A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development

Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, Alfonso Pierantonio

A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development

Davide Di Ruscio [2], Frédéric Jouault[1], Ivan Kurtev[1], Jean Bézivin[1], Alfonso Pierantonio [2]

[1] ATLAS team, INRIA and LINA
France

[2] Dipartimento di Informatica
Università degli Studi di L'Aquila
Italy

— Design, Experimentation, Languages —

RESEARCH REPORT

N° 06.03

April 2006

LINA, Université de Nantes – 2, rue de la Houssinière – BP 92208 – 44322 NANTES CEDEX 3
Tél. : 02 51 12 58 00 – Fax. : 02 51 12 58 12 – http://www.sciences.univ-nantes.fr/lina/

Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, Alfonso Pierantonio

A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development

20 p.

Les rapports de recherche du Laboratoire d'Informatique de Nantes-Atlantique sont disponibles aux formats PostScript® et PDF® à l'URL :
`http://www.sciences.univ-nantes.fr/lina/Vie/RR/rapports.html`

Research reports from the Laboratoire d'Informatique de Nantes-Atlantique are available in PostScript® and PDF® formats at the URL:
`http://www.sciences.univ-nantes.fr/lina/Vie/RR/rapports.html`

A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development

Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, Alfonso Pierantonio

diruscio@di.univaq.it, frederic.jouault@univ-nantes.fr,ivan.kurtev@univ-nantes.fr,jean.bezivin@univ-nantes.fr

Abstract

Domain-Specific Languages (DSLs) are high level languages defined for combining expressivity and simplicity by means of constructs which are close to the problem domain and distant from the intricacies of underlying software implementation constraints. In contrast with general purpose languages, DSLs are typically not useful for generic tasks in multiple application domains. The specification of a DSL is a complex task and requires a lot of knowledge about the domain. In the context of Model Driven Engineering (MDE) metamodeling based techniques are quite commonplace in the syntax specification of DSLs. The definition of their semantics still presents difficulties.
In this paper, a practical experiment is proposed where Abstract State Machines (ASMs) are used as a formal ground for giving, in a precise and unambiguous way, the dynamic semantics of Session Programming Language (SPL), a DSL defined for the development of telephony services over the Session Initiation Protocol (SIP). This experiment is performed in the context of a MDE framework called AMMA (Atlas Model Management Architecture). Although still under development, the approach proposed here illustrates a practical and generic solution to define the precise dynamic semantics of DSLs.

1 Introduction

Specific metamodels able to capture knowledge and concepts of given problem domains play a key role in Model Driven Engineering (MDE). Thus, the degree of success of such an approach strongly depends on the expressiveness and precision of the involved modeling languages and tools. Over the last years, a number of techniques have been proposed for specifying modeling languages as Domain-Specific Languages (DSLs) [28]. Such languages are tailored for a specific problem domain providing concepts and connectives which are familiar to the domain experts who do not usually have any knowledge with general purpose languages (GPLs). Unfortunately, the definition of DSLs suffers from several difficulties, which range from imprecise domain analysis to language under-specification and result in an overall reduced effectiveness. While metamodeling technologies like MOF [24] or EMF [13] have been extensively used for syntax specification and can be considered commonplace nowadays, the definition of semantics is still a challenge and often there is not a consensus for dealing with this issue (see [27] for an interesting discussion on the need of a more popular semantics). Designing languages still remains a difficult task and rigour and formality are unavoidable. Furthermore, the semantic tools being used must provide enough pragmatic qualities such as modularity, extensibility, ease of maintenance of the specifications and programming environment generation to enable the verification of models at earlier stages of language design.

This paper helps defining DSLs by means of MDE techniques. AMMA [9] is a MDE framework based on a set of basic DSLs (e.g. KM3 [1] and ATL [5]) that allows the definition of new DSLs by considering them as a set of coordinated models. These mainly consists of a domain definition metamodel and models describing the syntaxes of the language being defined. In order to cope with the semantics aspects, and not only with most syntactic and transformation issues, AMMA has to be extended with new formalisms.

Some DSLs have semantics describing properties of time evolving systems. We refer to this as dynamic semantics. However, not all DSLs have dynamic semantics. In this paper Abstract State Machines (ASMs) [11] are used for specifying dynamic semantics of DSLs. ASMs have a good combination of formality and pragmatic qualities and they have been used with success for the semantics specification of full scale programming languages. Furthermore, ASMs are executable and several compilers and tools are available both from academy and industry supporting the compilation and simulation of ASMs specifications. The paper proposes a practical experiment by defining the semantics of Session Processing Language (SPL) [14], a DSL designed for implementing telephony services over the SIP protocol [26].

The structure of the paper is as follows. Section 2 provides the basic definitions and describes the proposed approach for the specification of DSLs in a model driven engineering setting. Section 3 briefly reviews the Abstract State Machines formalism, defines its metamodel written in KM3 and the EBNF for specifying the concrete syntax accoring to the XASM dialect [7]. Section 4 proposes the case study where the semantics of SPL is given. After relating the approach with other works, some conclusions are given in Section 6.

