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 reasoning 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]. In MDE the different aspects of a DSL are captured by different models: the domain concepts are represented in a metamodel; languages like OCL [3] enable the specification of additional well-formedness constraints [4]; model transformation is a possible solution for DSL-to-DSL and even DSL-to-GPL translations; etc. AMMA [2] (ATLAS Model Management Architecture) is an MDE framework, which provides such capabilities 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 [5]). 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 an extension to 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 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 order 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 [6] (Kernel MetaMetaModel), ATL [7] (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 [8] (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 Grammarware. The concept of DSL, 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.

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).

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 [9] or GraphViz [10], but also XML.

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.

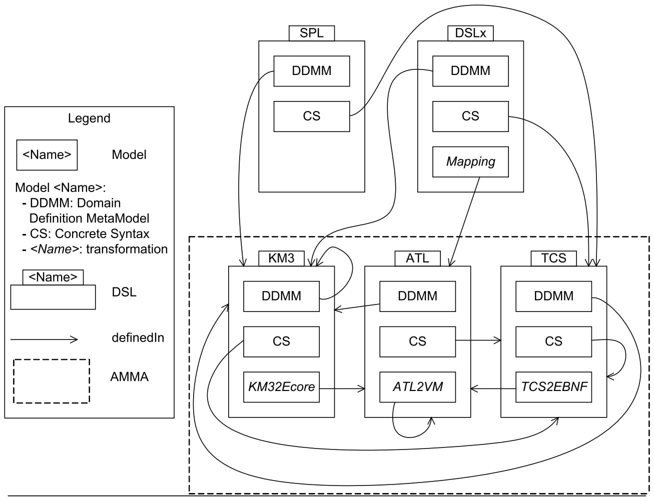

Figure 1. AMMA core DSLs

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

AMMA is built upon the vision described in the previous section. 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 [11] 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*). 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.

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.

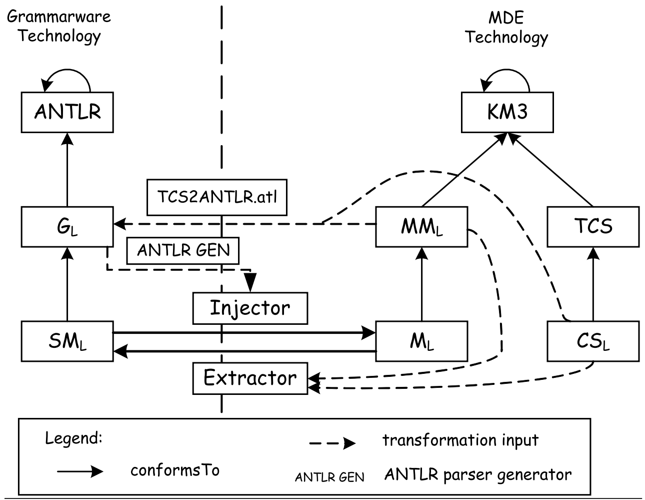

Figure 2. Overview of TCS usage

2.3 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 [6].

KM3 is a metametamodel that has concepts similar to those found in MOF [12] but is simpler than MOF. 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 [6]). 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.

3. TCS: Bridging Metamodels and Grammars

Many of the problems related to textual concrete syntaxes are already solved in the Grammarware technology. There is no reason to rebuild such facilities in AMMA. What we need is a pair of translators between models and their textual representations. TCS is a language that allows specification and automatic generation of these translators between Grammarware and MDE 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 2. Assume we want to build a DSL called L. In AMMA 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 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. 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 conforming to MM_L. Figure 2 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 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 production 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 [13] in the *CPL2SPL* example, which is described in [14].

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*.

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. Sections 3.4, and 3.5 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. 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 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.

- **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.
- **Symbol table** handling enables the use of cross-references. Let us consider, for instance, a variable use such as `us` on line 7 of Listing 1. Without cross-references, the variable expression would simply include the name of the variable without any link to its declaration. Using TCS symbol table handling, a reference from the variable use to the variable declaration is possible. This construct supports multiple environments with nesting.
- **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.

3.5 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.
- **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.

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 [13]. 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 [4]. 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.
- **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(*) [15]. 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).
- **Case insensitive and blank-delimited languages** are currently not correctly supported. Preliminary experiments suggest that the first issue should not be difficult to solve. The second issue requires a close cooperation between lexer and parser, which may not be easy to do in the general case.

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 [16] (XMI). 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 [17] (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 [11] (Java Emitter Templates) enable flexible generation of code. This solution is mostly unidirectional (model-to-text) but offers almost total independence between the source metamodel and the target grammar. There need not even be a grammar at all. Templates are often used to perform a semantic transformation as well as a syntactic pretty printing.
- **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 proposes to deal with this issue: Model to Text [18]. 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.

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 allow 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. 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 also used TCS to specify the concrete syntaxes of AMMA languages: KM3, ATL, and TCS itself. 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 is being partially supported by ModelPlex, IST European project.

References

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

[2] Bézivin, J., Jouault, F., Kurtev, I., Valduriez, P.: Model-based DSL Frameworks. In: Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, OR, USA, ACM (2006) to appear.

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

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

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

[6] 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, LNCS 4037, Bologna, Italy (2006) 171–185

[7] 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

[8] 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)

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

[10] 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

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

[12] 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)

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

[14] 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)

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

[16] 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)

[17] 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)

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