Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs

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

Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs

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* —

LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE

RESEARCH REPORT

N^o^ 06.02

April 2006

UNIVERSITÉ DE NANTES

ECOLE DES MINES DE NANTES

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

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

Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs

20 p.

Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs

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

Over the last years, Model Driven Engineering platforms evolved from fixed metamodel tools to systems with variable metamodels. This enables dealing with a variety of Domain Specific Languages (DSLs). These generic platforms are increasingly adopted to solve problems like code generation. However, these environments are often limited to syntax definitions. The AMMA platform conceives DSLs as collections of coordinated models defined using a set of core DSLs. For broadening the approach to semantics definition, AMMA should thus be extended. The paper presents an extension of the core DSLs of AMMA to specify the dynamic semantics of a range of DSLs by means of Abstract State Machines. Thus, DSLs can be defined not only according to their abstract and concrete syntaxes but also to their semantics in a uniform and systematic way. The approach is validated by means of the semantic bootstrap of the ATL transformation language.

1 Introduction

Over the last years, Model Driven Engineering (MDE) platforms evolved from tools based on fixed metamodels (e.g. a UML CASE tool with ad-hoc Java code generation facilities) to complex systems with variable metamodels. In MDE, metamodels are used to specify the conceptual structure of modeling languages. The flexibility in coping with an open set of metamodels enables the handling a variety of Domain Specific Languages (DSLs), i.e. languages which are close to a given problem domain and distant from the underlying technological assets.

The current MDE platforms are increasingly adopted to solve such problems as code generation [25], semantic tool interoperability [5], checking models [6], and data integration [16]. However, these platforms are often limited to specifying the syntactical aspects of modeling languages such as abstract and concrete syntax. Defining of precise models and performing various tasks on these models such as reasoning, simulation, validation, verification, and others require that precise semantics of models and modeling languages are available. To achieve this, existing MDE platforms have to be extended with capabilities for defining language semantics.

In this paper we use the ATLAS Model Management Architecture (AMMA) as a framework for defining DSLs following MDE principles. AMMA treats a DSL as a collection of coordinated models, which are defined using a limited set of core DSLs. The current set of core DSLs allows to cope with most syntactic and transformation definition issues in language definition. In order to broaden the approach to semantics definition, AMMA should be extended with additional generic facilities.

The paper presents an extension of AMMA to specify the dynamic semantics of a wide range of DSLs by means of Abstract State Machines [10] (ASMs), which are introduced in the framework as a further core DSL. Thus, DSLs can be defined not only by their abstract and concrete syntax but also by their semantics in a uniform and systematic way. The approach is validated by means of the semantic bootstrap of the ATL transformation language.

The structure of the paper is as follows. Section 2 provides the basic definitions and describes the interpretation of DSLs in a MDE setting. Section 3 briefly reviews the ASMs formalism. Section 4 describes the current state of the AMMA framework. Section 5 presents the extension of AMMA with ASMs. In Section 6 a case study is proposed where the dynamic semantics of ATL is proposed. After relating the approach with other works, some conclusions are given in Section 8.

2 Domain-Specific Languages and Models

DSLs are languages able to raise the level of abstraction beyond coding by specifying programs using domain concepts [27]. In particular, by means of DSLs, the development of systems can be realized by considering only abstractions and knowledge from the domain of interest. This contrasts with General Purpose Languages (GPLs), like C++ or Java, that are supposed to be applied for much more generic tasks in multiple application domains. By using a DSL the designer does not have to be aware of implementation intricacies, which are distant from the concepts of the system being implemented and the domain the system acts in. Furthermore, operations like debugging or verification can be entirely performed within the domain boundaries.

Over the years, many DSLs have been introduced in different application domains (telecommunications, multimedia, databases, software architectures, Web management, etc.), each proposing constructs and concepts familiar to experts and professionals working in those domains. However, the development of a DSL is often a complex and onerous task. A deep understanding of the domain is required to perform the necessary analysis and to elicitate the requirements the language has to meet.