2 Domain-Specific Languages and Models

2.1 Background

Domain-Specific Languages (DSLs) are languages able to raise the level of abstraction beyond coding by specifying programs directly using domain concepts [28]. In particular, by means of DSLs the development of systems can be realized by considering only abstractions and knowledge over the considered domain in contrast to general purpose languages, like C++ or Java, that are supposed to be useful for much more generic tasks in multiple application domains. In the former case, the designer does not have to know any programming knowledge, i.e. she/he has not to be aware of implementation intricacies which are distant from the logic and essentials of the system being implemented.

Over the years, many DSLs have been introduced in different application domains (telecommunications, multimedia, database, software architectures, Web management, etc.) each proposing constructs and concepts familiar to experts and professionals working over those domains. Generally, DSL programs are concise, self-documenting and can be reusable, even if the development of a DSL is itself a complex and onerous task. A deep understanding of the domain is required for performing the necessary analysis and to elicitate the requirements the language

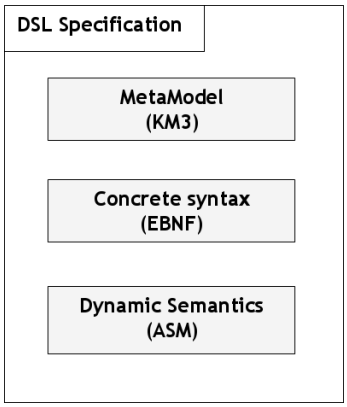

Figure 1: Proposed DSLs specification approach

have to meet. As any other language, a DSL consists of concrete and abstract syntaxes, and possibly a semantics definition which is implicitly or explicitly formulated. Additionally, in MDE a DSL can be viewed as a collection of coordinated models, each of them specifying one of the following aspects:

- *Domain definition metamodel.* The abstract syntax of a DSL is given by means of a Domain Definition MetaModel (DDMM) which introduces the basic entities of the domain and their mutual relations. These descriptions play a central role in the definition of the DSL, for example a DSL for Petri nets will contain the concepts of places, transitions and arcs. Furthermore, the metamodel should state that arcs are only between places and transitions;

- *Concrete syntaxes.* A DSL may have different concrete syntaxes which are defined by a transformation model mapping the DDMM onto a display surface metamodel. Instances of display surface metamodels are SVG or DOT [18], but also XML. An example of such a transformation for a Petri net DSL is the mapping from places to circles, from transitions to rectangles and from arcs to arrows. The display surface metamodel will then have the concepts of Circle, Rectangle and Arrow;

- *Dynamic semantics.* Many DSLs have a dynamic semantics based on the notion of transitions from state to state that happen in time. Dynamic semantics may be given in multiple ways, for example, by mapping to another DSL having itself a dynamic semantics or even by means of a GPL. In this paper we focus on DSLs with dynamic semantics;

- *Additional operations over DSLs.* In addition to canonical execution, there are plenty of other possible operations manipulating programs written in a given DSL. Each may be defined by a similar mapping represented by a transformation model. For example if one wishes to query DSL programs, a standard mapping of the DDMM onto Prolog may be useful. The study of these operations over DSLs presents many challenges and it is an open research subject.

Taking into account the mentioned aspects, the rest of the section introduces the proposed approach for the definition of DSLs by paying more attention to how to specify their dynamic semantics.

2.2 Proposed approach to DSLs specification

As claimed above, designing a DSL is a difficult task as different aspects have to be taken into account. According to the proposed approach, Figure 1 depicts the different parts composing a DSL specification. A number of modeling languages are involved for this purpose and each of them can be considered, in turn, a DSL. The description of the basic entities of the domain and their relations are provided by using KM3 [1], a metamodeling language

which presents advantages over MOF and Ecore especially regarding usability and minimality. In fact, only essential concepts, such as Class, Attribute and Reference, are available in KM3. The concrete syntax is described by models written in EBNF which is a DSL able to bridge the abstract syntax to the textual one that is going to be used by the user of the language. Finally, the specification of the dynamic semantics has to be provided too. Unfortunately, there is not a generally accepted formalism or technique for doing it as over the last decades several semantics have been proposed but none emerged as universal and commonplace, as for instance happened to the EBNF for context-free syntaxes.

Since we are interested in language design (rather than in analyzing or in verifying language properties, e.g. whether a type-system is safe) our attention is devoted towards those mathematical formalisms which present enough pragmatic qualities and allow the designer to convey her/his design decisions into documents being still able to backtrack, modularize, enhance specifications. In this respect, Abstract State Machines [11] have been extensively used to give semantics to full scale languages, such as C [20], C++ [29], Java [12], Oberon [22] and Prolog [10], to mention a few. In this paper we proposes ASMs as a framewrok for the dynamic semantics specification of DSLs.

