCS with appropriate constructs.
- **Case insensitive languages** are currently not correctly supported. Two aspects have to be taken into account: keywords and identifiers. Preliminary experiments suggest that this issue should not be difficult to solve
- **Blanks delimited languages** are yet another challenge since they require a close cooperation between lexer and parser. The TCS block construct, which is only used for pretty printing at the moment, could probably be extended for this purpose. Special literals could also be used to represent mandatory blanks. However, we do not anticipate this issue to be easy to solve in the general case.
- **Complex References** between model elements. The current version of TCS only supports simple string-based references such as the variable to variable declaration example presented here. There are more complex scenarios, such as attaching a method call to its corresponding method declaration (e.g. in Java). These cannot be handled by TCS in its present version as they require much more than simple string-based references. A possible solution would be to have a pivot metamodel between the grammar and the desired metamodel. In this pivot (i.e. a syntactical metamodel) all necessary compromises are done. Then, model transformations between both metamodels can be written to resolve complex references. This pivot technique may also be used to overcome other limitations of TCS.

5. Related Work

There exist various solutions to give concrete syntaxes to DSLs. In this section, we focus on DSLs whose abstract syntax is defined as a metamodel and a textual syntax is supplied. Below we comment on some approaches for giving concrete syntax to modeling languages in the context of MDE:

- **XMI.** The Object Management Group (OMG) default model serialization standard is XML Model Interchange [23] (XMI). It is based on XML, which may be considered as a special kind of textual syntax. One of XML advantages is that it can be parsed efficiently without knowing about the DTD or Schema (i.e. metamodel). Another advantage of XMI compared to TCS is that it does not need anything more than the metamodel. This standard specifies rules to automatically derive the corresponding Schema from the metamodel. However, XMI syntax is rather verbose. It is intended for serialization and exchange of models between modeling tools. It is difficult for humans to directly use the XMI syntax for expressing models.
- **HUTN.** The OMG has also specified a standard for serializing models with a non-XML textual syntax. Similarly to TCS, an implementation of Human Usable Textual Notation [24] (HUTN) typically requires a parser generator, which is not the case for XMI. In contrast to TCS, the grammar is automatically generated. An obvious advantage of this approach is that any model can be represented in textual notation at a very low cost. However, HUTN imposes very strict constraints on the notation. Users cannot provide their own syntax customizations. TCS enables user-specified syntax with a greater flexibility than HUTN and therefore the specification of more user-friendly syntaxes.
- **Code generation templates.** Tools like EMF JET [16] (Java Emitter Templates) enable flexible generation of code. This solution is mostly unidirectional (model-to-text) but offers almost total independance between the source metamodel and the target grammar. There need not even be a grammar at all. It is also common to see code generators writen with templates, which also perform a model transformation. For instance, UML to Java code may be performed in one step with this solution. This may be interesting in some cases, but we believe that splitting the model transformation phase and code generation phase is better. For UML to Java code generation we may have an explicit Java metamodel. An UML model is translated to a model conforming to the Java metamodel and then the model is serialized into code. We see at least two advantages of this approach. Firstly, the target language metamodel (e.g. Java) may be reused to compute metrics, refactor code, transform to or from other languages, etc. Secondly, the conceptual mapping between source and target languages (UML and Java) is explicit while in the direct code generation it is hidden in syntax-oriented code.
- **MOF Model to Text.** XMI and HUTN are not suitable for code generation because there is no control on the target syntax. Another OMG standard is consequently being worked on to deal with this issue: Model to Text [25]. The requirements are for unidirectional translation of models to text. The comments and example given above about code generation templates are also true for this solution. Moreover, we also expect that there will soon be another MOF Text to Model standard.

- **Defining a visual concrete syntax.** The work presented in [26] proposes an approach for defining visual syntaxes for modeling languages. It is based on defining a set of mediator classes that relate language metamodel elements and the classes for visual elements (boxes, arrows, etc.). TCS differs from this approach in two major points. TCS aims at textual syntax definition. Instead of using a framework for defining mediator classes we use a DSL for specifying the relations between the metamodel and the grammar.

6. Conclusion

In this paper we presented TCS: a DSL for providing concrete syntaxes to DSLs defined in or with the AMMA framework. The constructs in TCS allows the software engineer to establish correspondences between elements in the language metamodel and their syntactic representation.

Our approach has several benefits. First, the developer is freed from the need to specify a grammar and its annotation in order to generate a parser. Instead she may focus on the syntax templates for language constructs and obtain the annotated grammar automatically. Second, the usage of a language such as TCS leads to a better separation of concerns. The details of the underlying parser generator are hidden from the language designer. This facilitates the replacement of one parser generator system with another. In the current implementation we rely on ANTLR version 2, which uses LL(k) grammars. Switching to another technology should only require a new ATL transformation that generates an annotated grammar for the new tool. TCS could this way benefit from more powerful parser generators such as ANTLR version 3, which uses LL(*) grammars, or other tools e.g. using LALR(1) grammars. Third, TCS specifications enable automatic generation of bidirectional bridges that perform the tasks for text-to-model and model-to-text conversion.

