Deriving a textual notation from a metamodel: an experience on bridging *Modelware* and *Grammarware*

Angelo Gargantini [⋆], Elvinia Riccobene [⋆⋆], and Patrizia Scandurra [⋆⋆⋆]

Abstract In this paper, we show how the OMG's metamodelling approach to *domain-specific language* definition can be exploited to infer human-usable textual notations (concrete syntaxes) from the conceptualization provided by metamodels (abstract syntaxes). We give general rules to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we show how to instruct a *parser generator* by these rules for generating a *compiler* which is able to parse the grammar and to transfer information about models into a MOF-based instance repository. The approach is exemplified for the *Abstract State Machines Metamodel* (AsmM), a metamodel for the Abstract State Machine (ASM) formal method, by illustrating the derivation of a textual notation to write ASM specifications conforming to the AsmM, and the process to input ASM models into a MOF repository.

1 Introduction

The Model-driven Engineering (MDE) [8] conceives *metamodelling* as a modular and layered way to endow a language or formalism with an *abstract notation*, so separating the abstract syntax and semantics of the language constructs from their different *concrete notations*. A language has to be equipped by at least a proper MOF-compliant metamodel which provides the language abstract syntax, an easy to learn concrete syntax, possibly graphic, a well-founded semantics, and a uniform style (through, e.g., the XML base format [29]) of representing language constructs for interchanging purposes.

The process of deriving textual or graphical concrete notations from a MOF compliant metamodel (*forward engineering*) is not yet well established, while the opposite, from EBNF grammars to MOF (*reverse engineering*), has been more intensively studied [14,17,6,27]. The forward process is more demanding than the reverse, since MOF-based metamodels inherently contain more information than EBNF grammars. Metamodels are graphs with special edges that specify different nodes relationships (generalizations, aggregations, compositions, and so on), whereas EBNF grammars can be presented as tree of nodes and directed edges, but the edges do not contain as much information as properties in a metamodel. A mapping from EBNF grammars to metamodels uses only a subset of the capabilities of metamodels, and the generated metamodel may need to be further enriched in order to make it more abstract.

Complex MOF-to-text tools, capable of generating text grammars from specific MOF-based repositories, exist [16,10]. These tools render the content of a MOF-based

[⋆] University of Bergamo, Italy - email: angelo.gargantini@unibg.it
[⋆⋆] University of Milan, Italy - email: riccobene@dti.unimi.it
[⋆⋆⋆] University of Catania, Italy - email: scandurra@dmi.unict.it

repository (known as a *MOFlet*) in textual form, conforming to some syntactic rules (grammar). However, although automatic, since they are designed to work with any MOF model and generate their target grammar based on predefined patterns, they do not permit a detailed customization of the generated language.

The work presented in this paper is part of our effort toward the development of a general framework for Abstract State Machine (ASM) tools interoperability. Indeed, the increasing application of the ASMs as formal and engineering method for hardware and software systems development, has caused a rapid development of tools around ASMs of various complexity and goals: tools for mechanically verifying properties using theorem provers or model checkers and execution engines for simulation and testing purposes [9]. However, ASM tools have been usually developed by individual research groups, are loosely coupled and have syntaxes strictly depending on the implementation environment. This makes the integration of tools hard to accomplish and prevents ASMs from being used in an efficient and tool supported manner during the software development life-cycle.

In [22,15], we adopted the MDE suggestion upon which a wide interoperability of tools can be reached through metamodels and model transformations: the metamodel of each tool is linked to those of other tools by a logical *pivot metamodel* which abstracts a certain number of general concepts and constructs about the domain-specific language. We defined a metamodel for ASMs with the intention to use it as the pivot metamodel toward a definition of a standard family of languages for the ASMs and a systematic integration of a number of ASM tools upon metamodelling techniques.

After developing the pivot metamodel, as further step, we intended to define a concrete syntax derived from the metamodel as textual notation to write ASM models conforming to the metamodel. Since ASM is not an object-oriented formalism (even if it can model OO concepts), we did not want to use existing tools as HUTN or Anti-Yacc since they provide concrete notations strongly reflecting the object-oriented nature of the MOF meta-language. Therefore, we define general rules how to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we show how to use these mapping rules to instruct the JavaCC *parser generator* for generating a *compiler* which is able to parse the grammar and to transfer information about models into a MOF-based instance repository.

Although the task (possibly iterative) of defining a metamodel for a language is not trivial and its complexity closely matches that of the language being considered, we like to remark that the effort of developing from scratch a new EBNF grammar for a complex formalism, like the ASMs, would not be less than the effort of realizing a MOF-compliant metamodel for the ASMs, and then deriving a EBNF grammar from it. Moreover, the metamodel-based approach has the advantage of being suitable to derive from the same metamodel (through mappings or projections) different alternative concrete notations, for various scopes like graphical rendering, model interchange, standard encoding in programming languages, and so on.