3 Abstract State Machines

3.1 Overview

ASMs [11] bridge the gap between specification and computation by providing more versatile Turing-complete machines. The ability to simulate arbitrary algorithms on their natural levels of abstraction, without implementing them, makes ASMs appropriate for high-level system design and analysis. ASMs specifications represents a formal basis to reason about the properties of systems which are described into unambiguous way. ASMs form a variant of first-order logic with equality, where the fundamental concept is that functions are defined over a set $\mathcal{U}$ and can be changed point-wise by means of transition rules. The set $\mathcal{U}$, referred to as the *superuniverse* in ASM terminology, always contains the distinct elements *true*, *false*, and *undef*. Apart from these, $\mathcal{U}$ can contain numbers, strings, and possibly anything, depending on the application domain.

By means of ASMs, systems can be modeled as sequences of state transitions. The state transitions are captured by means of ASMs rules that are executed if corresponding predicates are verified. Being slightly more formal, we define the *state* λ of a system as a mapping from a signature Σ (which is a collection of function symbols) to actual functions. We write f_λ for denoting the function which interprets the symbol f in the state λ. Subsets of $\mathcal{U}$, called universes, are modeled by unary functions from $\mathcal{U}$ to $\{$*true*, *false*$\}$. Such a function returns *true* for all elements belonging to the universe, and *false* otherwise. A function f from a universe U to a universe V is a unary operation on the superuniverse such that for all $a \in U$, $f(a) \in V$ or $f(a) = undef$. The universe *Boolean* consists of *true* and *false*. A basic ASM *transition rule* is of the form

$$f(t_1, \ldots, t_n) := t_0$$

where $f(t_1, \ldots, t_n)$ and t_0 are closed terms (i.e. terms containing no free variables) in the signature Σ. The semantics of such a rule is : evaluate all the terms in the given state, and update the function corresponding to f at the value of the tuple resulting of evaluating $(t_1, \ldots, t_n)$ to the value obtained by evaluating t_0. Rules are composed in a parallel fashion, so the corresponding updates are all executed at once. Apart from the basic transition rule shown above, there also exist *conditional* rules where the firing depends on the evaluated boolean condition-term, *do-for-all* rules which allow the firing of the same rule for all the elements of a universe, and lastly *extend* rules which are used for introducing new elements into a universe. Transition rules are recursively built up from these rules.

ASMs have been used with success in numerous applications and also for specifying the semantics of different languages (like C, Java, SDL, VHDL) [3]. Additionally, ASMs are executable and several compilers and tools are available both from academy and industry supporting the compilation and simulation of ASMs specification. For such reasons we have chosen to use ASMs in our approach as a formal framework for the specification of the dynamic semantics of DSLs. Furthermore, in the sequel of the paper, ASMs rules are given in the XASM [7] dialect compiler.

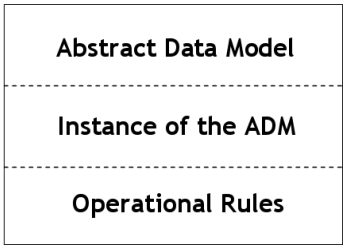

Figure 2: General Structure of the abstract machine giving semantics to a DSL

Giving dynamic semantics to a DSL with ASMs consists of the specification of an abstract machine that is able to simulate models defined by means of the given DSL. The machine has to be generic enough to express the behavior of all correct models. As depicted in Figure 2 the ASMs specification of such a machine is composed of the following parts:

- *Abstract Data Model (ADM).* It describes the constructs of the language and all the additional elements, language dependent, that are necessary for modeling dynamics (like environments, states, configurations etc.). In particular, ADM consists of universes and functions. Generally, for imperative DSLs the static part should be automatically generated from the metamodel of the language. Anyway, the obtained abstract data model has to be refined and extended with the elements concerning the dynamic part specification on which the operational semantics of the language is based;
- *Instance of the ADM.* It encodes the model that has been defined with the given DSL and that we want to verify. The encoding is based on the abstract data model previously described and it gives the initial state of the abstract machine. Such an encoding should be automatically obtained by means of model transformation. Starting from a model conforming to the metamodel of the considered DSL, an equivalent model conforming to the ASMs metamodel and based on the ADM can be generated;
- *Operational Rules.* The meaning of the models defined with the given DSL is specified by means of operational rules expressed in form of transition rules. They are opportunely fired starting from the given instance of the ADM, and they modify the dynamic elements language dependent like environment, state etc. The evolution of such elements gives the model dynamic semantics and simulate its behavior.

An example describing the definition of DSLs by using the proposed approach is provided in Section 4 where main attention is payed to the description of the dynamic semantics. Essentially, the ASMs specification based on the structure depicted in Figure 2 will be presented for the Session Processing Language (SPL) DSL.

