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

Angelo Gargantini

University of Bergamo - Italy

Elvinia Riccobene

University of Milan - Italy

Patrizia Scandurra

University of Catania - Italy

Outline

- Model-driven Engineering (MDE) for **language definition**
 - Deriving **human-usable textual notations** from metamodels
- Our experience on bridging **Modelware** and **Grammarware**
 - Case study: deriving a textual notation for the Abstract State Machines from their MOF metamodel
- Rules from MOF to EBNF/JavaCC
- Conclusions

Model-driven Engineering (MDE)

- an emerging paradigm for software development
- for **language definition** too
 - to endow a language or formalism with an **abstract notation** provided by a metamodel
 - to separate the abstract syntax and semantics of the language from its different **concrete notations**
 - one concrete notation - language - should be human readable (not only XML/XMI)

Deriving human-usable textual notations from metamodels

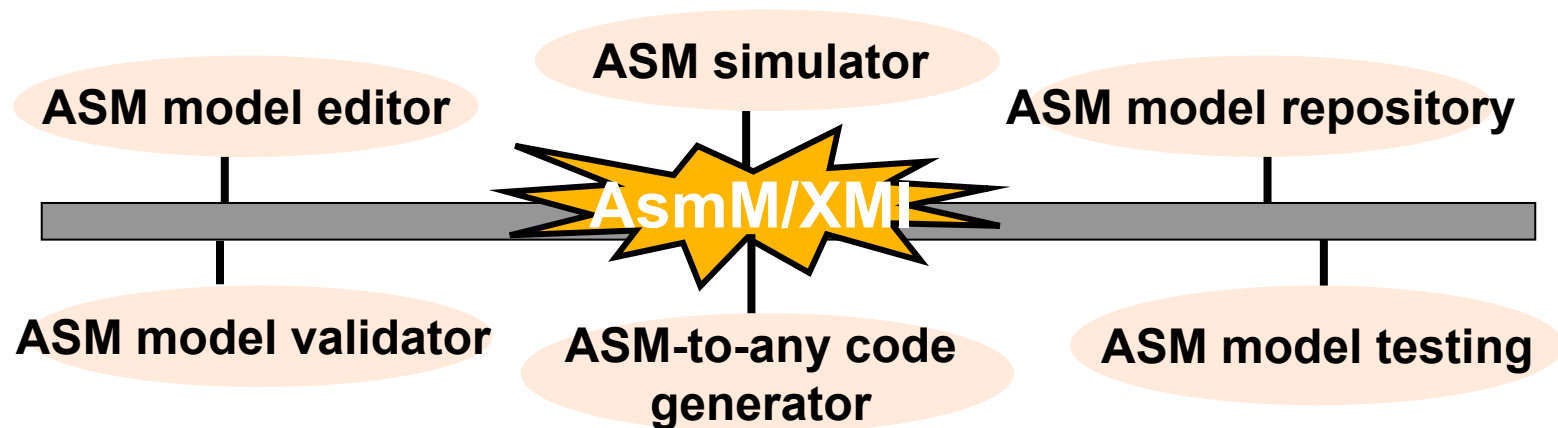
- Deriving textual concrete notations from a MOF-compliant metamodel - **forward engineering** - is not yet well established!
- Complex MOF-to-text tools exist
 - HUTN, Anti-Yacc, MOF2Text, JET for EMF, etc.
 - capable of generating text (grammars) from specific MOF-based repositories
 - either they do not allow a detailed customization of the generated language or they are only one way: from MOF to text
- The “opposite” process
 - from EBNF grammars to MOF - **reverse engineering** – has been more intensively studied