As any other computer language (including GPLs), a DSL consists of concrete and abstract syntax definition and possibly a semantics definition, which may be formulated at various degrees of preciseness and formality. In the context of MDE we perceive a DSL as a collection of coordinated models. We are in this way, leveraging the unification power of models [4]. Each of the models composing a DSL specifies one of the following language aspects:

- *Domain definition metamodel.* As we discussed before, the basic distinction between DSLs and GPLs is based on the relation to a given domain. DSLs have a clearly identified, concrete problem domain. Programs (sentences) in a DSL represent concrete states of affairs in this domain. 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). It plays a central role in the definition of the DSL. For example, a DSL for directed graph manipulation will contain the concepts of nodes and edges, and will state that an edge may connect a source node to a target node. Similarly, 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 transformation models that maps the DDMM onto display surface metamodels. Examples of display surface metamodels are SVG or DOT [18], but also XML. A possible concrete syntax of a Petri net DSL may be defined by mapping from places to circles, from transitions to rectangles, and from arcs to arrows. The display surface metamodel in this case has the concepts of Circle, Rectangle, and Arrow;

- *Dynamic semantics.* Generally, DLSs have different types of semantics. For example, OWL [28] is a DSL for defining ontologies. The semantics of OWL is defined in model theoretic terms. The semantics is static, that is, the notion of changes in ontologies happening over time is not captured. 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 governed by the dynamic semantics, there are plenty of other possible operations manipulating programs written in a given DSL. Each may be defined by a mapping represented by a model transformation. 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 is currently an open research subject.

3 Abstract State Machines

3.1 Overview

ASMs [10] 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

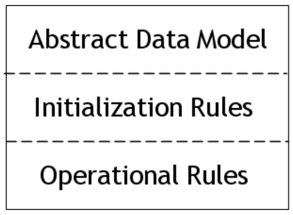

Figure 1: General Structure of the abstract machine specifying the dynamic semantics of a DSL

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.

3.2 DSL Dynamic Semantics Specification with ASMs

In general, giving dynamic semantics to a DSL with ASMs consists of the specification of an abstract machine able to interpret programs defined by means of the given DSL. The machine has to be generic enough to express the behavior of all correct programs. As depicted in Fig. 1 the ASMs specification of such a machine is composed of the following parts:

- *Abstract Data Model (ADM).* It consists of universes and functions corresponding to the constructs of the language and to all the additional elements, language dependent, that are necessary for modeling dynamics (like environments, states, configurations, etc.);

- *Initialization Rules.* They encode the source program that has been defined with the given DSL. The encoding is based on the abstract data model. It gives the initial state of the abstract machine;

- *Operational Rules.* The meaning of the program is defined by means of operational rules expressed in form of transition rules. They are conditionally fired starting from the given instance of the ADM, modifying the dynamic elements like environment, state etc. The evolution of such elements gives the dynamic semantics of the program and simulates its behavior.

ASMs have been used with success in numerous applications and also for specifying the semantics of different languages (like C, Java, SDL, VHDL) [20]. Additionally, ASMs are executable and several compilers and tools are available both from academy and industry supporting the compilation and simulation of ASMs specification. In the rest of the paper the XASM [2] dialect will be used for the description of the ASMs specifications. They can be compiled with the available compiler.

4 The AMMA Framework

AMMA (ATLAS Model Management Architecture) is an MDE framework for building DSLs. It provides tools to specify different aspects of a DSL (see section 2). These tools are based on specific languages. The domain of each of this tool corresponds to one of the aspects of a DSL. AMMA is currently organized around a set of three core DSLs:

- *KM3.* The Domain Definition MetaModel (DDMM) of a DSL is captured as a KM3 [21] metamodel. KM3 is based on the same core concepts used in OMG/MOF [26] and EMF/Ecore [11]: classes, attributes and references. Compared to MOF and Ecore, KM3 is focused on metamodeling concepts only. For instance,