3.2 KM3 metamodel of ASMs

Before the example, some details have to be provided. In particular, in order to bring Abstract State Machines in a Model Driven Engineering setting, where everything is a model, an ASMs metamodel has to be defined. This

asm $A(a_1 : T_1, \ldots, a_n : T_n) \rightarrow a_0 : T_0$
⟨asm meta information⟩
is
⟨universe, function, and subasm declarations⟩
⟨initialization rules⟩

⟨asm rules⟩
endasm

Figure 3: General Structure of an XASM machine

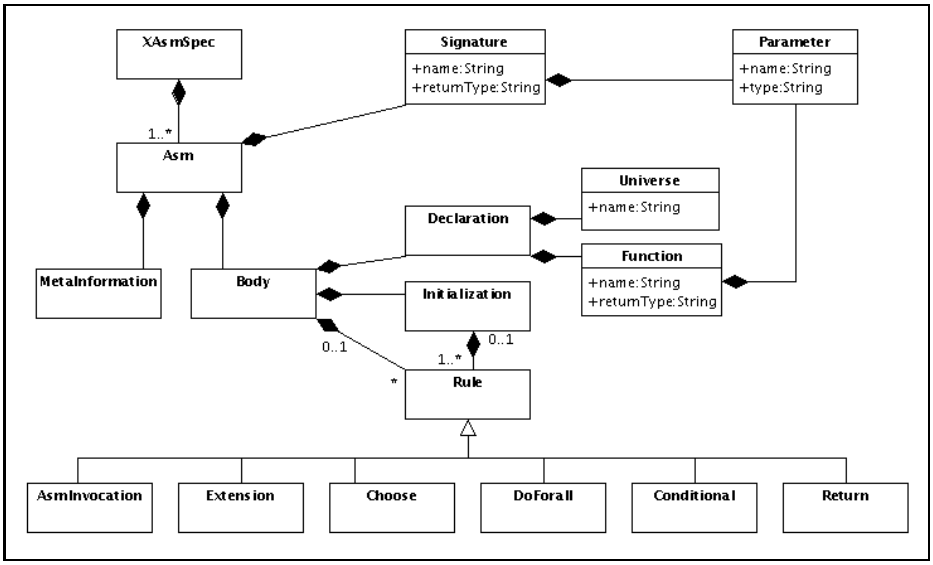

Figure 4: Fragment of the XASM metamodel

enables the definition of ASMs specifications as models. Furthermore, a bridge from the MDE technical space to the XASM [7] concrete syntax has to be provided. For such reasons, in the rest of the section, the ASMs metamodel defined by means of KM3 is presented, and the EBNF model specifying the XASM concrete syntax is also described.

The idea of defining a metamodel for ASMs is not new. In [25] a first step towards a common abstract representation of ASMs has been presented. The main motivation of this work was to define a standard interchange format for a systematic integration of a number of loosely-coupled ASMs tools. Building on the work done in [25] an adapted ASMs metamodel for our purposes has been developed by means of the KM3 formalism. This language has been very useful in supporting rapid and precise definition of metamodels for various situations. It is a textual formalism and even if this seems counter intuitive, in our experience DSL designers have been asking for textual languages instead of visual languages for the definition and modification of metamodels.

Generally, an ASM specification consists of a number of machines and if XASM is the compiler used, each of them has the general structure depicted in Figure 3. In particular an abstract state machine has a signature consisting of a name, a list of parameters and a return type. As defined in [19] types are not part of the core ASM language, but in XASM they can be supplied to the declaration of a function and are used to detect static semantic inconsistencies of the specification. The meta information part contains information concerning the role of the ASM if it is defined as an asset that can be reused by other machines. The body of the machine is composed of different portions. In particular, a list of function and universe declarations is provided together with the list of other used machines that represents certain parts of the overall specification. After the rules that define the initial state of the machine, the transition rules of the form specified in Sec. 3.1 are defined. These rules specify the dynamics of the machine establishing how the value of the declared functions changes eventually invoking some other machines part of the overall specification. For a detailed and complete description of the structure depicted in Figure 3, interested readers can refer to [7].

Starting from the general structure depicted in Figure 3, the corresponding metamodel has been defined in KM3 and a fragment of it is depicted in Figure 4. For readability reason the metamodel is presented in Figure 4 in the standard visual notation of class diagrams. The KM3 code of the complete specification of the metamodel can be downloaded from [4].

```
1  XAsmSpec ::= Asm
2
3  Asm ::= "asm" Signature MetaInformation "is" Body "endasm"
4
5  Body ::= Declaration Initialization Rule*
6
7  ConditionalRule ::= "if" Expression "then" Rule
8                      ("elseif" Rule)*
9                      ("else" Rule)?
10                     "endif"
11
12 Parameter ::= IDENT ":" Type
```

Figure 5: Fragment of the EBNF model for XASM

3.3 ASM syntax

The metamodel presented above enables the definition of ASMs specification in the MDE technical space. In order to compile and execute specifications expressed as models, a specification of XASM concrete syntax is required. For such a purpose an EBNF model has been defined. An excerpt of this model is given in Figure 5.

In order to ”pretty-print” the ASMs metamodel to the XASM concrete syntax an additional entity is required: a mapping between the KM3 metamodel and the EBNF model. Several solutions are available. For instance, an ATL transformation may be used to translate the parse tree into an ASMs model and the other way around. The solution we used is called Textual Concrete Syntax (TCS). It is a DSL for the specification of concrete syntaxes. A detailed description of TCS and of how to perform this model to text translation is however outside of the scope of this paper, which focuses more on semantics, rather than syntax.