Our experience on Bridging Modelware and Grammarware

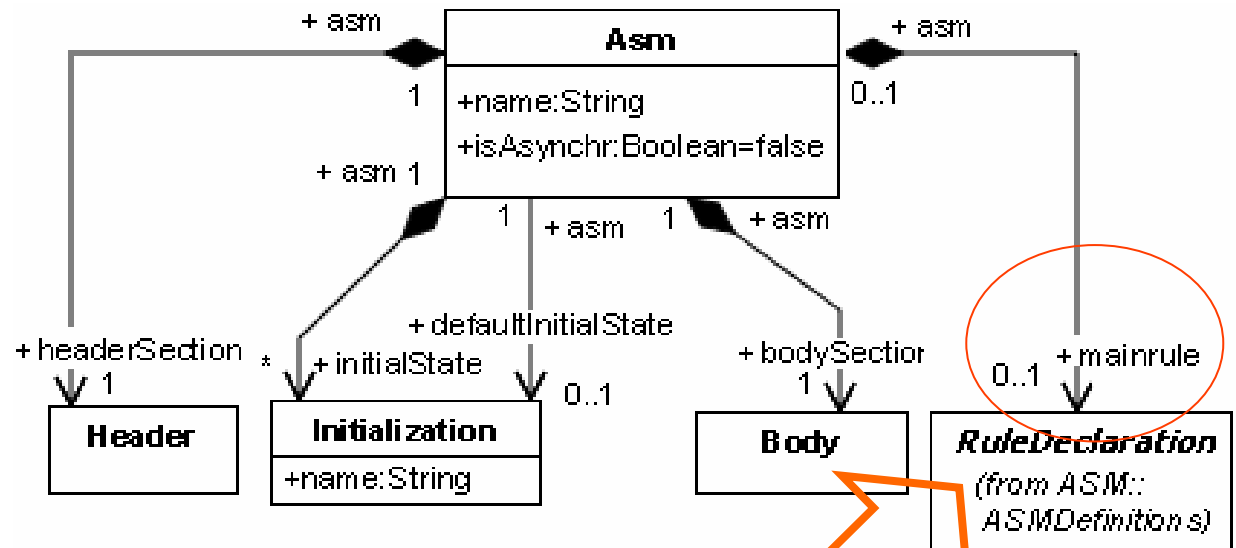
- We define general **rules** to derive a **context-free EBNF** grammar from a **MOF-compliant** metamodel (MOF v. 1.4)
- We use these mapping rules to instruct a **JavaCC parser generator** to get a compiler capable of
 - **parsing** the grammar and
 - transferring information about models **into** a MOF-based **instance repository** (text to MOF objects)
- We apply the proposed approach to the **Abstract State Machine metamodel (AsmM)**

Case Study: the ASM Metamodel (AsmM)

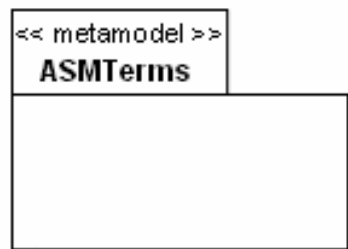
- AsmM: a **complete** MOF-based meta-level representation of Abstract State Machines (ASMs)
- **AsmM** for ASM tools integration
 - like a “PIVOT” metamodel joining metamodels for languages underlying tools, and model transformations
 - a common interchange format built upon XML/XMI



Asm Metamodel (115 classes, 114 associations, 150 OCL rules)



The language of terms



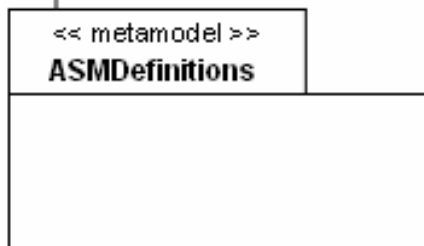
OCL constraint:

context **Asm** inv:

