Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages

Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, Fabien Latry

Building DSLs with AMMA/ATL

a Case Study on SPL and CPL Telephony Languages

Frédéric Jouault[1] Jean Bézivin[1] Charles Consel[2] Ivan Kurtev[1] Fabien Latry[2]

[1]ATLAS team, INRIA and LINA [2]Department of Telecommunications, INRIA / LaBRI

{frederic.jouault,jean.bezivin,ivan.kurtev}@univ-nantes.fr {charles.consel,fabien.latry}@labri.fr

Abstract

Domain-Specific Languages (DSLs) enable more concise and readable specifications than General Purpose Languages (GPLs). They are for this reason increasingly used. This DSL approach presents, however, many challenges. One of them is the prototyping and implementation of the numerous DSLs that are required to replace a single GPL. This work presents a case study of implementing two telephony languages: SPL and CPL. It shows how a DSL building framework like AMMA can be used to this purpose.

1. Introduction

Domain-Specific Languages (DSLs) are increasingly used. They directly represent domain concepts at the syntactical level. This enables concise specifications that may even be understood or specified by non-programmer domain experts.

They are, however, many issues that DSL designers encounter. For instance, deriving a DSL from domain knowledge requires more than just placing a domain expert and a language engineer in the same room. Methodologies need to be developed. Some solutions to this first problem may be found in [1].

Another problem is the implementation of these DSLs. Firstly, although a single General Purpose Language (GPL) is often enough to build complex systems, many DSLs are often required for the same task. Each DSL can indeed only capture a limited aspect of a system. Consequently, the development of a DSL cannot be as expensive as for a GPL. Secondly, it may help the DSL designers in their work to implement proof-of-concept prototypes at different stages. We therefore consider that inexpensive development and rapid prototyping are essential for the success of the DSL approach. We tackle here this problem of implementing DSLs.

Our proposal is to use DSL building frameworks [2]. Examples of such frameworks are GME [3, 4] (Generic Modeling Environment), Microsoft DSL Tools [5], and the one we developed: AMMA [2, 6] (ATLAS Model Management Architecture). In this work, we report on an experiment consisting of the implementation of two languages specific to the domain of internet telephony. The first one is SPL [7] (Session Processing Language), and the second one is CPL [8] (Call Processing Language).

The outcome of this experiment provides an interesting example of DSL building. Three aspects of each DSL are taken into account: abstract syntax, concrete syntax, and dynamic semantics. Moreover, our case study allows for different approaches to be illustrated. SPL has a textual concrete syntax whereas CPL is XML-based. Additionally, both languages being in the same domain, one can be defined using the other.

The paper is organized as follows. Section 2 presents AMMA. Sections 3 and 4 respectively describe how SPL and CPL are built. Section 5 concludes.

2. ATLAS Model Management Architecture

This section briefly presents AMMA and three of its core DSLs: KM3 (Kernel MetaMetaModel), ATL (ATLAS Transformation Language), and TCS (Textual Concrete Syntax). A more complete description can be found in [2].

2.1 Overview

AMMA is built on a model-based vision of DSLs, which is presented in [2]. A DSL is considered as a set of coordinated models. Each of these models represents one facet of the language. For instance:

- **Abstract Syntax.** Domain concepts and their relations are captured in a metamodel called a Domain Definition MetaModel (DDMM).
- **Concrete Syntax.** Concrete syntaxes of DSLs can be represented as models. One possibility is to specify a transformation from concrete to abstract syntax. Section 4.3 gives an example of this applied to CPL. Another possibility is to represent a concrete syntax as a model conforming to an EBNF metamodel (i.e. a grammar) or to the TCS metamodel (see below).
- **Semantics.** DSLs have several kinds of semantics that may be captured by models. For instance, dynamic semantics can be captured as an Abstract State Machine [9] model. Alternatively, the semantics of DSL_A may be implemented in terms of the semantics of DSL_B by writing a transformation from DSL_A to DSL_B.