The automation that we pursue comes with paying the price of certain compromises in the abstract and concrete syntaxes. The usage of TCS leads to less freedom in syntax customization compared to an approach in which the grammar is specified by hand and a dedicated parser is developed just for one specific language. However, our goal is to provide a solution for rapid development of concrete syntaxes for DSLs. If the problem at hand is to develop a single, eventually general purpose language then the efforts for developing a dedicated parser are worthwhile. If, however, a large number of DSLs are to be developed quickly then an automated generative solution is a better option.

Apart from the example presented throughout the paper (i.e. SPL) we performed other experiments by applying TCS to the languages found in AMMA: KM3, ATL, and TCS itself. We were able to specify the syntaxes of these languages by using TCS. The result of this experiment is encouraging since it shows that TCS can handle non-trivial concrete syntaxes, such as the syntax of ATL, which uses OCL, without making any critical compromise.

Acknowledgments

We would like to thank Charles Consel and his team who designed the SPL language, which we used to illustrate TCS. This work has been partially supported by ModelWare, IST European project 511731.

References

[1] Kort, J., Klint, P., Klusener, S., Lmmel, R., Verhoef, C., Verhoeven, E.J.: Engineering of Grammarware, `http://www.cs.vu.nl/grammarware/`. (2005)

[2] Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)

[3] GME: The Generic Modeling Environment, Reference site, `http://www.isis.vanderbilt.edu/Projects/gme`. (2006)

[4] Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-based DSL Frameworks. (2006) submitted for publication.

[5] OMG: UML OCL 2.0 Specification, OMG Document ptc/03-10-14, `http://www.omg.org/docs/ptc/03-10-14.pdf`. (2003)

[6] Bézivin, J., Jouault, F.: Using ATL for Checking Models. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia (2005)

[7] Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In Uwe Amann, Mehmet Aksit, A.R., ed.: Proceedings of the European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, LNCS 3599, Springer-Verlag GmbH (2005) 33–46

[8] Parr, T., Quong, R.: ANTLR: A Predicated LL(k) Parser Generator. Software — Practice and Experience **25**(7) (1995) 789–810

[9] Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy. (2006) to appear.

[10] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2006) 128–138

[11] Jouault, F., Kurtev, I.: On the Architectural Alignment of ATL and QVT. In: Proceedings of ACM Symposium on Applied Computing (SAC 06), model transformation track, Dijon, Bourgogne, France (2006)

[12] Burgy, L., Consel, C., Latry, F., Lawall, J., Réveillère, L., Palix, N.: Language Technology for Internet-Telephony Service Creation. In: IEEE International Conference on Communications. (2006) to appear.

[13] Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing. Int. J. Hum.-Comput. Stud. **43**(5-6) (1995) 907–928

[14] Andersson, O., et al.: W3C Working Draft of Scalable Vector Graphics (SVG) 1.2, `http://www.w3.org/TR/SVG12/`. (2005)

[15] Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software — Practice and Experience **30**(11) (2000) 1203–1233

[16] Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Brodsky, S.A., Grose, T.J.: Eclipse Modeling Framework. Addison Wesley (2003)

[17] Bézivin, J., Kurtev, I.: Model-based Technology Integration with the Technical Space Concept. In: Proceedings of the Metainformatics Symposium, Springer-Verlag (2005)

[18] Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: CoopIS, DOA'2002 Federated Conferences, Industrial track. (2002)

[19] OMG: Meta Object Facility (MOF) 2.0 Core Specification, OMG Document formal/2006-01-01, `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`. (2006)

[20] ATLAS team: ATLAS MegaModel Management (AM3) Home page, `http://www.eclipse.org/gmt/am3/`. (2006)

[21] Jouault, F., Bézivin, J., Consel, C., Kurtev, I., Latry, F.: Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In: Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France. (2006) to appear.

[22] Parr, T.: ANTLR v3, `http://antlr.org/v3/index.html`. (2006)

[23] OMG: MOF 2.0 / XMI Mapping Specification, v2.1, OMG Document formal/2005-09-01, `http://www.omg.org/cgi-bin/doc?formal/2005-09-01`. (2005)

[24] OMG: Human-Usable Textual Notation, v1.0, OMG Document formal/2004-08-01, http://www.omg.org/cgi-bin/doc?formal/2004-08-01. (2004)

[25] OMG: MOF Model to Text Transformation Language, http://www.omg.org/cgi-bin/apps/doc?ad/04-04-07.pdf. (2004)

[26] Fondement, F., Baar, T.: Making Metamodels Aware of Concrete Syntax. In Hartman, A., Kreische, D., eds.: Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings. Volume 3748 of Lecture Notes in Computer Science., Springer (2005) 190–204