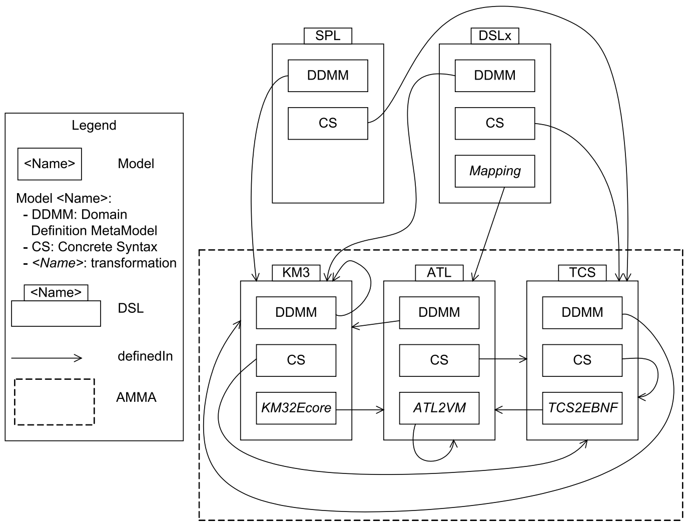

Figure 2: Present State of AMMA

the Java code generation facilities offered by Ecore are not supported by KM3. The default concrete syntax of KM3 is a simple text-based notation.

- *ATL.* Transformations between DSLs are represented as ATL [23] [22] (ATLAS Transformation Language) model transformations. Such transformations can be used to implement the semantics of a source DSL in terms of the semantics of a target DSL. Other potential uses of ATL are: checking models [6], computing metrics on models, etc.

- *TCS.* Textual concrete syntaxes of DSLs are specified in TCS (Textual Concrete Syntax). This DSL captures typical syntactical concepts like keywords, symbols, and sequencing (i.e. the order in which elements appear in the text). With this information, models can be serialized as text and text can be parsed into models. Text to model translation is, for instance, achieved by combining the KM3 metamodel and TCS model of a DSL and generating a context-free grammar.

Figure 2 gives an overview of AMMA as a set of core DSLs. Two other DSLs are shown: SPL [12] (Session Processing Language), which is a language for the domain of internet telephony, and DSLx, which stands for any DSL. The DDMM of each DSL is specified in KM3. TCS is used to specify concrete syntaxes. ATL transformations *KM32Ecore*, *ATL2VM*, and *TCS2EBNF* are used to respecively map the semantics of KM3 to EMF/Ecore, of ATL to the ATL Virtual Machine [22], and of TCS to EBNF (Extended Backus-Naur Form).

Using AMMA does not necessarily means using only these three core DSLs. For instance, MOF or Ecore metamodels can also be used and transformed from and to KM3. Moreover, UML class diagrams specifying metamodels can be used too (i.e. with the UML2MOF.atl transformation). Other AMMA DSLs are also currently the subject of active research, for example AMW [16] (ATLAS Model Weaver) and AM3 [8] (ATLAS MegaModel Management). An overview of AMMA including AMW and AM3 can also be found in [7].

5 Extending AMMA with ASMs

There is currently no tool in AMMA to formally capture the *dynamic semantics* of DSLs. Only informal semantic mappings between DSLs can be specified in the form of ATL transformations. The main principle on which AMMA is built is to consider everything as a model [4]. Following this unification idea, the *dynamic semantics* of a DSL should also be specified as a model. What is required is a DSL in which to specify this semantic model.

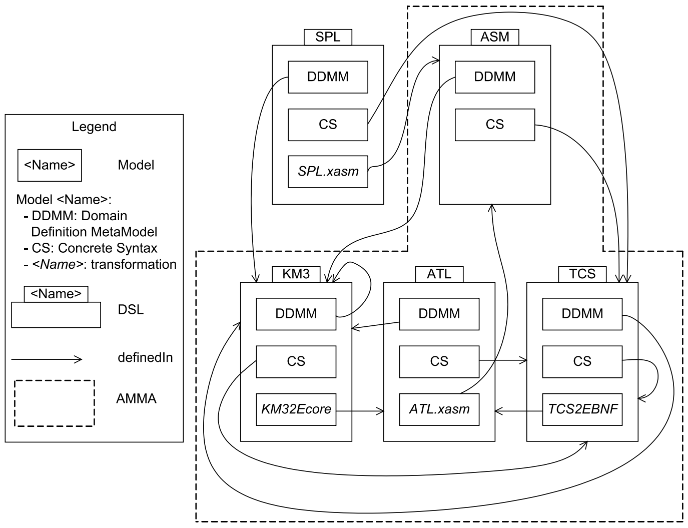

Figure 3: Extending AMMA with ASMs

We decided to integrate ASMs in AMMA instead of designing a new DSL from scratch. For this purpose, we need to specify a KM3 metamodel and a TCS model for ASMs. Figure 3 shows how the ASMs DSL is defined on top of AMMA: its DDMM is specified in KM3 whereas its concrete syntax is specified in TCS. The KM3 metamodel for ASMs is available on the Eclipse GMT website [3]. ASMs may now be considered as an AMMA DSL. Note that there is no semantics specification for ASMs. The reason is that we get this semantics by extracting ASMs models into programs that we can compile with an ASMs compiler.