The paper is organized as follows. ASMs and the ASM metamodel are presented in Section 2. Section 3 introduces the rules for deriving an EBNF grammar from a MOF metamodel and discusses how to instruct the JavaCC parse generator to build our compiler. An example of ASM written in the textual notation derived by the metamodel is shown in Section 4. Related and future work are given in Sections 5 and 6.

2 Overview of the case study

Abstract State Machines *Abstract State Machines* (ASMs) [9] are a system engineering method that guides the development of hardware and software systems seamlessly from requirements capture to their implementation. Even if the ASM method comes with a rigorous scientific foundation [9], the practitioner needs no special training to use it since Abstract State Machines are a simple extension of Finite State Machines, and can be understood correctly as pseudo-code or Virtual Machines working over abstract data structures. The computation of a machine is determined by firing *transition rules* describing the modification of the functions from one state to the next. The notion of ASMs moves from a definition which formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, and in a structured and recursive way, *Structured or Turbo ASMs*, to a generalization where multiple agents interact in a synchronous way, or asynchronous agents proceed in parallel at their own speed and whose communications may provide the only logical ordering between their actions, *Synchronous/Asynchronous Multi-agent ASMs*.

We assume the reader is familiar with the ASM method. A complete presentation on the ASMs and their successful application in different fields can be found in [9].

The AsmM metamodel The *Abstract State Machines Metamodel* (AsmM) is a complete meta-level representation of ASMs concepts based on the OMG's MOF metalanguage. The specification of AsmM [7] comprises: (i) an *abstract syntax*, i.e. a MOF-compliant metamodel and OCL constraints representing in an abstract (and visual) way concepts and constructs of the ASM formalism as described in [9]; (ii) an *interchange syntax*, i.e. a standard XMI-based format automatically derived from the AsmM, for the interchange of ASM models; and, (iii) a Java application program interface (API) based on the standard Java Metadata Interfaces (JMI) [18] for MOF 1.4 for managing and accessing metadata (that in our case are ASM models) in a MOF-based repository. For the metamodel *semantics*, we adopt the ASM semantics in [9].

We developed the metamodel in a modular way reflecting the natural classification of ASMs in Basic, Turbo, and Multi-Agent (Sync/Async). Metamodelling representation results into class diagrams (a natural visual rendering of MOF models). Each class is also equipped with a set of relevant *constraints*, OCL invariants written to fix how to meaningful connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams. The complete metamodel contains about 115 classes, 114 associations, and 130 OCL constraints and is organized in one package called `ASM` which is further divided into five sub-packages:

- the ASMStructure package or *the structural language* defines the architectural constructs (modules and machines) required to specify the backbone of an ASM model (this package contains the hierarchy of classes rooted by the class `Asm`);
- the ASMDefinitions package or *the definitional language* contains constructs (functions, domains, rule declarations, etc.) which characterize algebraic specifications;

– the ASMTerms package or *the language of terms* provides all kinds of syntactic expressions which can be evaluated in a state of an ASM;
– the ASMTransitionRules package or *the language of rules* contains all possible transition rules scheme of Basic and Turbo ASMs;
– the DataTypes package specifies further MOF types used to define the AsmM itself (for typing classes attributes).

A standard AsmM library of predefined domains and functions is provided in [7].

We have drawn the AsmM with the Poseidon UML tool (v.4.0) [21] empowered with an ancillary tool UML2MOF which transforms UML models to MOF 1.4 and is provided by the MDR Netbeans framework [3]. The XMI format has been generated automatically from the AsmM: according to the MOF 1.4 to XMI 1.2 mapping [28], a XML document type definition file, commonly named DTD, has been generated from the AsmM in the MDR framework. Similarly, the AsmM JMI was obtained automatically by the MDR framework according to the standard MOF-to-Java mapping [18]. Through the AsmM JMI, a Java client (like the parser generated by the JavaCC, see sec. 3) can access the AsmM packages, create instances of classes of the AsmM, and set and modify attributes and associations of these instances.

3 From MOF to EBNF: how to generate a context-free grammar and instruct a parser generator

In this section, we give mapping rules to derive an EBNF grammar from a MOF metamodel. Even if they have been used to provide the ASM formal method with a textual notation, here called *AsmM-CS*, conforming to the AsmM, they are general enough and do not rely on the specific domain language.

For the MOF to EBNF mapping, we take into account all MOF 1.4 constructs which bring information about the domain knowledge, except constructs like operations and exceptions which are related to the execution semantics of MOF-based repositories rather than to the concepts being meta-modeled, and packages which are used to group elements within a metamodel for partitioning and modularization purposes only. These constructs, however, and, in general, the whole structure of the metamodel are taken into account inside the parser to instantiate and query the content of a MOF repository.