AMMA provides a set of core DSLs that are used to specify each model of a DSL. Figure 1 shows four of these core DSLs: KM3, ATL, TCS, and ASM (Abstract State Machines). Three other DSLs: DSLx, CPL, and SPL are also shown. Each of these DSLs is represented by a set of models defined using AMMA core DSLs. The first three core DSLs are described in the next section. As for the last one (i.e. ASM), please refer to [10], which describes how we extended AMMA with it.

2.2 KM3

The KM3 language is intended to be a lightweight textual metamodel definition language. It enables easy creation and modification of metamodels. The metamodels expressed in KM3 have good readability properties. These metamodels may be easily converted to/from other notations like Emfactic or XMI. KM3 has a clear semantics, partially presented in [11].

Figure 1 shows that the DDMM of KM3 is expressed in KM3 (box *DDMM : KM3*). Its concrete syntax is defined in TCS (box *CS : TCS*). Its semantics is implemented by a transformation to Ecore [12] written in ATL (box *KM32Ecore : ATL*). Other models are not shown here. For instance, a mapping to MOF 1.4 also exists, and formal semantics of KM3 is also expressed in Prolog. A library

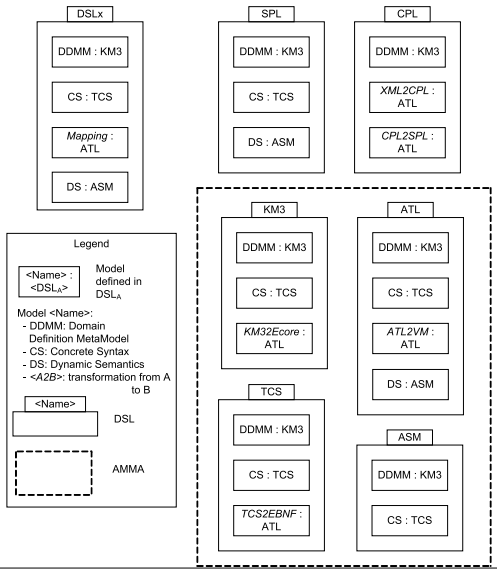

Figure 1. AMMA core DSLs

of KM3 metamodels and their translations to various formats like Ecore, MOF 1.4, and pictures is available on [13].

2.3 ATL

ATL is a hybrid model transformation DSL. Its declarative part enables simple specification of many problems, while its imperative part helps in coping with problems with higher complexity. Informal semantics of ATL is presented in [14] along with a non-trivial case study. More than forty different scenarios accounting for more than a hundred individual transformations are available on [13].

Figure 1 shows that the DDMM of ATL is expressed in KM3 (box *DDMM : KM3*). Its concrete syntax is defined in TCS (box *CS : TCS*). Its semantics is implemented by a transformation to ATL Virtual Machine (see [15]) written in ATL (box *ATL2VM : ATL*). Partial formal semantics of ATL is also expressed in ASM [10] (box *DS : ASM*).

2.4 TCS

TCS is a DSL aimed at specifying context-free textual concrete syntaxes of DSLs. From such specifications, models can be serialized into their textual equivalent and text can be parsed into models. In other words, a TCS specification defines a bidirectional translation between a textual representation of a model and its internal representation. The choice of context-free languages was mainly motivated by the observation that programming languages use them extensively. TCS models provide a way to attach syntactic elements, such as keywords and symbols, to elements of the DDMM of a DSL. A detailed description of TCS is out of the scope of this paper.

Figure 1 shows that the DDMM of TCS is expressed in KM3 (box *DDMM : KM3*). Its concrete syntax is defined in TCS (box *CS : TCS*). Its semantics is implemented by a transformation to EBNF written in ATL (box *TCS2EBNF : ATL*).

3. Session Processing Language

3.1 Overview

SPL programs are used to control telephony agents (e.g. clients, proxies) implementing the SIP (Session Initiation Protocol) [16] protocol. SIP concepts are directly available in the language. Consequently, SPL programs are able to concisely and simply express any telephony service. Additionally, SPL is capable of guaranteeing critical properties that could not be verified with a GPL. SPL programs run on a Service Logic Execution Environment for SIP.