4 Session Processing Language

4.1 Overview

The Session Processing Language (SPL) [14] is a domain-specific language whose goal is to ease the development of telephony services based on a Service Logic Execution Environment for SIP [26].

The language offers high-level abstractions that frees the service developer from low-level programming details and technical hassles. One of the most important concepts in SPL is the *session* consisting of a set of handlers and states. The former defines how to deal with a protocol request or events occuring on the platform, while the latter allows some data to be maintained across a set of handlers. In SPL different kinds of sessions have been provided and hierarchically arranged as shown in the example of Figure 6 (which is borrowed from [14]). The `service` session is the main session which contains the `registration` and `dialog` sessions respectively. In particular, the example shows the implementation of a service that counts the calls that have been forwarded to a secretary when the SIP user associated with the service is unable to take the call. When the user registers in the service the counter is set to `0`, increased each time a call is forwarded to the secretary, and finally logged when the user unregisters.

A session at any level has access to all variables of its ancestor sessions. For example, Figure 6 the external function `log`, declared in the outermost `processing` block, is used in the contained `registration` session. Such a session is created for each user who registers on the SIP platform by sending the `REGISTER` request. The session defines the `cnt` variable, a `REGISTER` handler that initializes the counter and an `unregister` handler which logs the counter. Finally, the session `dialog` manages a communication between parties: a dialog is created by the `INVITE` request and confirmed by the `ACK` request and terminated with the `BYE` request. In the example, only the `INVITE` handler has been defined which increments the counter when an incoming call is rejected by the user.

In the sequel, the dynamic semantics to SPL is defined by means of an abstract machine which is structured according to Figure 2.

```
1  service sec_calls {
2    processing {
3      local void log (int);
4      registration {
5        int cnt;
6        response outgoing REGISTER() {
7          cnt = 0;
8          return forward();
9        }
10       void unregister() {
11         log (cnt);
12       }
13       dialog {
14         response incoming INVITE() {
15           response r = forward;
16           if (r != /SUCCESS) {
17             cnt++;
18             return forward 'sip:secretary@nist.gov';
19           } else
20           return r;
21         }
22       }
23     }
24   }
25 }
```

Figure 6: An example of SPL program

4.2 Dynamic semantics

In order to specify the dynamic semantics of SPL, we need to define the abstract data model (ADM) which has to be generated from the metamodel of the language. In particular, since SPL is an imperative language, the ADM should contain all the information necessary to describe not only the semantic actions corresponding to the language primitives, but also how the control flows throughout the program and within the constructs. The specification of the control flow can be given intensionally for each construct (see for instance [8]) in order to inductively decorate the parse tree with all the information necessary for the control flow, i.e. establishing how the control flows from one task to another within the same construct and from one contruct to another.

In Figure 8 a fragment of the abstract data model of SPL is illustrated. In particular, the universes *Processing*, *Registration* and *Dialog* are defined (see line `6`) in order to be extended and initialized according to the sessions specified in an arbitrary SPL program. The different kinds of statements provided by the language induces the definition of the corresponding universes which are subsets of the universe *Statement* as described in the lines `12-13`. The definition of the universes in the lines `8-10` and `15-17` are based on the availability of declarations and expressions in SPL.

The control flow specifying the execution order of the statements is defined during an initial static analysis of the source model and stored with the functions of lines `20-22`. The dynamic semantics heavily relies on such functions which are updated together with the system state by the transition rules in order to let the control evolve through the specified control flow. For uniformity, all the concepts of the abstract data model that have to be