We also provide guidance on how to assemble a JavaCC file given in input to the JavaCC [1] parser generator to automatically produce a parser for the EBNF grammar of the AsmM-CS. This parser is more than a grammar checker: it is able to process ASM models written in AsmM-CS and to create instances of the AsmM in a MDR MOF repository trough the use of the AsmM JMIs.

A JavaCC file contains a sequence of Java-like method declarations each representing the EBNF production rule for a non terminal symbol and corresponding to an identically named method in the final generated Java parser. Each JavaCC method begins with a set of Java declarations and code (to access the MOF repository, create instances of the classes of the metamodel using the AsmM JMIs), which become the initial declarations and code of the generated Java method and hence are executed every time the non-terminal is parsed.

A JavaCC method continues with the *expansion unit* statement, or parser actions, to instruct the generated parser on how to parse symbols and make choices. The expansion unit corresponds to the <expression with symbols>of the EBNF rule and may contain Java code within braces to perform actions like set attributes and references. The expansion unit can also include *lookaheads* of various types – local, syntactic, and semantic – (see [1] for details). Lexical and syntactical analysis errors can be caught and reported using standard Java exception handling. The JavaCC grammar file for the Asm-CS can be found in [7] and consists of about 6852 lines of code. We report here some fragments of it in `typewriter` font.

Note that, the grammar and the input file for the parser generator can be further optimized and enriched. For example, suitable methods were added to the AsmM-CS in order to allow alternative representations of the same concepts (i.e. a class instance in the metamodel can admit many equivalent notations) such as the interval notation for sets/sequences/bags of reals, special expressions to support the infix notation for some functions on basic domains (like plus, minus, mult, etc.), and so on.

Following sections explain the set of **rules** on how to map MOF 1.4 constructs into EBNF and into JavaCC. We assume the reader familiar with MOF and EBNF.

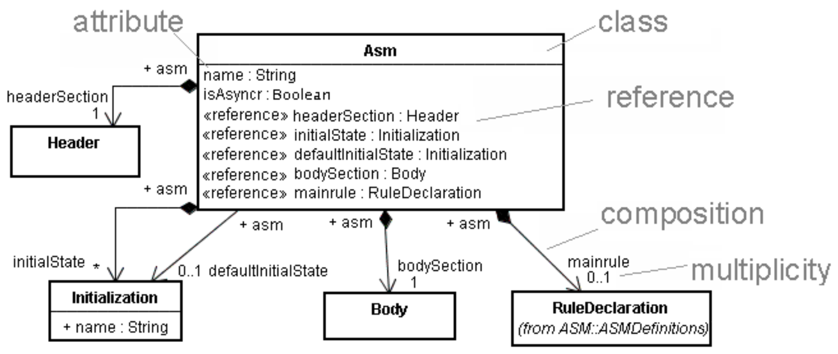
3.1 Class

A MOF class acts as the namespace for attributes and outgoing role names on associations.
Rule 1: A class *C* is always mapped to a non terminal symbol C. User-defined keywords – optional and chosen depending on how one wants the target textual notation appears – delimit the expression with symbols in the derivation rule for C. The expression represents the actual content of the class and is determined by the *full descriptor*[1] of the class according to the other rules below. For each class *C*, we introduce in JavaCC one method which has the following schema.

```
C C(): { // create result, a new instance of C in the repository
         // temp variables for attributes and references
}{ // expansion unit
   <startC> // expression starting delimiter
   // read content of C and fill the instance result
   <endC> // expression ending delimiter
{ return  result;}}
```

The method has signature `C()` and returns a JMI instance of the class `C`. When executed to parse the grammar symbol C, it creates a new instance of `C` called `result` and initializes a list of variables to store attributes and references of *C*. Then it starts parsing the content of *C* enclosed between the keywords `<startC>` and `<endC>`, i.e. it reads attributes and references of *C*, as explained in the following sections, and sets the attributes and references of `result`. In the end it returns `result`.

[1] A full descriptor is the full description needed to describe an object. It contains a description of all of the attributes, associations, etc. that the object contains, including features inherited from ancestor classes.



attribute
class
reference
composition
multiplicity
Asm
name : String
isAsyncr : Boolean
«reference» headerSection : Header
«reference» initialState : Initialization
«reference» defaultInitialState : Initialization
«reference» bodySection : Body
«reference» mainrule : RuleDeclaration
+ asm
headerSection
1
Header
initialState *
0..1 defaultInitialState
Initialization
+ name : String
bodySection
1
Body
mainrule
0..1
RuleDeclaration
(from ASM::ASMDefinitions)

Figure 1. Example of class with attributes, references and associations

Rule 2: The start symbol of the grammar is the non-terminal symbol corresponding to the *root class* of the metamodel, i.e. the class from which all the other elements of the metamodel can be reached.