Listing 1 gives a simple example of an SPL service. Every incoming call is redirected to SIP address `sip:phoenix@barbade.enseirb.fr`. The target address is declared on line 3. Lines 6-8 correspond to the definition of the action to perform on incoming calls. The `return` statement at line 7 forwards the call.

Listing 1. Simple SPL program: forwarding a call

```
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 }
```

3.2 Abstract and Concrete Syntaxes

Abstract syntax of SPL is specified in KM3. This corresponds to the box *DDMM : KM3* of the SPL DSL in Figure 1. Listing 2 gives an excerpt of the SPL metamodel. Lines 1-5 define *Service*, which has a name, and can contain declarations and sessions. A *Declaration* (lines 7-9) has a name and may be a *VariableDeclaration* (lines 11-14), which has a type and an optional initialization expression (`initExp`, line 13). A *Session* (line 16) may be a *Registration* (lines 18-20) containing other sessions or a *Method* (lines 22-27). A *Method* has a return type, a direction (see the *Direction* enumeration at lines 35-39), a name, and statements. *Expression*, *TypeExpression*, and *Statement* (lines 29-33) are abstract classes, which are extended to specify the full SPL language (not given here). The full version of this metamodel can be found in AM3 [13] metamodel library.

Listing 2. SPL metamodel excerpt

```
1  class Service {
2    attribute name : String;
3    reference declarations[*] ordered container :
         ↪Declaration;
4    reference sessions[*] ordered container : Session;
5  }
6
7  abstract class Declaration {
8    attribute name : String;
9  }
10
11 class VariableDeclaration extends Declaration {
12   reference type container : TypeExpression;
13   reference initExp[0-1] container : Expression;
14 }
15
16 abstract class Session {}
17
18 class Registration extends Session {
19   reference sessions[*] ordered container : Session;
20 }
21
22 class Method extends Session {
23   reference type container : TypeExpression;
24   attribute direction : Direction;
25   attribute name : String;
26   reference statements[1-*] ordered container : Statement
         ↪;
27 }
28
29 abstract class Expression {}
30
31 abstract class TypeExpression {}
32
33 abstract class Statement {}
```

```
34
35 enumeration Direction {
36   literal inout;
37   literal in;
38   literal out;
39 }
```

Concrete syntax of SPL has been implemented in TCS according to the syntax specified in [17]. This corresponds to the box *CS : TCS* of the SPL DSL in Figure 1. A grammar is automatically derived from both the KM3 metamodel and the TCS model to parse SPL programs into SPL models. SPL models can also be serialized to programs using a TCS interpreter written in Java. We only describe SPL syntax informally here by referring to Listing 1. The *Service* `SimpleForward` spans over lines 1-11. Line 3 contains a *VariableDeclaration* of type `uri` initialized with the address to forward to. Lines 5-10 correspond to a *Registration*, which contains a single *Method* (lines 6-8). This *Method* returns a `response` and is called on incoming (i.e. `Direction::in`) invites.

3.3 Dynamic Semantics

Several definitions of SPL semantics can be given. For instance, static and dynamic (by means of transition rule) semantics for SPL is specified in [17]. We have also developed an executable specification of SPL dynamic semantics using (ASM). This corresponds to the box *DS : ASM* of the SPL DSL in Figure 1. Part of this ASM model has been automatically derived from the SPL KM3 metamodel. This corresponds to the definition of SPL data model. Other parts like environments and state machines are not present in the metamodel. They had to be manually translated from the specification.

4. Call Processing Language

4.1 Overview

CPL is a standard [8] scripting language for the SIP protocol. It offers a limited set of language constructs. CPL is supposed to be simple enough so that it is safe to execute untrusted scripts on public servers. Listing 3 gives a CPL example, which is equivalent to the SPL example given in Listing 1.