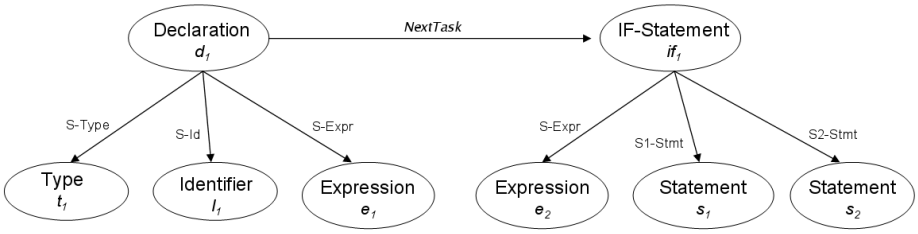

Figure 7: An example of source program encoding

```
1   universe Node
2   universe Statement
3   universe Expression
4   universe Session < Node
5   universe Service
6   universe Processing, Registration, Dialog < Session
7   universe Statement, Expression < Node
8   universe Declaration < Node
9   universe Variable < Node
10  universe Identifier < Node
11
12  universe If, Assignment < Statement
13  universe ExtMethodInvocation < Statement
14
15  universe ForwardExpr < Expression
16  universe Equal < Expression
17  universe RequestURI < Expression
18  universe User
19
20  function CurTask() -> Node
21  function NextTask(n:Node) -> Node
22  function TaskType(n:Node) -> NodeType
23
24  function S-Expr(x:If)-> Expression
25  function S1-Stmt(x:If)-> Statement
26  function S2-Stmt(x:If)-> Statement
27
28  universe RegistrationInstance
29  universe DialogInstance
30  universe PlatformRespone
31  universe SuccessResponse={OK, ACCEPTED, SUCCESS} < PlatformResponse
32  universe ErrorResponse={ERROR} < PlatformResponse
33
34  function HandlerNode(x:String) -> Node
35  external function getMessage() -> String
36  external function EvaluateExpression(e:Expression) -> _
37  external function ForwardToPlatform(_,_) -> PlatformResponse
38  subasm ExecuteHandler(h:Node)
39  subasm AssignValue(x:Identifier, e:Expression)
40  function owner(_)->User
41  function user
```

Figure 8: Fragment of the Abstract Data Model for SPL

involved in the encoding of the program are considered as *Node*. This justify why the universe `Node` of Figure 8 is super-universe of a number of universes according to the hierarchy given in the EBNF definition of the language syntax, e.g. the rule

```
Stmt ::= If | Assignment | ...
```

induced an order-sorted relation which states that the universes *If* and *Assignment* are subsets of *Node*. Furthermore, for each node a number of selectors are provided in order to get access to its internal structure according to the corresponding EBNF rule. For example, for any node of kind *If* the selectors *S-Epxr*, *S1-Stmt* and *S2-Stmt* are defined (see lines `24-26`) according to the occurences of the non terminals in the *If* rule[1]. To better understand such concepts, let's consider the lines `15-20` of the SPL program in Figure 6. This fragment has been defined by means of the following SPL productions

```
Declaration ::=  Type Identifier "=" Expr
If          ::=  "if"  "(" Expr ")" Stmt "else" Stmt
```

and in Figure 7 the corresponding program encoding is depicted and enriched with (simplified) control flow information. These are essentially based on the function *NextTask* that, given the current node of the program (maintained by the function *CurTask*), returns the node that has to be followed. In the example, after the evaluation of the declaration d_1, the if_1 *IF-Statement* has to be executed.

[1] The index in the *S1-Stmt* and *S2-Stmt* function name denotes the first and second occurence of the non-terminal *Stmt*, respectively.

```
1  ...
2  if (TaskType(CurTask) = Declaration)
3  then
4      ExecuteDeclaration(S-Type(CurTask), S-Id(CurTask),
5                         EvaluateExpr(S-Expr(CurTask)))
6                        )
7      CurTask:=NextTask(CurTask)
8  endif
9
10 if (TaskType(CurTask) = If-statement)
11 then
12     if (EvaluateExpr(S-Expr(CurTask)))
13     then
14         CurTask:=S1-Stmt
15     else
16         CurTask:=S2-Stmt
17     endif
18 endif
19 ...
```

Figure 9: Choice of the transition rule to be executed

```
1  ...
2  if (not (exists x in RegistrationInstance : owner(x)=user))
3  then
4     if (ExecuteHandler(HandlerNode("REGISTER")))
5     then
6        if (ForwardToPlatform("INVITE",user) = SUCCESS)
7        then
8           extend DialogInstance with d
9              owner(d):=user
10             CurNode:=NextTask(CurNode)
11          endextend
12       endif
13    ...
14    endif
15 endif
16 ...
```

Figure 10: Fragment of the INVITE handler semantics

As said above, for each node a corresponding transition rule denoting the corresponding semantic action has to be selected and executed. The selection is performed according to the function *TaskType* which return the syntactical category of the node, as shown in the ASM rules of Figure 9 where the right transition rule is executed depending on the type of the current node, subsequently the value of the function *CurTask* is updated. In the example two auxiliary sub-machines have been used, respectively: *ExecuteDeclaration* will extend the environment with a new identifier according to the provided parameters, while *EvaluateExpr* given an expression returns the corresponding evaluation.

The approach presented in Figure 9 is not enough for specifying complete behavior of SPL programs. In fact, as described in Section 4.1, the control flow can be dynamically affected also by the messages coming from the SIP platform. Depending on the received message, the corresponding handler has to be executed. Thus, the ADM specifies a number of external functions able to interact with and provide access to the SIP platform. In particular, for each service, a queue of message is maintained and the function *getMessage()* (line 35) is used for extracting the last incoming message and returning its type (e.g. `INVITE`, `REGISTER`, etc.). To execute the right handler, the function *HandlerNode* (line 34) is defined. Given a message, this external function returns the pointer to the node of the statement that has to be executed. Once obtained, this pointer can be used as parameter in the sub-machine *ExecuteHandler* (line 38) that will execute the statements having as first node pointed by the parameter.