Example Fig. 1 shows the MOF model of an ASM defined by a name, a Header (to establish the signature), a Body (to define domains, functions, and rules), a mainrule, an Initialization (for the set of initial states), and one initial state elected as default (designed by the association end defaultInitialState). For this class we introduce a non terminal Asm in the grammar and a method `Asm Asm()` in JavaCC. The derivation rule of the non terminal Asm has the keyword "asm" as starting delimiter and no ending delimiter (`<EOF>` in JavaCC code). The class Asm is the root element of the metamodel, therefore its corresponding non terminal is chosen as start symbol of the grammar.

3.2 Multiplicity

Rule 3: Multiplicity values are mapped to repetition ranges. A 0..1 multiplicity (zero or one) is mapped to brackets [ ] or a question mark ?. A * multiplicity (zero or more) corresponds to the application of the Kleene star operator *. A 1..* multiplicity (one or more) corresponds to the the Kleene cross operator +. A n multiplicity (exactly n) corresponds to the operator {n}. A n..* (n or more) multiplicity corresponds to the operator {n,}, a n..m multiplicity (at least n but not more than m) corresponds to the operator {n,m}.

3.3 Data Type

MOF supports two kinds of data type: *primitive data types* like Boolean, Integer, and String; *constructors* like enumeration types, structure types, collection types, and alias types to define more complex types. Primitive data types do not have a direct representation in terms of EBNF elements, while in JavaCC are mapped to the correspondent primitive data types. However, they are used to transform attributes in EBNF concepts (see the next section for details). For structured data types, we do not introduce new

EBNF rules, since each attribute of structured type can be turned in an attribute of Class type by replacing the structured data type with a class definition in the metamodel.

3.4 Attribute (instance-level)

The representation of attributes of a class *C* within the expression on the right-hand of the derivation rule of the nonterminal C depends on the *type* (a MOF data type or a class of the metamodel) of the attribute and on its *multiplicity* (optional, single-valued, or multi-valued). Usually, optional attributes are represented when their value is present, and are not represented when their value is absent.
Rule 4: Attributes of *Boolean* type are represented as keywords (terminal symbols) reflecting the name of the attribute and followed by a question mark ? to indicate it is optional. At instance level, the presence of the keyword in a textual specification indicates that the attribute value is true, and vice-versa.
Rule 5: Attributes of *String* type are represented by a string literal value <STRING> preceded by an optional keyword which reflects the name of the attribute. If a class has an attribute "*name*" of String type, then that attribute is used as *identifier* for objects of the class[2]. We represent the identifier for a class *C* in EBNF by a non terminal <ID_C> which is a sequence of string literals (optional constraints can be given on characters in an identifier). The identifier can be used to retrieve an instance of *C* when needed. In the following, we refer to an object *by name*, if we use its name to univocally refer to it. In JavaCC we introduce a function `C getCByName()` which reads the string ID_C and retrieves the instance of *C* with that name.

No restrictions are placed on the order of the attribute and reference representations (see sec. 3.6) within the production for the non terminal of a class. Although it is expected that the produced grammar has a consistent ordering of the syntactic parts, such ordering is fixed during the derivation process of the grammar from the metamodel (e.g. through interactive wizards), but no extensions (like stereotypes or special tags) are imposed on the MOF-based metamodel in order to reflect the linear order of EBNF.

Example 1 For the attributes of the Asm class in Fig. 1, by **rule 4** and **rule 5**, we introduce the following EBNF derivation rule and JavaCC method:
Asm ::= "asm" ("isAsyncr")? <ID_Asm>

```
Asm Asm(): {Asm result = AsmStructure.getAsm().createAsm();
            String name;
            boolean isAsyncr = false;
} { "asm" ["asyncr" { result.setAsynchr(true);}]
            name = <ID_Asm> { result.setName(name);}
            //read the header, initial states, body, ...
        <EOF>
{ return result;}}
```

Rule 6: Attributes of *Enumeration* type are represented as a choice group of keywords which reflect the name of the enum literals.

[2] Note that in MOF2 the 'isId' attribute can be used for this purpose.

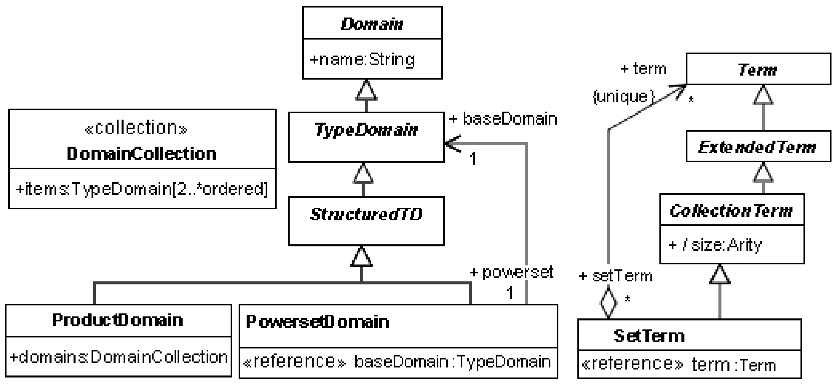


Figure 2. Example of a collection data type and of a multi-valued reference

Rule 7: Attributes of *Integer* type are represented by an optional keyword for the name of the attribute followed by a literal representation of the attribute value.
Rule 8: Attributes of *Collection data type* are represented by an optional keyword which reflect the name of the attribute, followed by a representation of the elements of the collection. Each element can be represented either *by-value*, i.e. by an occurrence of the non terminal of the typing class, or *by-name*, i.e. an occurrence of the identifier if any. Moreover, elements of the collection can be optionally enclosed within parentheses (and), and separated by comma.