A1: `mainrule->notEmpty()` implies `mainrule.arity=0`

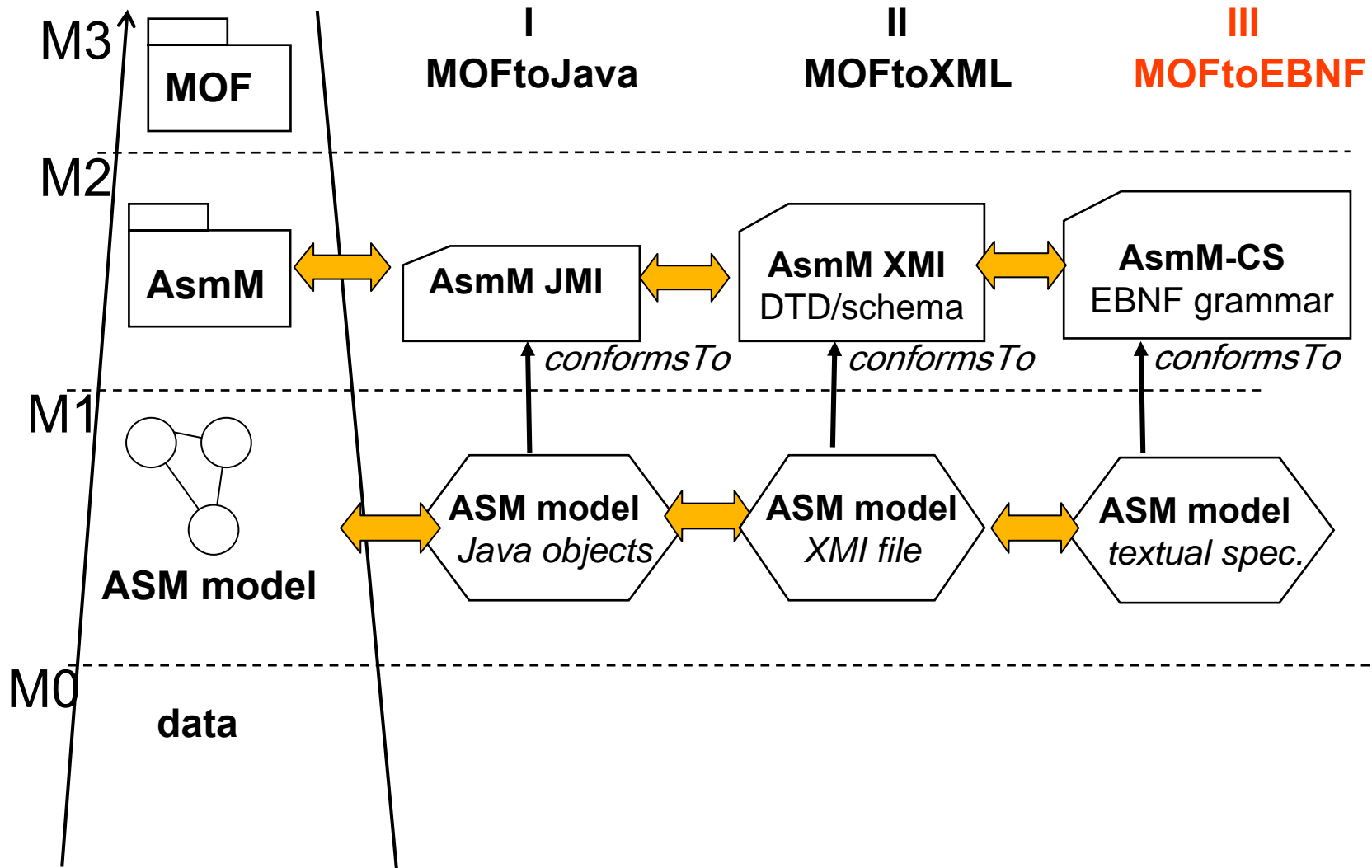
The structural language

The definitional language



The language of rules

Derivative artifacts from the AsmM



Rules from MOF to EBNF/JavaCC

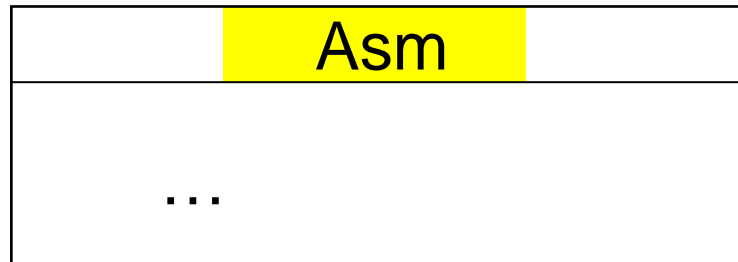
- Translation for the following MOF (1.4) elements to EBNF/ JavaCC
 1. Class
 2. Data Type (primitive and constructors)
 3. Attribute (boolean, String, ...)
 4. References (Association and Association End)
 5. Generalization
 6. (OCL) Constraints

Class

1 A class $C \rightarrow$ a non terminal symbol C . User-defined keywords delimit the expression in the derivation rule for C . The expression represents the actual content of C and is determined by the **full descriptor** (for attributes and associations) of the class according to the other rules.

2 The start symbol is the non-terminal corresponding to the *root* class.

MOF



EBNF **Asm** ::= "asm" ...

Class - JavaCC

- We instruct **JavaCC**, a traditional *Parser Generator* for grammars, to instantiate a MOF-based repository through JMIs
- For each class **C**, in JavaCC one method **C()** parses the grammar symbol **C** and returns a JMI instance of the class **C**

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

```
Asm Asm(): {  
    Asm result = AsmStructure.getAsm().createAsm();  
    // temp variables for attributes and references  
} // expansion unit  
"asm" // start Asm  
// read content of Asm and fill the instance result  
<EOF> // end Asm  
}
```

Multiplicity

3 Multiplicity values → to repetition ranges.

MOF multiplicity

EBNF/JavaCC operators

0..1 → brackets [] or a question mark ?