The dynamic semantics of a given handler, apart from the behavior of the statements defined in its body, is based also on some pre-conditions which have to be verified, e.g. as depicted in Figure 10 when a service receive an *INVITE* message the corresponding handler has to verify whether the SIP user sending the message is registered to the service. If not, the handler invokes the *REGISTER* handler (line 4), when available, and then proceeds with

```
1  ...
2  if (ForwardToPlatform("REGISTER",user) = SUCCESS)
3  then
4     extend RegisterInstance with r
5          owner(r):=user
6          CurNode:=NextTask(CurNode)
7     endextend
8  endif
9  ...
```

Figure 11: Fragment of the REGISTER handler semantics

```
1  ...
2  if (end_child_session(r))
3  then
4     RegisterInstance(r):=false
5     CurNode:=NextTask(CurNode)
6     ...
7  endif
8  ...
```

Figure 12: Fragment of the unregister handler semantics

the execution of the handler (line 6) if the registration does not fail.

The execution of a handler requires some interaction with the platform. This is accomplished by means of the external function *ForwardToPlatform* able to forward expressions to the platform and to capture the response which is, in turn, a success or a failure as described in the lines 20-22 of Figure 8. Such a function is used also for the semantics definition of the *INVITE* and *REGISTER* handlers (as shown in Figure 10 and Figure 11). In fact, the body of a handler can be executed only if the platform responses with SUCCESS to the request of INVITE, or REGISTER respectively, performed by a given user.

Finally, concerning the semantics of the *unregister* handler (see Figure 12), if an unregister message arrives and there are still sessions associated with the given registration, this will be deleted (line 4) only after the completition of the child sessions (forced by the sub-muchine end_child_session used in line 2). After this operation, the statements of the *unregister* handler will be executed (line 5).

5 Related Work

The work presented here is a first experiment in giving dynamic semantics of DSLs in the context of MDE. The intention is extend the AMMA framework for supporting the dynamic semantics specification of DSLs. In this direction a more extended experiment has be done in [17] where the ASM formalism is integrated in AMMA and the dynamic semantics of ATL is also given.

The work closest to the one presented here is described in [15] where the ASMs formalism (in particular AsmL [2]) is used as a common semantic framework to define the semantic domain of domain-specific modeling languages. The approach is based on basic behavioral abstractions, called semantic units, that are tailored for the studied problem domain. Semantic units are specified as ASMs. Such semantic units are then anchored to the abstract syntax of the modeling language being specified by means of model transformations. The major difference with the work described here is that, in our approach, the ASMs mechanism is integrated in the AMMA platform. In that way the semantic specifications are models and may be manipulated by operations over models (e.g. model transformations). In the semantic anchoring approach the semantics specification is given outside the model engineering platform, in this case the Generic Modeling Environment (GME).

In the context of MDE some other approaches for semantics specification have been proposed. The approach of Xactium [16, 6] follows the canonical scheme for the specification of semantics of programming languages. In this scheme the semantics is defined by specifying mappings (known as semantic mappings) from abstract syntax to semantic domain. Both the abstract syntax and the semantic domain are given as metamodels. The semantic

mapping is specified by model elements (mostly associations). This approach has become known as denotational metamodeling. The work presented in [21] extends the denotational metamodeling approach by defining Meta Relations as a mechanism for specifying semantic mappings between the abstract syntax and the semantic domain. The dynamic semantics specification (part of the semantic domain) is given by graph transformation rules. This approach is called Dynamic Metamodeling. In our approach the semantic domains and semantic mappings are defined as parts of ASMs. Dynamic aspect is defined by transition rules.

The language Kermeta [23] is a metamodeling language that contains constructs for specifying operations of metamodel elements. These operations may be used for specifying the operational semantics of metamodels and thus the semantics of DSLs expressed in Kermeta. In our approach the operational semantics expressed in ASMs is clearly separated from the metamodel (abstract syntax).

6 Conclusions and Future Work

First generation MDE platforms were usually based on fixed metamodel tools. A typical example is a UML CASE tool with Java code generation facilities. In second generation MDE platforms, we find variable metamodel tools, i.e. tools where the metamodels are not hardwired. This capability allows to deal with a variety of DSLs, in a regular organization. A typical example is the AMMA platform.

As we learn to build and to use these new MDE platforms, we recognize their capabilities to solve many problems, e.g. code generation, program verification, data integration, etc. However these environments are still limited to syntactical interoperability. This paper has reported an initial experiment to bring semantic capabilities to such an environment. AMMA is a framework based on a set of DSLs and allows to define new DSLs. The motivating example of telephony service development has permitted us to show how the three AMMA's basic DSLs (KM3, ATL, and TCS) can cope with most syntactic and transformation issues. In order to broaden the approach to semantics definition, AMMA should be extended with generic facilities. One way is to use ASMs as has been presented in this paper.

One of the lessons learned by this experiment is that ASMs are appropriate for the specification of the dynamic semantics of a wide range of DSLs in the model driven engineering. In fact the obtained specifications can be used both for having readable documentations of the DSL and for having executable descriptions useful for reasoning in the early stages of the language development.

The experimental work presented in this paper remains to be extended in a number of ways. On one hand, the complete semantics of SPL has to be defined and, on the other hand, the efforts have to be devoted in the definition of model to text translations (based on TCS) for obtaining XASM code from models conforming to the proposed ASMs metamodel. In this way the ASMs specifications can be compiled with the XASM compiler and then executed. Furthermore, other experimental works are needed to completely proof the suitability of ASMs in AMMA for the dynamic semantics specification of DSLs and to compare the approach with other semantic frameworks.

Acknowledgments

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

References

[1] KM3 User Manual. The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt.

[2] The Abstract State Machine Language. www.research.microsoft.com/fse/asml.

[3] The ASM Michigan Webpage. http://www.eecs.umich.edu/gasm/.

[4] XASM Metamodel in KM3. The Eclipse Generative Model Transformer (GMT) project, http://eclipse.org/gmt.

[5] ATLAS team: ATLAS Transformation Language (ATL), 2006. Home page, http://www.eclipse.org/gmt/atl.

[6] José M. Álvarez, Andy Evans, and Paul Sammut. Mapping between Levels in the Metamodel Architecture. In *UML*, pages 34–46, 2001.

[7] Matthias Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer, 2000.

[8] Matthias Anlauff, Samarjit Chakraborty, Philipp W. Kutter, Alfonso Pierantonio, and Lothar Thiele. Generating an action notation environment from Montages descriptions. *STTT*, 3(4):431–455, 2001.

[9] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture, European MDA Workshops: Foundations and Applications*, volume 3599, pages 33–46. Springer, 2004.

[10] Egon Börger. A logical operational semantics of full prolog. In *Mathematical Foundations of Computer Science 1990*, pages 1–14. Springer, August 1990.

[11] Egon Börger. High Level System Design and Analysis using Abstract State Machines. In *FM-Trends 98, Current Trends in Applied Formal Methods*, volume 1641 of *LNCS*, pages 1–43. Springer, 1999.

[12] Egon Börger and Wolfram Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, ed., *Formal Syntax and Semantics of Java*, number 1523 in LNCS. Springer, 1998.

[13] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[14] Laurent Burgy, Charles Consel, Fabien Latry, Julia Lawall, Nicolas Palix, and Laurent Rveillre. Language Technology for Internet-Telephony Service Creation. In *IEEE Int. Conf. on Communications*, 2006. To appear.

[15] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic Anchoring with Model Transformations. In *ECMDA-FA*, volume 3748 of *LNCS*, pages 115–129. Springer, Oct 2005.

[16] Tony Clark, Andy Evans, Paul Sammut, and James Willans. *Applied Metamodeling: A Foundation for Language Driven Development*. 2004.

[17] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical report RR. 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), April 2006. Submitted for publication.

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

[19] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, ed., *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[20] Yuri Gurevich and Jim K. Huggins. The Semantics of the C Programming Language. *Computer Science Logic*, 702:274–309, 1993.

[21] Jan Hendrik Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modelin Languages*. Doctoral thesis, University of Paderborn, Paderborn, Germany, 2005.

[22] Philipp W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[23] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS*, pages 264–278, 2005.

[24] OMG. Meta Object Facility (MOF) 2.0 Core Specification, 2003. OMG Document ptc/03-10-04.

[25] Elvinia Riccobene and Patrizia Scandurra. Towards an Interchange Language for ASMs. In *ASM: International Workshop on Abstract State Machines: Theory and Applications*. LNCS, 2004.

[26] A. Roach. Session Initiation Protocol SIP-Specific Event Notification, June 2002. RFC 3265, IETF.

[27] David A. Schmidt. On the need for a popular formal semantics. In *ACM Conf. on Strategic Directions in Computing Research*, volume 32, pages 115–116. ACM SIGPLAN Notices, January 1997.

[28] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *SPLC*, volume 3714 of *LNCS*, pages 198–209. Springer, Oct 2005.

[29] Charles Wallace. The Semantics of the C++ Programming Language. In E. Börger, ed., *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development

Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, Alfonso Pierantonio

Abstract

Domain-Specific Languages (DSLs) are high level languages defined for combining expressivity and simplicity by means of constructs which are close to the problem domain and distant from the intricacies of underlying software implementation constraints. In contrast with general purpose languages, DSLs are typically not useful for generic tasks in multiple application domains. The specification of a DSL is a complex task and requires a lot of knowledge about the domain. In the context of Model Driven Engineering (MDE) metamodeling based techniques are quite commonplace in the syntax specification of DSLs. The definition of their semantics still presents difficulties.
In this paper, a practical experiment is proposed where Abstract State Machines (ASMs) are used as a formal ground for giving, in a precise and unambiguous way, the dynamic semantics of Session Programming Language (SPL), a DSL defined for the development of telephony services over the Session Initiation Protocol (SIP). This experiment is performed in the context of a MDE framework called AMMA (Atlas Model Management Architecture). Although still under development, the approach proposed here illustrates a practical and generic solution to define the precise dynamic semantics of DSLs.

Design Tools and Techniques Miscellaneous

Categories and Subject Descriptors: D.2.3 [**Software**]: Software Engineering; D.2.m [**Software**]: Software Engineering

LINA, Université de Nantes
2, rue de la Houssinière
B.P. 92208 — F-44322 NANTES CEDEX 3