Example 2 The class ProductDomain in Fig. 2 has an attribute which is a collection of type-domains. By **rule 1**, we use the initial keyword Prod as delimiter for ProductDomain. We apply **rule 8** omitting the keyword for the attribute domains and we choose to represent the elements of the collection in a by-name fashion (TypeDomain inherits the attribute name from the ancestor class Domain), separated by comma. The feature ordered is reflected by the ordering in the EBNF. The multiplicity 2..* corresponds to the operator {2,}, which is turned into the form $a(a)^+$ with a a syntactic part.
ProductDomain ::= "Prod" "("<ID_Domain> ("," <ID_Domain>)+ ")"
The following method in JavaCC is associated to the class ProductDomain. It creates a new `ProductDomain`, read the delimiters in keyword form and read the list of type-domains by the method `getTypeDomainByName()`, adding them to a new list `domains`. In the end, `domains` is assigned to the domains attribute.

```
ProductDomain ProductDomain(): { ProductDomain result =
             AsmDefinitions.getProductDomain().createProductDomain();
             Collection domains = new LinkedList();
             TypeDomain td;
}{"Prod" "("  td = getTypeDomainByName()
               {/* add td to the domain list */ domains.add(td);}
           ("," td = getTypeDomainByName()
               {/* add td to the domain list*/
domains.add(td); } )+ ")"
```

```
            { /* set the domains */ result.setDomains(domains);}
  { return result;}}
```

Rule 9: Attributes whose type is a *class* of the metamodel are represented by keywords which reflect the name of the attribute, followed by either a full representation of the instance (or *by-value*), i.e. an occurrence of the non terminal of the typing class, or *by-name*, taking into account the multiplicity.
Rule 10: Attributes of *Alias type* are represented depending on the aliased type.
Rule 11: *Derived attributes*[3] are not mapped to EBNF concepts, since the parser can infer them, and then instantiate them in a MOFlet, from other existing elements (which are instead expressed at EBNF level). In JavaCC they are set at the end of the method, just before returning the result.
Rule 12: Other MOF features like visibility, isLeaf, isRoot, changeability, and default values are not considered for an EBNF representation.

3.5 Association and Association End

Associations are represented in terms of their ends, and association ends are represented in EBNF in terms of their corresponding references (see next section). Only *eligible* association ends are represented (**Rule 13**). An association end is considered eligible[4] if it is navigable, if there is no explicit MOF reference for that end within the same outermost package, and if the association of the end is owned by the same package that owns the type of its opposite end (to avoid circular package dependencies). Moreover, similarly to attributes, derived association ends (even if eligible) are ignored.

3.6 Reference

MOF references are a means for classes to be aware of class instances that play a part in an association, by providing a view into the association as it pertains to the observing instance. Here, MOF references are inferred by each *eligible* association end. Therefore, the EBNF representation of a reference depends on the nature (simple, shared aggregation, composite aggregation) of the association to which it refers.
Rule 14: A reference in a *simple association* (that is, the associated instance can exist outside the scope of the other instance) is represented by an optional keyword, which reflects the name of the reference or the role name of the association end, followed by either a by-value or a by-name representation if any, taking into account the multiplicity. Moreover, referenced collections can be optionally enclosed within parentheses, and the syntactic parts for its elements are separated by comma.

[3] The MOF flag *isDerived* determines whether the contents of the notional value holder is part of the explicit state of a class instance, or is derived from other state. Derived attributes or association ends are denoted with a slash (/) preceding the name.

[4] We take this definition from the UML profile for MOF 1.4 by the MDR framework [3]. It is used to automatically imply MOF references by association ends. MOF references are implied, in fact, by each *eligible* UML association end.