The next step is to use our newly created ASMs DSL. We experimented by specifying the dynamic semantics of the SPL telephony language and reported our results in [15]. Fig. 3 shows this as the *definedIn* relation between *SPL.xasm* and the ASMs DSL. In this work, we show how the *dynamic semantics* of ATL can also be specified with ASMs. Section 6 gives details on how this is done. Figure 3 represents this as a *definedIn* arrow going from *ATL.xasm* to the ASMs DSL.

6 Case Study: Giving Dynamic Semantics to ATL

6.1 ATL Syntax in a Nutshell

ATL is a hybrid model transformation DSL containing a mixture of declarative and imperative constructs. Its declarative part enables simple specification of many problems, while its imperative part helps in coping with problems of higher complexity. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. During the execution of a transformation source models may be navigated but changes are not allowed. Target models cannot be navigated.

Before describing the specification of the dynamic semantics of ATL, its syntax is presented by means of examples that will be considered in the overall section. Transformation definitions in ATL form *modules*. A module contains a mandatory *header* section, *import* section, and a number of *helpers* and *transformation rules*. Header section gives the name of a transformation module and declares the source and target models (lines 1-2, Fig. 4). The source and target models are typed by their metamodels. The keyword *create* indicates the target model, whereas the keyword *from* indicates the source model. In the example of Fig. 4 the target model bound to the variable OUT is created from the source model IN. The source and target metamodels, to which the source and target model conform, are PetriNet and PNML [9] respectively.

Helpers and transformation rules are the constructs used to specify the transformation functionality. In this paper we consider only transformation rules as basic constructs for expressing the transformation logic.

```
 1 module PetriNet2PNML;
 2 create OUT : PNML from IN : PetriNet;
 3 ...
 4 rule Place {
 5         from
 6                 e : PetriNet!Place
 7                 --(guard)
 8         to
 9                 n : PNML!Place
10                 (
11                         name <- name,
12                         id <- e.name,
13                         location <- e.location
14                 ),
15                 name : PNML!Name
16                 (
17                         labels <- label
18                 ),
19                 label : PNML!Label
20                 (
21                         text <- e.name
22                 )
23 }
```

Figure 4: Fragment of a declarative ATL transformation

Declarative ATL rules are called *matched rules*. They specify relations between *source patterns* and *target patterns*. The name of a rule is given after the keyword rule. The source pattern of a rule (lines 5-7, Fig. 4) specifies a set of *source types* and an optional *guard* given as a Boolean expression in OCL. A source pattern is evaluated to a set of matches in source models. The target pattern (lines 8-22, Fig. 4) is composed of a set of *elements*. Each of these elements (e.g. the one at lines 9-14, Fig. 4) specifies a *target type* from the target metamodel (e.g. the type Place from the PNML metamodel) and a set of *bindings*. A binding refers to a feature of the type (i.e. an attribute, a reference or an association end) and specifies an expression whose value is used to initialize the feature. In some cases complex transformation algorithms may be required and it may be difficult to specify them in a declarative way. For this issue ATL provides two imperative constructs: *called rules*, and *action blocks*. A called rule is a rule called by other ones like a procedure. An action block is a sequence of imperative statements and can be used instead of or in combination with a target pattern in matched or called rules. The imperative statements in ATL are the well-known constructs for specifying control flow such as conditions, loops, assignments, etc.

In the rest of the paper, only the dynamic semantics of ATL declarative rules will be presented. We believe that this does not compromise the validity of the approach since ASMs have already been used for specifying the semantics of several imperative languages.

6.2 Dynamic Semantics of ATL

The operational context of ATL is shown in the left hand side of Fig. 5. An ATL transformation is a model (M_{ATL}) conforming to the ATL metamodel (MM_{ATL}) and it is applied to a source model (M_a) in order to generate a target one (M_b). The source and the target models conform to the source (MM_a) and target (MM_b) metamodels respectively. Parts of the Abstract State Machines (in the right side of Fig. 5) able to interpret ATL transformations are automatically derived from the components in the left hand side of the figure.

The Abstract Data Model (ADM) consists of universe and function declarations needed for the formal encoding of the given ATL transformation and the source and target models. These declarations can be automatically obtained via model transformations from metamodels described in KM3. For example, we transform the KM3 fragment of the PetriNet metamodel (Fig. 6) to the corresponding ASMs code in Fig. 7. The *KM32ASM* ATL transformation performs this canonical translation. For each class in the metamodel, a corresponding universe is specified. If the class is an extension of other classes in the metamodel, the sub-setting facility of ASMs is used. For example, the class Transition (Fig. 6) is transformed into the universe PetriNet_Transition declared as a subset

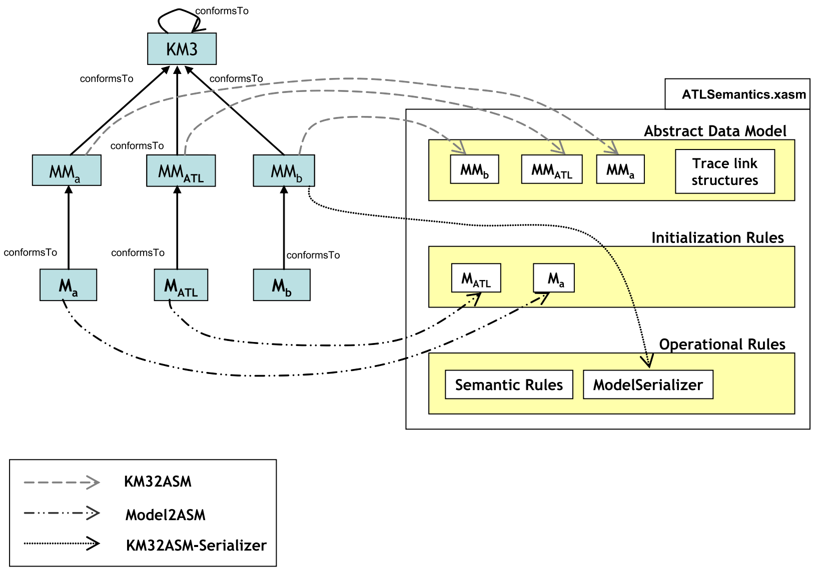

Figure 5: Structure of the dynamic semantics specification of ATL

of the universe PetriNet_Element. The references of the classes are encoded as boolean functions. For example, the incoming arcs of a transition will be encoded with the function incomingArc whose value will be true for all the transitions and arcs (in this case place to transition arcs) that are connected and false otherwise.

The ADM also includes the declaration of universes and functions used for the specification of the dynamic part that evolves during the execution of an ATL transformation. This declaration cannot be automatically generated as it depends on the operational rules that specify the dynamic semantics of ATL. In particular, as explained in the following, the dynamic semantics of ATL is based on the execution of transformation rules. Executing a rule on a match (i.e. elements of the source model) creates a trace link that relates three components: the rule, the match and the newly created elements in the target model. The universe TraceLink (see Fig. 8) contains the trace links that are generated during the execution of the transformations. The source and target elements of the trace link are maintained in the universes SourceElement and TargetElement respectively. For each of them the functions element and patternElement are provided. The function element returns the element of the source model that has matched with the given rule. When applied to an element in TargetElement universe, it returns the new element that has been created in the target model.

The patternElement function, when applied to a source element, returns the source pattern definition of the corresponding ATL rule. The source pattern is a member of universe ATL_SimpleInPatternElement. This universe is derived from the ATL metamodel. In a similar way, when the function is applied to a target element, it returns the target pattern member of the universe ATL_SimpleOutPatternElement (line 12).

The Initialization Rules of the machine depicted in Fig. 5 encode in a formal way the source model and the ATL transformation that has to be interpreted. The encoding is based on the ADM previously described and it gives the initial state of the abstract machine. This encoding can be automatically obtained by transforming the source

```
class Transition extends Element {
      reference incomingArc[1-*] : PlaceToTransition oppositeOf to;
      reference outgoingArc[1-*] : TransitionToPlace oppositeOf from
  }
...
```

Figure 6: Part of the PetriNet metamodel expressed in KM3

```
universe PetriNet_Transition < PetriNet_Element
function incomingArc(a:PetriNet_Transition, b:PetriNet_PlaceToTransition)->Bool
function outgoingArc(a:PetriNet_Transition, b:PetriNet_TransitionToPlace)->Bool
...
```

Figure 7: Part of the PetriNet metamodel specification

model and the ATL program (see the *Model2ASM* transformation in Fig. 5).

The Operational Rules of the machine in Fig. 5 play a key role in the specification of the dynamic semantics of ATL. In particular, the *Semantic rules* part describes the dynamics related to the execution of ATL transformation rules. These rules interpret the given ATL transformation applied to the provided source model (M_a) and generate a formal representation of the target model (M_b).

The execution of ATL transformation rules can be described by means of an algorithm [23] consisting of two steps. In the first step all the source patterns of the rules are matched and the target elements and trace links are created. In the second step the feature initializations of the newly created elements are performed on the base of the previously created trace links and following the bindings specified in the rule target patterns. In the following the ASMs specification encoding these steps are explained with details.

6.2.1 Matching Rules

The formal specification of the first step of the algorithm is based on the sub-machine MatchRule shown in Fig. 9. This machine is invoked for each matched rule contained in the given ATL module. For example, for the module in Fig. 4, the machine is invoked just once for the interpretation of the Place rule.

Given a matched rule, the machine searches in the source model the elements that match the type of the source pattern. In the lines 5-8 the machine selects the elements that defines the source pattern of the matched rule in the universes induced by the ATL metamodel. Such elements are used in the lines 10-11 for the determination of the universe identifier (of the source metamodel) containing the elements that match the source pattern of the considered rule. For example, for the source pattern of the rule in Fig. 4, the lines 10-11 return the universe identifier PetriNet_Place of the source PetriNet metamodel. To obtain this the external functions getValue and sValue are used to handle primitive values.

For each element of the source model contained in the obtained universe, the universes TraceLink and SourceElement have to be extended and the corresponding functions have to be updated (lines 12-16). Furthermore, the universe TargetElement has to be extended for each new element that will be created according to the target pattern of the matched rule (lines 18-32). The identifier of the universes belonging to the target metamodel that have to be extended are determined by means of the code in the lines 23-24. For example, for the transformation of Fig. 4, the universes that will be extended by the MatchedRule machine will be PNML_Place, PNML_Name and PNML_Label belonging to the encoding of the PNML metamodel.

```
universe TraceLink
function rule(t:TraceLink, r: ATL_MatchedRule)->Bool
function sourcePattern(t:TraceLink, x:SourceElement)->Bool
function targetPattern(t:TraceLink, x:TargetElement)->Bool

universe SourceElement
function element(t:SourceElement)->_
function patternElement(t:SourceElement)->ATL_SimpleInPatternElement

universe TargetElement
function element(t:TargetElement)->_
function patternElement(t:TargetElement)->ATL_SimpleOutPatternElement
```

Figure 8: ASM specification for the trace links management

```
asm MatchRule(e:ATL_MatchedRule)
...
is

 choose ip in ATL_InPattern, ipe in ATL_SimpleInPatternElement,
        ipet in ATL_OclModelElement, op in ATL_OutPattern
         : inPattern(e,ip) and elements(ip, ipe)
           and type(ipe,ipet) and outPattern(e,op)
    do forall c in
            $sValue(getValue("name",(getValue("model",ipet))))+"_"
            +sValue(getValue("name",ipet))$
       extend TraceLink with tl and SourceElement with se
          sourcePattern(tl,se) := true
          patternElement(se) := ipe
          element(se) := c
          rule(tl,e) := true
          do forall ope in ATL_SimpleOutPatternElement
             if (elements(op, ope)) then
                 extend TargetElement with te
                    do forall opet in ATL_OclModelElement
                       if (type(ope,opet) ) then
                         extend
                              $sValue(getValue("name",getValue("model",opet)))+"_"
                              +sValue(getValue("name",opet)))$ with t
                           targetPattern(tl, te) := true
                           element(te) := t
                           patternElement(te) := ope
                         endextend
                       endif
                    enddo
                 endextend
             endif
          enddo
       endextend
    enddo
    ...
 endchoose

endasm
```

Figure 9: MatchRule sub-machine specification

6.2.2 Applying Rules

After the creation of the trace links induced by the matched rules, the feature initializations of the newly created elements have to be performed. For example, during the execution of the MatchedRule machine on the rule Place in Fig. 4, the PNML_Place universe is extended with new elements for which the functions *name*, *id* and *location* have to be initialized. The ASMs rules in Fig. 10 set these functions.

For all the trace links, all the bindings of each target pattern have to be evaluated. The bindings are contained in the ATL_Binding universe corresponding to the Binding concept of the ATL metamodel. The properties value and propertyName are also part of the binding specification in the metamodel. For example in the binding `location <- e.location` (line 13, Fig. 4), propertyName corresponds to the attribute `location` whereas value is the OCL expression `e.location`. The lines 6-10 play a key role for the feature initializations of the new elements added during the first step of the algorithm. The external function oclEval is called for the evaluation of the OCL expression of the binding. The value obtained by this evaluation (see line 7), will be then used for the initialization of the target element feature named with the value of propertyName (see line 8). The available oclEval implementation is able to evaluate basic OCL expressions. The improvement of this function for supporting the evaluation of complex OCL expressions could be done by using an available work that describes the dynamic semantics of OCL 2.0 by using ASMs [17]. Due to space limitation, the ASMs code of the oclEval function is not provided here. After the expression of a binding has been evaluated, the resulting value is first resolved before being assigned to the corresponding target element. For this resolution (line 8, Fig. 10) the external function resolve (Fig. 11) is used. The resolution depends on the type of the value. If the type is primitive then the value is simply returned (line 4, Fig. 11). If the type is a metamodel type there are two possibilities: when the value is a target

```
do forall tl in TraceLink
 do forall te : (targetPattern(tl,te))
  choose pe : patternElement(te)=pe
     do forall b in ATL_Binding
       if(bindings(pe,b)) then
         let vExp = getValue("value", b) in
           let v = oclEval(tl, vExp) in
            setValue(sValue(getValue("propertyName",b)), element(te), resolve(v))
           endlet
         endlet
       endif
     enddo
  endchoose
 enddo
enddo
```

Figure 10: Apply rule specification

```
 1 asm resolve(el:_)->_
 2 ...
 3 is
 4  if (isString(el) or isBoolean(el) or
 5      (exists te in TargetElement: element(te)=el)) then
 6
 7        return el
 8  else
 9      choose tl in TraceLink, se in SourceElement,
10             te in TargetElement, op in ATL_OutPattern
11             : element(se)=el) and sourcePattern(tl,se) and
12               targetPattern(tl,te) and elements(op,patternElement(te))and
13               order(op,patternElement(te))=1
14
15        return element(te)
16     endchoose
17  endif
18 endasm
```

Figure 11: Resolve function specification

element (like line 11 in Fig. 4), it is simply returned (line 5, Fig. 11); when the value is a source element (line 12, Fig. 4), it is first resolved into a target element using trace links (line 9-13, Fig. 11). The resolution results in an element from the target model which is then returned (line 15).

6.2.3 Serializing Target Model

Once the semantic rules have been executed, it is necessary to see the results of their execution. For this purpose, the *ModelSerializer* sub-machine in Fig. 5 is called to obtain a textual representation of the generated algebra encoding the target model. This serializer depends on the target metamodel (MM_b). The *KM32ASMSerializer* ATL transformation automatically generates the ASMs code that prints the contents and the values of the universes and functions encoding the obtained target model (M_b).

All the ASMs specifications and the ATL transformations described in this paper are available for download from [3]. Furthermore, the given semantics specification has been validated by formally interpreting the already available *PetriNet2PNLM* [3] ATL transformation.

7 Related Work

The work described here is an extension of an experiment we performed previously, which is reported in [15]. In this experiment we used ASMs to provide dynamic semantics of the SPL language. In this paper we integrate the ASMs mechanism in the AMMA platform and provide another experiment by giving the dynamic semantics of

ATL itself. In [13] ASMs are used as a semantic framework to define the semantics 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 [14, 1] 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 [19] 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 [24] 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).

8 Conclusions and Future Work

In this paper we presented an approach for specifying dynamic semantics of Domain Specific Languages in the context of Model Driven Engineering. Abstract State Machines formalism was integrated into the AMMA platform as a semantics specification framework. ASMs is defined as a part of the set of core AMMA DSLs along with KM3, TCS, and ATL. This allows semantics specifications to be treated as models following the vision that a DSL is a set of coordinated models. In addition, it is still possible to use the existing ASM tools (working outside AMMA) for model simulation and validation purposes.

It should be noted that not all DSLs have dynamic semantics. Furthermore, as the review presented in Related Work section shows, there are different ways for specifying dynamic semantics. The general problem of semantics specification of modeling languages is therefore still open and requires further research. The following observations can be used as a starting point in this direction.

Models are representations of systems. There exists a general system theory that classifies systems in various dimensions, for example, static and dynamic systems. Dynamic systems, in turn, may be classified as discrete or continuous on the base of the underlying model of time. The semantics of a modeling DSL should reflect the nature of the modeled systems. This may be a criteria for selecting the suitable semantics description formalism. In general, it is more likely that more than one semantic framework will be needed to solve problems. Furthermore, we have to consider the practical merits of various frameworks.

Finally, the purpose of semantics specification should be the main factor for selecting the semantic framework. The mature engineering disciplines are based on solid theoretical foundations that allow modeling process that guarantees at a large extent production of reliable systems. The same goal should be pursued in software engineering as well. Semantics formalisms should answer the need for which the model is built: simulation, analysis, reasoning, verification, and/or validation. We need more experimental work in each one of these possible uses of models.

Acknowledgments

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

References

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

[2] 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.

[3] ATLAS team and Davide Di Ruscio. *AM3 Metamodel and Transformation Zoos*. http://www.eclipse.org/gmt/am3/zoos/, 2006.

[4] Jean Bézivin. On the Unification Power of Models. *J. Software and Systems Modeling*, 4(2):171–188, 2005.

[5] Jean Bézivin, Hugo Brunelière, Frédéric Jouault, and Ivan Kurtev. Model Engineering Support for Tool Interoperability. In *Procs of WiSME*, Montego Bay, Jamaica, 2005.

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

[7] Jean Bézivin, Frédéric 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 of *LNCS*, pages 33–46. Springer, 2004.

[8] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Procs of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[9] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *ICATPN*, pages 483–505, 2003.

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

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

[12] 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.

[13] 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.

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

[15] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development. Technical report RR. 06.03, Laboratoire d'Informatique de Nantes-Atlantique (LINA), April 2006.

[16] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Applying Generic Model Management to Data Mapping. In *Procs of the Journées Bases de Données Avancées (BDA05)*, 2005.

[17] Stephan Flake and Wolfgang Mueller. An ASM Definition of the Dynamic OCL 2.0 Semantics. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, eds., *UML 2004*, volume 3273 of *LNCS*, pages 226–240. Springer, 2004.

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

[20] Jim Huggins. The ASM Michigan Webpage. http://www.eecs.umich.edu/gasm.

[21] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *FMOODS*, 2006. To appear.

[22] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In *Procs of ACM SAC 06, Model Transformation track*, Dijon, Bourgogne, France, 2006. To appear.

[23] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer, january 2006.

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

[25] OMG. MOF Model to Text Transformation. OMG Document ad/05-05-04.pdf .

[26] OMG. *Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04*. `http://www.omg.org/docs/ptc/03-10-04.pdf`, 2003.

[27] 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.

[28] World Wide Web Consortium (W3C). Web Ontology Language (OWL). http://www.w3.org/2004/OWL.

Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs

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

Abstract

Over the last years, Model Driven Engineering platforms evolved from fixed metamodel tools to systems with variable metamodels. This enables dealing with a variety of Domain Specific Languages (DSLs). These generic platforms are increasingly adopted to solve problems like code generation. However, these environments are often limited to syntax definitions. The AMMA platform conceives DSLs as collections of coordinated models defined using a set of core DSLs. For broadening the approach to semantics definition, AMMA should thus be extended. The paper presents an extension of the core DSLs of AMMA to specify the dynamic semantics of a range of DSLs by means of Abstract State Machines. Thus, DSLs can be defined not only according to their abstract and concrete syntaxes but also to their semantics in a uniform and systematic way. The approach is validated by means of the semantic bootstrap of the ATL transformation language.

Design Tools and Techniques Miscellaneous

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