* → the Kleene star operator *

1..* → the Kleene cross operator +

exactly n → the operator {n}

n..* → the operator {n,}

n..m → the operator {n,m}

Data Types

- MOF 1.4 supports two kinds of data type
 - ❑ *primitive* data types: Boolean, Integer, and String
 - ❑ *constructors*: enumeration, structure types, collection types, and alias types
- 1. **primitive data types** do not have a direct EBNF representation, but in JavaCC they are mapped to the correspondent primitive types
- 2. **constructors** (enumeration, collections, ...)
 - ❑ no new mapping rules
 - ❑ each attribute of structured type is turned in an attribute of Class type

Attribute, Boolean

4 Attribute of Boolean type → keyword followed by ? The presence of the keyword indicates that the attribute value is true, and vice-versa.

MOF

Asm
isAsynchr: Boolean
...

EBNF

Asm ::= "asm" ("asynchr")? ...

JavaCC

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

Attribute, String

5 Attributes of String type → 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 it is used as identifier (and a non terminal <ID_C> is introduced (ID_C can be personalized)).

The name can be used to retrieve an instance of class C (getCbyName)

MOF

Asm
isAsynchr: Boolean
name: String
...

EBNF Asm ::= “asm” (“asynchr”)? <ID_Asm>

JavaCC

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

Other Attributes

6 Attributes of **Enumeration type** → a choice group of keywords which reflect the name of the enum literals

7 Attributes of **Integer type** → an optional keyword for the name of the attribute followed by a literal representation of the attribute value.

9 Attributes whose **type is a class of the metamodel** → 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.

10 Attributes of **Alias type** are represented depending on the aliased type.

11 Derived attributes are not mapped to EBNF concepts, since they can be inferred from other existing elements (which are instead expressed in EBNF).

12 Other MOF features like visibility, isLeaf, isRoot, changeability, and default values are not considered for an EBNF representation.

Attribute: Collection

8 Attributes of Collection data type → 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.
- Elements of the collection can be optionally enclosed within (), separated by comma.

MOF

ProductDomain
domains: DomainCollection

<<collection>> DomainCollection

by-name

EBNF

ProductDomain ::= "Prod" "
("<ID_Domain> ("," <ID_Domain>)+ ")"

JavaCC

```
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;}}
```

Association

- MOF Reference is implied by each eligible UML AssociationEnd
- Association and association ends are represented in EBNF in terms of their corresponding references

13 Only eligible association ends are represented

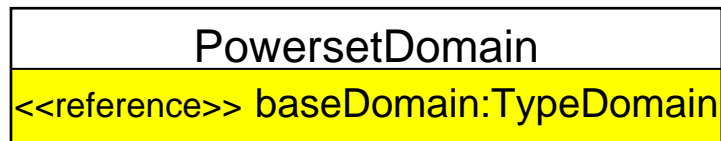
Def. An association end is *eligible*,

- if 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

Reference – simple association

Rule 14: A reference **in a simple association** → 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.

MOF



EBNF

```
PowersetDomain ::= "Powerset"  
                "(" <ID_TypeDomain> ")"
```

JavaCC

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

by-name

Reference - aggregation

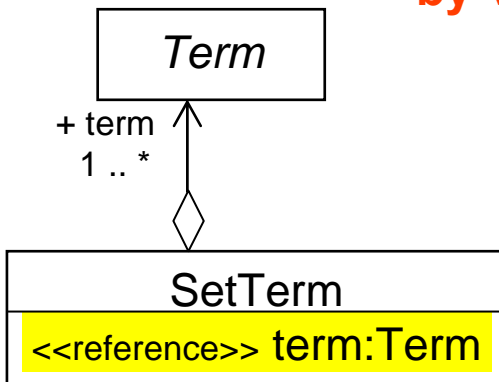
14 A reference **in a shared/composite aggregation** → in the production rule of the “whole” class, in a **by-value** fashion, i.e. as a non terminal preceded by an optional keyword reflecting the name of the reference or of the role end,

- Combined with other parts of the production taking into account the multiplicity