Example 1 In Fig. 2, MOF references are shown as attributes with «reference» stereotype, as implied by each *eligible* association end. By **rule 1** for the class PowersetDomain we decided to use the initial keyword Powerset as delimiter, while by **rule 14** we omit the keyword for the reference baseDomain. This last is represented by-name, and the elements of the referenced collection (in this case just one element) are enclosed within parentheses (and).
EBNF: PowersetDomain ::= "Powerset" "(" <ID_TypeDomain> ")"
In JavaCC we introduce a new method `PowersetDomain`, which reads the delimiters in keyword form and read the type-domain by name calling the method `TypeDomain getTypeDomainByName()`.

```
PowersetDomain PowersetDomain(): { PowersetDomain result =
                 AsmDefinitions.getPowersetDomain().createPowersetDomain();
                 TypeDomain baseDomain;
}{ "Powerset" "("  baseDomain = getTypeDomainByName() ")"
          { /*set the baseDomain */
result.setBaseDomain(baseDomain);}
{ return result;}}
```

Rule 15: In a *shared aggregation* (white-diamond, weak ownership, i.e. the part may be included in several aggregates) or in a *composite aggregation* (black-diamond, the contained instance does not exist outside the scope of the whole instance), the reference to the contained instance is represented in the production rule of the non terminal corresponding to the whole class in a *by-value* fashion, i.e. as a non terminal (corresponding to the class of the contained instance) preceded by an optional keyword [5] reflecting the name of the reference or of the role end, and combined with other parts of the production taking into account the multiplicity. Moreover, referenced collections can be optionally enclosed within parentheses, and the syntactic parts for its elements are separated by comma. A reference to the whole instance (if any) is not represented.

Example 2 By **rule 1** for the class SetTerm in Fig. 2 we decided to use the keywords { and } as delimiters, while by **rule 15** we omit the keyword for the reference term. This last is represented in a by-value fashion, and the elements of the referenced collection are not enclosed within parentheses, but are separated by comma. Finally, by **rule 11** the derived attribute size is not represented at EBNF level; however (see the JavaCC code below) inside the parser its value is calculated and set accordingly.
EBNF: SetTerm ::= "{" Term ("," Term)* "}"
The term reference in the SetTerm class is a multi-valued reference. In this case the reference is set by adding elements to the collection returned by the JMI operation `public java.util.Collection getTerm()`. This JMI method returns the value of the reference term, i.e. the collection of elements (as terms) of the set-term.

```
SetTerm SetTerm():{
SetTerm result = AsmTerms.getSetTerm().createSetTerm();
                    Collection term = result.getTerm();
                    Term t;
```

[5] Note that for shared/composite aggregations, the initial keyword for the reference is necessary in case of more than one reference (with different roles) to the same (aggregated) class.

```
}{ "{" t=Term() { term.add(t); }
     ( "," t=Term() { term.add(t);} )* "}"
     { /*sets the derived attribute size*/
result.setSize(term.size());}
{return result;}}
```

Example 3 By the rules above, the production rule for the Asm class in Fig. 1 can be completed as follows.
EBNF: Asm ::= "asm" ("isAsyncr")? <ID_Asm> Header (Initialization)* ("default" Initialization)?
Body ("main" RuleDeclaration)?

```
Asm Asm() : { Asm result = structurePack.getAsm().createAsm();
              String name;
              boolean isAsyncr = false;
              Header headerSection;
              Initialization initialState;
              Body bodySection;
              RuleDeclaration mainrule;
}{ "asm"  [ "asyncr" {result.setAsynchr(true);}]
    name = <ID_Asm> {result.setName(name);}]
    //reads the header and sets the header reference of the ASM
    headerSection = Header(){ result.setHeaderSection(headerSection);}
    //reads the initial states and the body of the ASM
    //reads the main rule of the ASM sets the main rule reference
   [ "main" mainrule = RuleDeclaration(){ result.setMainrule(mainrule);}]
   <EOF>
{ return result;}}
```

3.7 Generalization

We distinguish between a generalization from an *abstract* class and a generalization from a *concrete* class.
Rule 16: In case classes $C_1,..,C_n$ inherit from an *abstract* class C, the production rule for the non terminal C is a choice group C::=C$_1$|..|C$_n$. Attributes and references inherited by classes C_i from the class C are represented in the same way in all production rules for the corresponding non terminals C$_i$.

Example Fig. 3 shows the complete classification of the ASM transition rules under the abstract class *Rule*. The production rule for the non terminal Rule follows.
EBNF : Rule ::= TermAsRule | BasicRule | TurboRule | DerivedRule
In JavaCC we introduce the following method, where `(...|...)` denotes the choice operator.

```
Rule Rule(): { Rule result;
}{ // expansion unit
    (result = TermAsRule() | ... | result = DerivedRule )
{ return result;}}
```

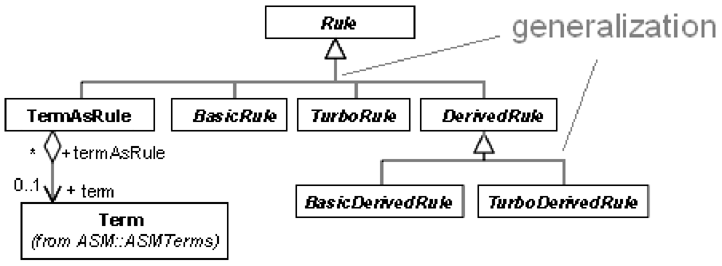


Figure 3. Example of Generalization from an abstract class

Rule 17: If classes $C_1,..,C_n$ inherit from a *concrete* class C, a production rule C::= C_c | C_1|...|C_n is introduced to capture the choice in the class hierarchy, together with a production rule for the new non terminal symbol C_c built according to the content (attributes and references) of of the superclass C. We assume that attributes and references of C inherited by classes C_i are represented in the production rules for the non terminals C_i as in that for C_c.