Listing 3. Simple CPL script: forwarding a call

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <cpl xmlns="urn:ietf:params:xml:ns:cpl"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="urn:ietf:params:xml:ns:cpl␣cpl.xsd␣
      ↪">
5   <incoming>
6     <location url="sip:phoenix@barbade.enseirb.fr">
7       <proxy/>
8     </location>
9   </incoming>
10 </cpl>
```

Lines 2-4 declare the default XML namespace and its location. The `incoming` element at lines 5-9 declares the actions that have to be performed on incoming calls. The `location` element adds the target address to the current environment. The `proxy` element forwards the call to the adress found in the current environment.

4.2 Abstract Syntax

Abstract syntax of CPL is specified in KM3. This corresponds to the box *DDMM : KM3* of the CPL DSL in Figure 1. Listing 4 gives an excerpt of this metamodel. The complete version is available in the AM3 [13] library. A CPL script is rooted by a *CPL* element (lines 1-3), which contains an *Incoming* (line 9) element. *NodeContainer*s (lines 5-7) like *Incoming* and *Location* (lines 13-15) may contain a *Node* (line 11) like *Action* (line 17). *Proxy* (line 21) is a special kind of *SignallingAction* (line 19) itself a special kind of *Action*.

Listing 4. CPL metamodel excerpt

```
1  class CPL {
2    reference incoming[0-1] container : Incoming;
3  }
4
5  abstract class NodeContainer {
6    reference contents[0-1] container : Node;
7  }
8
9  class Incoming extends NodeContainer {}
10
11 abstract class Node {}
12
13 class Location extends Node, NodeContainer {
14   attribute url : String;
15 }
16
17 abstract class Action extends Node {}
18
19 abstract class SignallingAction extends Action {}
20
21 class Proxy extends SignallingAction {}
```

4.3 Concrete Syntax and Dynamic Semantics

Both CPL concrete syntax and semantics are handled by model transformations in ATL.

CPL concrete syntax is XML-based. TCS is therefore not really useful here. The solution we implemented is the following. We use a generic XML parser to go from the XML document to an XML model conforming to an XML metamodel. This has an extremely low cost since these XML parser and metamodel are provided as part of AMMA. In a second step, we transform our XML model into a CPL model using ATL. This corresponds to the box *XML2CPL : ATL* of the CPL DSL in Figure 1. Listing 5 gives an excerpt of this *XML2CPL* transformation (line 1). It transforms an XML model into a CPL model (line 2) by using a library of XML helpers (line 4) providing the `getElemsByNames` operation on XML elements. A single rule is shown: rule *CPL* (lines 6-17), which transforms the root of the XML document into a CPL element. Nested incoming element is attached to this root (lines 13-15).

Listing 5. XML to CPL transformation excerpt, written in ATL

```
1  module XML2CPL;
2  create OUT : CPL from IN : XML;
3
4  uses XMLHelpers;
5
6  rule CPL {
7    from
8      s : XML!Root (
9        s.name = 'cpl'
10     )
11   to
12     t : CPL!CPL (
13       incoming <- s.getElemsByNames(
14           Sequence {'incoming'}
15         )->first()
16     )
17 }
```

A second transformation (*CPL2SPL*) provides an implementation of CPL semantics by translating CPL concepts into their SPL equivalent concepts. This corresponds to the box *CPL2SPL : ATL* of the CPL DSL in Figure 1. Listing 6 provides an excerpt of this transformation. Line 2 declares source and target models respectively conforming to CPL and SPL. Rule *CPL2Program* (lines 4-19) transforms the root CPL element (lines 5-6) into an SPL program (lines 8-10), an `unnamed` service (lines 11-15) and a dialog (lines 16-18).

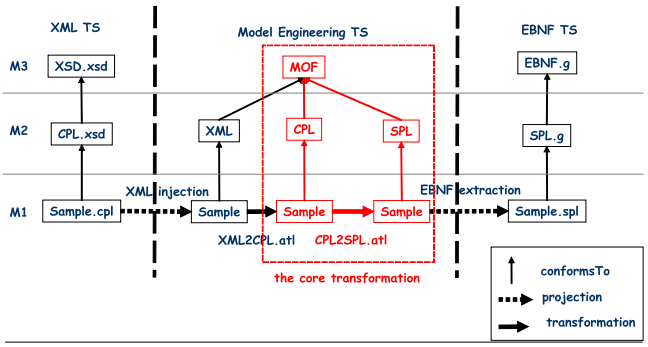

Figure 2. Full CPL to SPL transformation scenario

Listing 6. CPL to SPL transformation excerpt, written in ATL

```
module CPL2SPL;
create OUT : SPL from IN : CPL;

rule CPL2Program {
  from
    s : CPL!CPL
  to
    t : SPL!Program (
      service <- service
    ),
    service : SPL!Service (
      name <- 'unnamed',
      declarations <- s.subActions,
      sessions <- dialog
    ),
    dialog : SPL!Dialog (
      methods <- Sequence {s.incoming, s.outgoing}
    )
}
```

Figure 2 shows the full transformation scenario. The CPL script `Sample.cpl` conforming to the CPL schema is first translated into an XML model conforming to an XML metamodel. Then it is transformed into a CPL model by *XML2CPL.atl*. The core transformation *CPL2SPL.atl* is then applied to generate an SPL model. The latter is then serialized into an SPL program using the TCS interpreter on the TCS syntax definition of SPL. This full transformation scenario (called *CPL2SPL*) is available in the AM3 [13] transformation library.

5. Conclusion

This paper has briefly presented our vision of DSLs as sets of models and its concretization: the AMMA DSL building framework. Details on the implementation with AMMA of two languages (SPL and CPL) specific to the domain of internet telephony have been given. This case study illustrates how AMMA core DSLs can be used to capture different facets of a DSL. KM3 is used to express Domain Definition MetaModels (e.g. of KM3, ATL, TCS, ASM, CPL, and SPL). Concrete syntaxes are defined in TCS, for instance for: KM3, ATL, TCS, ASM, and SPL. Dynamic semantics can be formally defined in ASM, which we have done for ATL and SPL.

Moreover, transformations from any DSL_A to any DSL_B can be implemented in ATL. This can, for instance, be used to implement the semantics of DSL_A in terms of the semantics of DSL_B (e.g. from CPL to SPL, see section 4.3). Such a transformation may then be used to translate programs expressed in DSL_A into programs expressed in DSL_B. Another use of ATL is to implement concrete syntaxes of DSLs (e.g. CPL using *XML2CPL*, see section 4.3).

Acknowledgments

This work has been partially supported by ModelWare, IST European project 511731.

References

[1] Thibault, S., Marlet, R., Consel, C.: Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation. Software Engineering **25**(3) (1999) 363–377

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

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

[4] Karsai, G., Gray, J.: Component Generation Technology for Semantic Tool Integration. In: Proceedings of IEEE Aerospace 2000 Conference, Big Sky, MT, March. (2000)

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

[6] 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

[7] 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.

[8] Lennox, J., Wu, X., Schulzrinne, H.: Call Processing Language (CPL): A Language for User Control of Internet Telephony Services, RFC 3880, `http://www.ietf.org/rfc/rfc3880.txt`. (2004)

[9] Börger, E.: High Level System Design and Analysis using Abstract State Machines. In: FM-Trends 98, Current Trends in Applied Formal Methods. Volume 1641. (1999) 1–43

[10] Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. (2006) submitted for publication.

[11] 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.

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

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

[14] 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

[15] 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)

[16] Rosenberg, J., et al.: SIP: Session Initiation Protocol, RFC 3261, `http://www.ietf.org/rfc/rfc3261.txt`. (2002)

[17] SPL: The Session Processing Language, Reference site, `http://phoenix.labri.fr/software/spl/`. (2006)