EBNF SetTerm ::= "{" Term ("," Term)* "}"

by-value

JavaCC



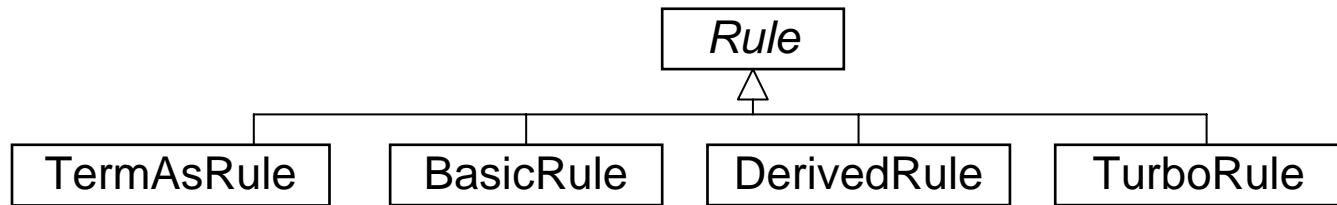
```
SetTerm SetTerm():{  
    SetTerm result = AsmTerms.getSetTerm().createSetTerm();  
    Collection term = result.getTerm();  
    Term t;  
    {" t=Term() { term.add(t); }  
    ( "," t=Term() { term.add(t); })* "  
    /*sets the derived attribute size*/ result.setSize(term.size());  
    {return result;}}
```

Generalization (from an *abstract* class)

Rule 16: In case classes C_1, \dots, C_n inherit from an abstract class C , the production rule for C is a choice group $C ::= C_1 | \dots | C_n$.

- 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 .

MOF



EBNF

$Rule ::= TermAsRule | BasicRule | TurboRule | DerivedRule$

JavaCC

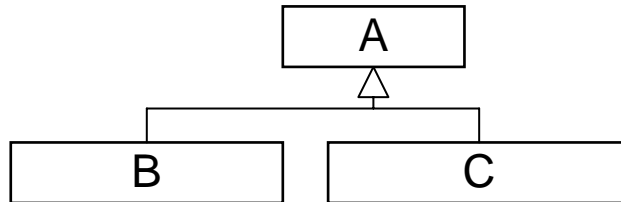
```
Rule Rule(): { Rule result;
  }{ // expansion unit
  (result = TermAsRule() | ... | result = DerivedRule() )
  { return result;}}
```

Generalization (from a concrete class)

Rule 17: If classes C_1, \dots, C_n inherit from a concrete class C ,

- a production rule $C ::= C_c \mid C_1 \mid \dots \mid C_n$ 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 of the superclass C . 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 .

MOF



EBNF

$A ::= A_c \mid B \mid C$

JavaCC

```
A A(): { A result;}{
(result = Ac() | ... | result = C() ) { return result;}}

A Ac() { A result;}{ // read content of A { return result;}}
```

OCL Constraints

- OCL constraints are **not** mapped to EBNF concepts
- Appropriate actions are added to the JavaCC code to check whether the input model is well-formed or not
- In our case, we explicitly implemented in Java an OCL checker
 - ❑ by hard-encoding the OCL rules of AsmM (one method `checkC` for OCL constraints of class `C`)
 - ❑ 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

A Flip-Flop Device in AsmM-CS

Header section

*Import/export clauses,
ASM signature*

```
asm flip_flop
import StandardLibrary
signature:
  domain State subsetof Natural
  dynamic controlled ctl_state : State
  dynamic monitored high : Boolean
  dynamic monitored low : Boolean
```

Initialization section

*Dynamic domains and
functions initializations*

```
default init initial_state:
function ctl_state = 0
function high = false
function low = false
```

Body section

*Static domains/functions
defs., rule decls., and
axioms*

```
definitions:
  domain State = {0,1}

rule 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=true and low =true)
```

Main rule

```
main rule r_flip_flop = par
  r_Fsm(ctl_state,0,1,high,<<skip>>)
  r_Fsm(ctl_state,1,0,low,<<skip>>)
endpar
```

Conclusions and future directions

- MOF can be used to define formal languages (**concrete** notations) in a flexible way
 - EBNF + JavaCC
- + integration of ASM tools upon metamodelling techniques
- **future**
 - Further investigation on the MOF-to-EBNF mapping
 - AsmM to MOF 2
 - Encoding our rules in some MOF to text framework to automatically generate the JavaCC from the metamodel
 - Rigorous approach to operations on models and metamodels