3.8 Constraint

OCL constraints are not mapped to EBNF concepts. Appropriate parser actions are added to the JavaCC code to instruct the generated parser on how to check whether the input model is well-formed or not according to the OCL constraints defined on the top of the metamodel. In our case, we explicitly implemented in Java an OCL checker by hard-encoding the OCL rules of AsmM. Constraint incompatibility errors are detected and reported using standard Java exception handling. Alternatively, an OCL compiler could be connected to the generated parser for the constraint consistency check.

4 The AsmM Concrete Syntax

The complete AsmM-CS grammar derived from the AsmM can be found in [7]. Fig. 4 shows the specification written in AsmM-CS of a Flip-Flop device originally presented in [9, page 47]. The first rule (FSM) models a generic finite state machine and the second rule (FLIPFLOP) instantiates the FSM for a Flip-Flop:

FSM(*i,cond,rule,j*) =
 if ctl_state = *i* **and** *cond* **then** {*rule,* ctl_state := *j*} **endif**
FLIPFLOP = {FSM(0,*high*,**skip**,1),FSM(1, *low*,**skip**,0)}

5 Related Work

The OMG standardization effort for a *Human Usable Textual Notation* for the Enterprise Distributed Object Computing (EDOC) standard [16], is the main attempt to generate text grammars from specific MOF metamodels. The HUTN proposal is conceptually closer to our approach, as it aims at providing a textual input language for human

```
asm FLIP_FLOP import STDL/StandardLibrary
signature:
 domain State subsetofNatural
 controlled ctl_state : State
 monitored high : Boolean
 monitored low : Boolean
default init initial_state:
 function ctl_state = 0
 function high = false
 function low = false
definitions:
domain State = {0,1}
macro r_Fsm ($ctl_state in State, $i in State, $j in State,
             $cond in Boolean, $rule in Rule ) =
  if $ctl_state=$i and $cond then par  $rule
                                       $ctl_state := $j endpar endif
axiom over high(),low(): not( high and low )
main rule r_flip_flop = par
  r_Fsm(ctl_state,0,1,high,<<skip>>)
  r_Fsm(ctl_state,1,0,low,<<skip>>) endpar
```

Figure 4. Flip-Flop Specification

consumption (XML was explicitly excluded as being insufficiently human-friendly on a large scale). However, although automatic, the HUTN approach is designed to work with any MOF model and generate their target notation based on predefined patterns, it does not, therefore, permit a detailed customization of the generated language which reflect the object-oriented nature of the MOF meta-language.

Another MOF-to-text tool is the Anti-Yacc [10], which can be used to extract the contents of MOF-based repositories in a textual form conforming to some specified syntax. This tool is, therefore, useful to extract information from a MOFlet taking the target language grammar as input, i.e. to realize *walkers* in a MOF repository for code generation, interfacing with legacy syntaxes, and general report writing.

In [14,17], a metamodel for the ITU language SDL-2000 [24] is obtained by a semi-automatic *reverse engineering* process that allows the derivation of a metamodel from the SDL grammar definition. A very similar method to bridge *grammarware* and *modelware* is also proposed by other authors in [6,27]. All these approaches are complementary to our *forward engineering* process. Moreover, while we adopt a pure MOF to write metamodels, in the approaches mentioned above, especially [27], a *MOF profile* which strictly reflects the organization and grouping mechanism of the EBNF formalism is used and therefore the target metamodel obtained from a source grammar requires to be heavily processed not only (as expected) to provide the additional semantics not captured by the EBNF, but also to remove information regarding EBNF technicalities.

Defining *graphical* concrete syntaxes on the top of metamodels is another problem already addressed by the OMG with the adoption of a standard for diagram interchange for UML2 (UML-DI) [26] and by numerous authors (see for example [25]) and meta-

CASE tools like GME [13], DOME [12], AToM3 [11], MetaCASE [19], etc. Indeed, it has to be considered different from the problem addressed in this paper, although related in goal – both concern the definition of concrete syntaxes of languages whose abstract syntax is already available in form of a metamodel.

Concerning the case-study, other previous proposals exist for the definition of a textual notation for ASMs. The AsmL [20] developed by the Foundation Software Engineering group at Microsoft is the greatest effort in this respect. AsmL is a rich executable specification language, based on the theory of ASMs, expression- and object-oriented, and fully integrated into the .NET framework. However, AsmL does not provide a semantic structure targeted for the ASM method. "One can see it as a fusion of the Abstract State Machine paradigm and the .NET type system, influenced to an extent by other specification languages like VDM or Z" [30]. Adopting a terminology currently used in the MDA vision, AsmL is a platform-specific modeling language (PSM) for the .NET type system. A similar consideration can be made also for the AsmGofer language [23]. An AsmGofer specification can be thought, in fact, as a specific PSM for the Gofer environment.

6 Conclusions

this paper, we propose a MOF-to-EBNF mapping to derive a *textual* concrete syntax from a language's metamodel (the abstract syntax). We also give guidelines helping a language designer to instruct, in a semi-automatic way, a traditional parser generator for grammars (like JavaCC) to generate a *metamodel-specific bridge* between the grammarware and the modelware technical spaces. We believe that the MOF is suitable as meta-language for deriving different concrete notations from metamodels, however, further investigations are necessary to decide if arbitrary grammars can be generated.

We implemented the approach for the definition of an EBNF grammar from the ASM metamodel for a textual notation for the ASM formal method. We also provide a parser which processes specifications written in the concrete syntax, checks for their consistency with the metamodel, and translates information about concrete models into a MOF-based instance repository. The applicability of our results to make possible the coupling of different types of ASM tools has been discussed in [22,15]. Soon, we plan to integrate the compiler with a proper Integrated Development Environment which acts as front-end for the modeler, and to develop an ASM virtual machine to simulate AsmM-CS specifications. Furthermore, we intend to upgrade the AsmM to MOF 2.0 and to use the Mof2Text standard or the QVT standard to specify the rules in a more formal way. We are also evaluating the possibility to exploit other metamodelling frameworks, like the ATL project [2], the MTL engine [4], the Xactium XMF Mosaic [5], to better support *model transformations* and *model evolution activities* such as code generation, reverse engineering, model refinement, model refactoring, etc..

References

1. Java Compiler Compiler - The Java Parser Generator. `https://javacc.dev.java.net/`.

2. The ATL model transformation language. `http://www.sciences.univ-nantes.fr/lina/atl/`.
3. The MDR (Model Driven Repository) for NetBeans. `http://mdr.netbeans.org/`.
4. The MTL Engine. `http://modelware.inria.fr/`.
5. The Xactium XMF Mosaic tool suite. `http://www.modelbased.net/www.xactium.com/`.
6. M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical report, Turku Centre for Computer Science, 2003.
7. The Abstract State Machines Metamodel (AsmM) website. `http://www.dmi.unict.it/∼scandurra/AsmM/`.
8. J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2):21–24, 2004.
9. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
10. D. Hearnden and K. Raymond and J. Steel. Anti-Yacc: MOF-to-text. In *EDOC*, pages 200–211, 2002.
11. J. de Lara and H. Vangheluwe. Using AToM^3 as a Meta-Case Tool. In *ICEIS*, pages 642–649, 2002.
12. Honeywell. Dome users guide. 2000.
13. M. J. Emerson, J. Sztipanovits, and T. Bapty. A MOF-Based Metamodeling Environment. *j-jucs*, 10(10):1357–1382, 2004.
14. J. Fischer, M. Piefel, and M. Scheidgen. Using Metamodels for the definition of Languages. In *Proc. of Fourth SDL And MSC Workshop (SAM04)*, 2004.
15. A. Gargantini, E. Riccobene, and P. Scandurra. AsmM the Abstract State Machines matamodel. Technical report, University of Bergamo - Department of Management and Information Technology, 2006.
16. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. `http://www.uml.org/`.
17. J. Fisher and E. Holz and A. Prinz and M. Scheidgen. Toolbased Language Development. In *Proc. of Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04)*, 2004.
18. Java Metadata Interface Specification, Version 1.0, 2002.
19. MetaCASE. ABC to MetaCASE Technology. White paper, 2004.
20. M. R. F. of Software Engineering Group. AsmL: The Abstract State Machine Language. `http://research.microsoft.com/foundations/AsmL/`.
21. Poseidon UML Tool. `http://www.gentleware.com`.
22. E. Riccobene and P. Scandurra. Towards an Interchange Language for ASMs. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer, 2004.
23. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.
24. SDL (Specification and Description Language. ITU Reccomandation Z.100. `http://www.itu.int`.
25. Thomas Baar. Making Metamodels Aware of Concrete Syntax. In *Proc. of ECMDA'05, LNCS Vol. 3748*, pages 190–204, 2005.
26. OMG. UML 2.0 diagram interchange specification, ptc/03-09-01.
27. M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005)*, Montego Bay, Jamaica, 2005.
28. OMG, XML Metadata Interchange (XMI) Specification, v1.2.
29. W3C, The Extensible Markup Language (XML).
30. Y. Gurevich and B. Rossman and W. Schulte. Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .