A Concrete Syntax derived from the Abstract State Machine Metamodel

Patrizia Scandurra ⋆, Angelo Gargantini ⋆⋆, Claudia Genovese ⋆, Tiziana Genovese⋆, and Elvinia Riccobene ⋆⋆⋆

Abstract In this paper we present a language for Abstract State Machine specifications. The ASM metamodel (AsmM), introduced in [18], is a MOF-compliant metamodel representing in an abstract and visual way the concepts and constructs of the ASMs formalism as described in [3]. Here we present a *concrete syntax* (AsmM-CS), an EBNF (extended Backus-Naur Form) grammar derived from the AsmM as a textual notation to be used by modelers to effectively write ASM models complaint with AsmM. We also give an overview of the technique applied to derive AsmM-CS from AsmM, showing how the OMG metamodel-based approach can be exploited to derive languages from metamodels.

1 Introduction

Metamodelling is nowadays supported by the Model-Driven Architecture (MDA) [11] approach as a modular and layered way to endow a language with an *abstract notation*, so separating the abstract syntax and semantics of the language constructs from their different *concrete notations.* In the MDA vision, a language has to be equipped by at least a proper metamodel-based definition of the abstract syntax of the language, an easy to learn concrete syntax, possibly graphic, a well-founded semantics, and a uniform style (through, e.g., the XML base format [25]) of representing language constructs for interchanging purposes.

As already discussed in [18], a standard interchange format is of particular interest for the ASM (Abstract State Machines) community, since ASM tools have been usually developed by individual research groups, are loosely coupled and have syntaxes strictly depending on the target environment (compare, for example, AsmGofer [19], ASM-SL [6], XASM [1], ASML [14]).

Taking advantage of the metamodel-based approach, we defined in [18] a new metamodel, called *Abstract State Machine Metamodel* (AsmM, in brief), for the ASM method. The AsmM framework introduces an *abstract syntax* – a MOF-compliant metamodel [12] representing in an abstract and visual way the concepts and constructs of the ASM formalism as described in [3] –, and an *interchange syntax* – a standard XMI-based format [24] automatically derived from the AsmM – for the interchange of ASM models. Here, we present a *concrete syntax* (AsmM-CS), namely an EBNF (extended Backus-Naur Form) grammar derived from the AsmM as a textual notation to write

⋆ Dipartimento di Matematica e Informatica - Università di Catania
⋆⋆ Dipartimento di Ingegneria Gestionale e dell'Informazione - Università di Bergamo
⋆⋆⋆ Dipartimento di Tecnologie dell' Informazione - Università di Milano

ASM models in a textual form. For the *AsmM semantics*, we assume the ASM semantics given in [3]. A complete description of the AsmM 1.0 specification is given in [2].

The AsmM proposal can be viewed as the first step towards a definition of a standard *family of languages* for the ASM formal method and a systematic integration of a number of loosely-coupled ASM tools based upon metamodelling techniques.

We like to remark that the effort of developing a new BNF grammar for an ASM language from scratch would not be less than the effort of developing a MOF-metamodel for ASMs and deriving a BNF grammar from it. Moreover, the metamodel approach has the advantage of being suitable to derive different BNF-like grammars from the same MOF-metamodel, and benefits of some other features as tool support, abstract graphical view, interchange format, standard encoding in programming languages, etc.

The paper is organized as follows. In Section 2, the AsmM concrete syntax is presented. Section 3 shows an example of an ASM specification written using our concrete syntax. The process of deriving the AsmM-CS from the AsmM is described in Section 4. Related and future work are given in Sections 5 and 6.

2 The AsmM Concrete Syntax (AsmM-CS)

To understand the content of the following sections, we assume the reader to be familiar with the AsmM. The AsmM concrete syntax can be divided roughly into four parts, namely the *structural language*, the *definitional language*, the *language of terms,* and the behavioural language (or *language of rules*). The structural language corresponds to the `ASMStructure` package of the AsmM abstract syntax and provides the constructs to describe the structure of an ASM. The definitional language corresponds to the `ASM-Definitions` package of the AsmM abstract syntax and provides the notation to define the basic ASM elements such as functions, domains, rules, and axioms. The language of terms corresponds to the `ASMTerms` package of the AsmM abstract syntax and provides all kinds of syntactic expressions which can be evaluated in an ASM state. Finally, the behavioural language corresponds to the `ASMTransitionRules` package of the AsmM abstract syntax and provides a notation to specify the ASM transition rule schemes.

The following sections present the notation of each syntactic category in a tabular way for better readability. The complete EBNF grammar can be found in [2].

We adopt the following conventions: keywords appear in **bold** face; a pair of square braces [] (not in bold face) indicates that the enclosed expression is optional and must not be considered part of the concrete notation; the notation $t_1,...,t_n$ indicates one or more elements.

2.1 The Structural Language

An ASM model is structured into four sections: a *header*, an *initialization*, a *body* and a *main rule*. Figure 1 shows the concrete syntax for each section of an ASM model. The name of the ASM is specified before the header section together with the optional keyword `isAsyncr` to specify if the ASM is an asynchronous multi-agent or not; `isAsyncr` has no meaning for single-agent ASMs. A lightweight notion of module is also supported; an ASM *module* is an ASM without a main rule and without a characterization of the set of initial states. A module has the same syntax of an ASM with the keyword `asm` replaced by the keyword `module`.

The header section consists of some *import clauses* and one *export clause* which describe the ASM interface for the communication with other ASMs or ASM modules. The *signature* contains the *declarations* of domains and functions. Every ASM is allowed to use only identifiers (for domains, functions and rules) which are declared within its header's signature or imported from other modules. The imported domains and functions will be statically added in the signature of the machine as new functions and the imported rules will enrich the module interface of the machine.

The initialization section consists of a set of *initial states*, one of which is elected as *default*. Figure 1 shows the schema of an initial state. An initial state defines an initial value for each dynamic function and each *concrete-domain* (see sect. 2.2 below) already declared in the signature of the ASM[1]. In addition, the initial state associates each *agent* of the machine with the agent *id* name and *program* (a named transition rule).

The body section consists of *definitions* of static concrete-domains and static/derived functions already declared in the signature[2], *declarations* (definitions) of transition rules, and *declaration* (definitions) of axioms stating assumptions and constraints on functions, domains, and transition rules of the ASM.

The main rule is a named transition rule denoted by the keyword `main`. It is closed (namely it does not contain free variables) so that its semantics depends only on the state of the machine. Executing an ASM means executing its main rule starting from a specified initial state. If the ASM has no main rule, by default, the ASM starts executing in parallel the agent's programs given by the agent initialization clauses in a specified initial state.

2.2 The Definitional Language

To declare an ASM function it is necessary to specify its name, domain, and codomain. The function name must be preceded by one of the keywords *static*, *dynamic* or *derived*, depending on the function type. Dynamic functions are further classified in *monitored*, *controlled*, *shared*, *out*, and *local*. Dynamic functions are allowed to be declared as *local* only in the scope of a turbo rule with local state (see sect. 2.4). They are not considered to be part of the signature of the ASM. Figure 2 shows the concrete syntax for a function declaration.

Our language admits the following domain (or universe) classification: *type-domains* and *concrete-domains*. The type-domains represent all possible *super domains*[3] and are further classified in: *basic type-domains*, domains for primitive data values like booleans, reals, integers, naturals, strings, etc.; *structured type-domains*, domains for building data structures (like sets, sequences, bags, maps, tuples, etc.) over other type-domains; *abstract type-domain*, dynamic user-named domains whose elements have no precise structure and are imported as fresh elements from a possibly infinite reserve

[1] Only dynamic functions and dynamic concrete-domains need to be initialized.

[2] Only static/derived functions and static concrete-domains need to be defined.

[3] In practical applications, the superuniverse $|S|$ of an ASM state S is usually divided into smaller universes. In the AsmM, these smaller *super domains* are called *type-domains*.

Model Element	Concrete Syntax
ASM	[**asynchr**] (**asm**|**module**) name
Header	[**import** m_1[**(**id_{11},...,id_{1h_1}**)**] ... **import** m_k[**(**id_{k1},...,id_{kh_k}**)**]] [**export** id_1,...,id_e] or [**export** *****] **signature:** [dom_decl_1 ... dom_decl_n] [fun_decl_1 ... fun_decl_m] where: – id_{i1},...,id_{ihi} are names for domains, functions and rules imported from module m_i (if omitted all the content of the export clause of m_i is imported), – id_1,...,id_e are names for domains, functions and rules exported from the ASM (export * is used to export all functions and rules), – dom_decl_i and fun_decl_i are declarations of domains and functions
InitialState	[**default**] **init** sn: [**domain** D_1**=** $Dterm_1$ ... **domain** D_n**=** $Dterm_n$] [**function** f_1[**(**p_{11} **in** D_{11},...,p_{1t_1} **in** D_{1t_1}**)**]**=** $Fterm_1$ ... **function** f_m[**(**p_{m1} **in** D_{m1},...,p_{mt_m} **in** D_{mt_m}**)**]**=** $Fterm_m$] [<a_1>**:agent** <r_1> ... <a_u>**:agent** <r_u>] where: – sn is the name of the initial state, – $Dterm_i$ and $Fterm_i$ are terms specifying the initial value of the domains D_i and functions f_i, – p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the functions f_i, – a_i and r_i are respectively the agents *id* and their associated *programs*
Body	**definitions :** [**domain** D_1**=** $Dterm_1$ ... **domain** D_s**=** $Dterm_s$] [**function** f_1[**(**p_{11} **in** D_{11},...,p_{1k_1} **in** D_{1k_1}**)**]**=** $Fterm_1$ ... **function** f_f[**(**p_{f1} **in** D_{f1},...,p_{fk_f} **in** D_{fk_f}**)**]**=** $Fterm_f$] [$rule\_decl_1$ ... $rule\_decl_r$] [$axiom\_decl_1$ ... $axiom\_decl_v$] where: – $Dterm_i$ and $Fterm_i$ are terms specifying the definitions of the domains D_i and functions f_i, – p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the functions f_i, –$rule\_decl_i$ and $axiom\_decl_i$ are declarations of rules and axioms
MainRule	[**main** rule_decl]

Figure 1. Structure of an ASM model

Model Element	Concrete Syntax
StaticFunction	**static** f: [D ->] C
DynamicFunction	[**dynamic**]**monitored** f: [D **->**] C [**dynamic**]**controlled** f: [D **->**] C [**dynamic**]**shared** f: [D **->**] C [**dynamic**]**out** f: [D **->**] C [**dynamic**]**local** f: [D **->**] C
DerivedFunction	**derived** f: [D **->**] C

Figure 2. Notation for declaring a function *f* from *D* to *C*

by means of *extend rules* (see sect. 2.4 below); and *enum domains*, finite user-named enumerations to introduce new concepts of type (e.g. one may define the enumeration *Color = {RED,GREEN,BLUE}* to introduce the new concept of “color”).

Concrete domains are, instead, user-named sub-domains of type-domains. As for functions, a concrete domain can be static or dynamic.

Model Element	Concrete Syntax
BasicTD	**basic domain** D
AbstractTD	**abstract domain** D
EnumTD	**enum domain** D **=** **{**$EL_1,...,EL_n$**}** where $EL_1,...,EL_n$ are the elements of the enumeration
StructuredTD	
ProductDomain	**Prod(**$D_1,D_2,...,D_n$**)**
SequenceDomain	**Seq(**D**)**
PowersetDomain	**Powerset(**D**)**
BagDomain	**Bag(**D**)**
MapDomain	**Map(**D_1,D_2**)** where $D,D_1,...,D_n$ are type-domains over which the structured domains are defined
ConcreteDomain	[**dynamic**] **domain** D **subsetof** TD where TD is the type domain identifying the structure of the elements of the concrete domain D

Figure 3. Notation for declaring a domain D

Figure 3 shows the notation for declaring a domain. As basic type-domains (defined in the standard library) we admit only: `Complex`, `Real`, `Integer`, `Natural`, `String`, `Char`, `Boolean`, `Rule`, and the singleton `Undef`. Moreover, two other special abstract type-domains are considered predefined: the `Agent` domain for agents, and the `Reserve` domain for representing a possibly infinite reserve which provides fresh elements to increase the working space of an ASM. The `Reserve` domain is considered abstract, and therefore *dynamic*, since it is updated automatically upon execution of an *extend rule* (see sect. 2.4 below) – it can not be updated directly by other transition rules –.

Finally, Figure 4 shows named transition rules and axioms declaration.

ModelElement	Concrete Syntax
RuleDeclaration	**`rule`** `[`**`macro`**`] R [(`x_1 **`in`** D_1`,...,`x_n **`in`** D_n`)]= rule` where: – `R` is a transition rule whose body is `rule`, – x_i are variables ranging in the domain D_i and specifying the formal parameters of `R`
Axiom	**`axiom over`** id_1`,...,`id_m`:term` where: – id_i are names of functions, domains and rules. Since functions can be overloaded (provided that their domains differ), to distinguish among functions with the same name, function names must be followed by the name of the domain enclosed in `()`, e.g., as in `f(D)`. – `term` is the term specifying the constraint

Figure 4. Notation for declaring rules and axioms

2.3 The Language of Terms

Terms are syntactic objects denoting elements of ASM states. Terms are interpreted in an ASM state *S* if the elements of the super-universe of *S* – the set of interpretation of function names of the ASM signature – are assigned to the variables of the terms.

As in first-order logic, we admit basic terms (variables, constants, and function applications) and in addition we introduce special terms like tuples, collection terms (sets, maps, sequences, and bags), variable-binding terms (let-terms, comprehension terms, finite quantification terms, etc.), etc. These last special terms have been added by borrowing concepts from the ASM-SL language [6].

The term *RuleAsTerm* is a special term, a *rule term*, used to represent a transition rule where a term is expected (e.g as actual parameter in a rule application to represent a transition rule). Its interpretation results, therefore, in a transition rule.

As in [3], we classify variables (not to be confused with 0-ary functions fixed by the ASM signature) in *logical variables*, *location variables* and *rule variables*. A location variable, appearing as formal parameter in a rule declaration, can be used in the rule body on the left-hand side of an update rule, while a rule variable can be used at places where a transition rule is expected. All other variables are considered *logical*. In a rule application, logical variables have to be replaced by generic terms; location variables have to be replaced by location terms – namely, function terms which start with a dynamic function fixed by the ASM signature –; and rule variables have to be replaced by rule terms.

Within the transition rules, each agent can identify itself by means of a special reserved 0-ary function *self* : *Agent*, which is interpreted by each agent *a* as *a*. Moreover,

Model Element	Concrete Syntax	Examples
ConstantTerm		
ComplexTerm	`x+iy` or `x-iy` where `x` and `y` are real numbers and **i** is the imaginary unit.	`2+i3, -2-i3`
RealTerm	As a floating point numeric literal.	`+3.4, -3.4, 3.4`
IntegerTerm	As a signed numeric literal.	`+3, -3`
NaturalTerm	As a numeric literal.	`3`
CharTerm	As a char literal delimited by single quotes.	`'a'`
StringTerm	A string literal delimited by double quotes.	`"hello"`
BooleanTerm	**`true`** , **`false`**	**`true, false`**
UndefTerm	**`undef`**	**`undef`**
EnumTerm	`e` where `e` is an element of an enumeration type-domain.	`RED`
VariableTerm	`v` where `v` is the name of a variable.	`$x`
FunctionTerm	`[id.]f[(t1,...,tn)]` where: - `f` is the function to apply, - `(t1,...,tn)` is a tuple term representing the actual parameters - `id` is the agent that apply the function `f`.	`max(2,3)` `abs(-4)` `self.f(5)`
LocationTerm	See FunctionTerm.	

Figure 5. Basic Terms Notation

for a function f defined from X to Y, both the expressions $f(self, x)$ and $self.f(x)$ denote the private version of $f(x)$ of the agent *self*.

2.4 The Behavioral Language (or the Language of Rules)

We classify transition rules in two groups: *basic* rules and *turbo* rules. According to [3], the former are simply rules, like the *skip rule* and the *update rule*, while the latter are rules, like the *sequence rule* and the *iterate rule*, introduced to support practical composition and structuring principles of the ASMs. Other rule schemes are derived from the basic and the turbo rules.

Figures 7, 8, and 9 show the concrete notation of all kinds of rules presented in [3].

Variables appearing in rules such as *let*, *forall* and *choose* are not free variable occurrences, but they are bound to the *scope* determined by the rule portion in which they are used. These variables are not stored in the state of the ASM but in a local environment. The formal parameters specified in a rule declaration are, therefore, the only freely occurring variables in the rule body.

We consider a *named rule application* a form of rule. We distinguish, in particular, a *macro rule application* $r[t_1,..,t_n]$ (a *macro-call rule*) – resulting in the execution of the expansion of r obtained replacing in the *body* part of r the formal parameters $x_1,..,x_n$ (of the declaration of r) with the corresponding values of the actual parameters $t_1,..,t_n$ – from a *turbo rule application* $r(t_1,..,t_n)$ (the *turbo-call rule*) with a *turbo submachine* semantics (see sect. 4.1.2 of [3]) to express in abstract form the usual imperative calling

Model Element	Concrete Syntax	Examples

ConditionalTerm	`if G then` t_{then} `[else` t_{else}`] endif`
	where:
	– `G` is a term representing a boolean condition
	– t_{then} and t_{else} are terms of the same nature
		`if $x>0 then 1`
		`   else if $x=0 then 0`
		`                else 0-1`
		`         endif`
		`endif`

DomainTerm	`D` name of a cocrete domain or a type-domain	`Integer, prod(Real,String)`
RuleAsTerm	`<<R>>` where `R` is a transition rule	`<<skip>>,<<f:=1>>`
TupleTerm	`(`t_1`,...,`t_n`)` where t_i are terms with a possibly distinct nature. The empty tuple is not allowed.	`(1,'a',<<skip>>),(true)`

CaseTerm	`switch t`
	`   case` t_1 `:` s_1 `... case` t_n `:` s_n
	`  [otherwise` s_{n+1}`]`
	`endswitch`
	where `t`, t_i are terms of the same nature, and s_j are terms of the same nature too.
		`switch $x`
		`  case 1:'a'`
		`  case 2:'b'`
		`  otherwise'c'`
		`endswitch`

CollectionTerm		
SequenceTerm	`[`t_1`,...,`t_n`]` `[]` for the empty sequence	`["hello‟,"bye"], [[],[1,2]]`
SetTerm	`{`t_1`,...,`t_n`}` `{}` for the empty set	`{[],[1,2],[1]}, {'a','b'}`
BagTerm	`<`t_1`,...,`t_n`>` `<>` for the empty bag	`<1,2,1>, <'a','b','a','b'>`
MapTerm	`{`t_1`->`s_1`,...,`t_n`->`s_n`}` `{->}` for the empty map	`{'a'->1,'b'->2,'c'->3}`
	where t_i are terms of the same nature.	
	Interval notation for finite sequences, sets and bags over reals:	

	`[`t_{low}`..`t_{upp}`[,s]]` `{`t_{low}`..`t_{upp}`[,s]}` `<`t_{low}`..`t_{upp}`[,s]>`
	where:
	– t_{low} and t_{upp} are the lower and the upper real numbers, and
	– `s` is an unsigned real number for the step (if omitted, `s=1`).
		`[1..4] ≡ [1,2,3,4]`
		`{1..2,0.5}≡{1.0,1.5,2.00}`
		`<1..10,2> ≡ <1,3,5,7,9>`

Comprehension Term		

SequenceCT	`[`$t_{v1,...,vn}$`|`v_1 `in` S_1`,...,`v_n `in` S_n `[with` $G_{v1,...,vn}$`]]`	`[g($x) | $x in [0..2*$n-1]`
		`        with $x mod 2=0]`

SetCT	`{`$t_{v1,...,vn}$`|`v_1 `in` D_1`,...,`v_n `in` D_n `[with` $G_{v1,...,vn}$`]}`	`{2+$x\|$x in {0..$n}}`
BagCT	`<`$t_{v1,...,vn}$`|`v_1 `in` B_1`,...,`v_n `in` B_n `[with` $G_{v1,...,vn}$`]>`	`<g($x)\|$x in <0..$n> >`
MapCT	`{`$t_{v1,...,vn}$`->`$s_{v1,...,vn}$`|`v_1 `in` D_1`,...,`v_n `in` D_n `[with` $G_{v1,...,vn}$`]}` where: – $v_1,...,v_n$ are variables and $t_{v1,...,vn}$ is a term, – S_i are collection terms where v_i take their value, and – $G_{v1,...,vn}$ is a term representing a boolean condition.	`{$x->2*$x|$x in{1,3,5,7}}`

LetTerm	`let(`v_1`=`t_1`,...,`v_n`=`t_n`)in` $t_{v1,...,vn}$ `endlet`
	where v_i are variables and $t_1,...,t_n,t_{v1,...,vn}$ are terms.
		`let ($double_x = $x+$x)`
		` in $double_x * $double_x`
		`Endlet`

FiniteQuantification Term		
ExistTerm	`(exist` v_1 `in` D_1`,...,`v_n `in` D_n `[with` $G_{v1,...,vn}$`])`	`(exist $x in X with $x=2)`

ExistUniqueTerm	`(exist unique` v_1 `in` D_1`,...,`v_n `in` D_n
	`[with` $G_{v1,...,vn}$`])`
		`(exist unique $x in X`
		` with $x=0)`

ForallTerm	`(forall` v_1 `in` D_1`,...,`v_n `in` D_n `[with` $G_{v1,...,vn}$`])` where: – $v_1,...,v_n$ are variables, – D_i are terms representing domains where v_i take their value, – $G_{v1,...,vn}$ is a term representing a boolean condition.	`(forall $x in X with $x>=0)`

Figure 6. Extended Terms Notation

Model Element	Concrete Syntax
SkipRule	**Skip**
UpdateRule	l **:=** t where t is a generic term and l can be either a location term or a location variable
BlockRule	**par** R_1 R_2 ... R_n **endpar** where $R_1, R_2, ..., R_n$ are transition rules
ConditionalRule	**if** G **then** R_{then}[**else** R_{else}] **endif** where G is a term representing a boolean condition, R_{then} and R_{else} are transition rules
LetRule	**let** (v_1 **=** t_1, ..., v_n **=** t_n) **in** $R_{v1,...,vn}$ **endlet** where $v_1, ..., v_n$ are variables, $t_1, ..., t_n$ are terms and $R_{v1,...,vn}$ is a transition rule
ForallRule	**forall** v_1 **in** D_1, ..., v_n **in** D_n **with** $G_{v1,...,vn}$ **do** $R_{v1,...,vn}$ where $v_1, ..., v_n$ are variables, $D_1, ..., D_n$ are terms representing domains, $G_{v1,...,vn}$ is a term representing a boolean condition and $R_{v1,...,vn}$ is a transition rule
ChooseRule	**choose** v_1 **in** D_1, ..., v_n **in** D_n **with** $G_{v1,...,vn}$ **do** $R_{v1,...,vn}$ [**ifnone** R] where $v_1, ..., v_n$ are variables, $D_1, ..., D_n$ are terms representing domains, $G_{v1,...,vn}$ is a term representing a boolean condition, $R_{v1,...,vn}$ and P are transition rules
MacroCallRule	r [$t_1, ..., t_n$] where r is the name of a macro rule and $t_1, ..., t_n$ are terms representing the arguments r**[]** stands for a macro application with no arguments
ExtendRule	**extend** D **with** $v_1, ..., v_n$ **do** $R_{v1,...,vn}$ where D is the name of the abstract type-domain to extend, $v_1, ..., v_n$ are logical variables which are bound to the new elements imported from the reserve, and R is a transition rule

Figure 7. Basic Rules Notation

Model Element	Concrete Syntax
SeqRule	**seq** R$_1$ R$_2$ ... R$_n$ **endseq** where R$_1$,R$_2$,...,R$_n$ are transition rules
IterateRule	**iterate** R **enditerate** where R is a transition rule
TurboCallRule	r(t$_1$,...,t$_n$) where r is the name of a transition rule and t$_1$,...,t$_n$ are terms representing the arguments. r**()** stands for a rule application with no arguments
TryCatchRule	**try** P **catch** l$_1$,...,l$_n$ Q where P,Q are transition rules and l$_1$,...,l$_n$ are either location terms or location variables
TurboReturnRule	l **<-** r**(**t$_1$,...,t$_n$**)** where l is either a location term or a location variable and r(t$_1$,...,t$_n$) is a *TurboCallRule* rule
TurboLocalStateRule	**local** f$_1$ **:** [D$_1$ **->**] C$_1$ **[** Init$_1$**]** ... **local** f$_n$ **:** [D$_n$ **->**] C$_n$ **[** Init$_n$**]** body where Init$_1$,...,Init$_n$ and body are transition rules, f$_i$ are local dynamic functions from domain D$_i$ to domain C$_i$

Figure 8. Turbo Rules Notation

Model Element	Concrete Syntax
CaseRule	**switch** t **case** t$_1$ **:** R$_1$ ... **case** t$_n$ **:** R$_n$ [**otherwise** R$_{n+1}$] **endswitch** where t,t$_1$,...,t$_n$ are terms and R$_1$,...,R$_n$,R$_{n+1}$ are transition rules
RecursiveWhileRule	**recwhile** G **do** R where G is a term representing a boolean condition and R is a transition rule
IterativeWhileRule	**while** G **do** R where G is a term representing a boolean condition and R is a transition rule

Figure 9. Derived Rules Notation

mechanism. Moreover, since it is assumed that turbo rule applications have a *call by-name* semantics, i.e. the formal parameters are substituted in the rule *body* by the actual parameters so that these are evaluated only later when they are used, a *call-by-value* evaluation of a rule application can be achieved (as suggested in sect. 4.1.2 of [3]) combining the rule *r* with a *let rule* as follows:

$$r(y_1,\ldots,y_n) = \texttt{let}\ (x_1 = y_1,\ldots,x_n = y_n)\ \texttt{in}\ body\ \texttt{endlet}$$

A *turbo-return rule* is a mechanism which allows one to retrieve the intended return value of a named turbo application rule *r* from a location *l* determined by the rule caller. Semantically speaking, a turbo-return rule denotes a rule with the overall effect of executing the body of *r*, where the 0-ary dynamic function denoted by *l* has been substituted for a reserved 0-ary function *result* which acts as placeholder where to store the intended return value. A good encapsulation discipline will take care, therefore, that *r* (i) contains an update rule of the form $result := t$, and it (ii) does not modify the values of terms appearing in *l*, since they contribute to determine the location where the caller expects to find the return value.

3 An Example of an ASM Specification

In this section, we show the ASM specification of a Flip-Flop. The original model was given at page 47 of [3] by means of the following two rules: the first one (FSM) which models a generic finite state machine and the second one which instantiates the FSM for a Flip-Flop.

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

The AsmM specification of the Flip-Flop follows[4]:

```
asm flip_flop
   signature:
     domain State subsetof Integer
     dynamic controlled ctl_state : State
     dynamic monitored high : Boolean
     dynamic monitored low : Boolean

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

[4] We assume the following rules to distinguish among names of variables, enumeration elements, domains, rules, and functions: a variable identifier starts always with an initial "$"; an enum literal is a string of length greater than or equal to two and consisting of upper-case letters only; a domain identifier begins always with an upper-case letter; a rule identifier always begins with the lower-case letter "r" followed by "_"; a function identifier always begins with a lower-case letter, but can not start with "r_".

```
definitions:
  domain State = {0,1}
  rule r_Fsm($i in State,$cond in Boolean, $rule in Rule,$j in State)=
       if ctl_st = $i and $cond
       then par
               $rule
               ctl_st := $j
            endpar
        endif

  axiom over high(Boolean),low(Boolean): not (high and low)

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

4 How to Derive an AsmM-CS from the AsmM: Mapping MOF to EBNF

MOF is a large OO modelling language with rich concepts to express information models. The main MOF modelling constructs are: *Package*, for containment of classes and associations; *Class*, which contains attributes and participates in associations; *Association*, which represents a set of links between instances of two specified classes and which can have aggregation and composition properties; *Attribute*, which can be either in the form of one of a range of data types or an instance of a class; and *Reference*, which is a class's view on an association in which it participates. For more details on these and other MOF modelling concepts, see the specification [12].

A mapping from MOF-based metamodels to EBNF grammars (*forward engineering*) is more demanding than the opposite (*reverse engineering*). The reason is that MOF-based metamodels inherently contain more information than EBNF grammars. An EBNF grammar can be presented as a tree of nodes and directed edges, but the edges themselves do not contain as much information as properties in a metamodel. Metamodels instead are graphs with special edges that can be interpreted in many ways (generalization semantics, aggregation semantics, composition semantics, etc.). 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.

The AsmM-CS has been derived from the semantic interpretation of the AsmM in order to simplify the mapping between the concrete syntax and the abstract one[5]. In our AsmM we used a subset of the MOF 1.4 constructs which is small enough to be

[5] This approach was inspired by the recent OCL 2.0 specification [13] (which is also based on a MOF-compliant metamodel) made to align the OCL language with respect to UML/MOF 2.0 [23].

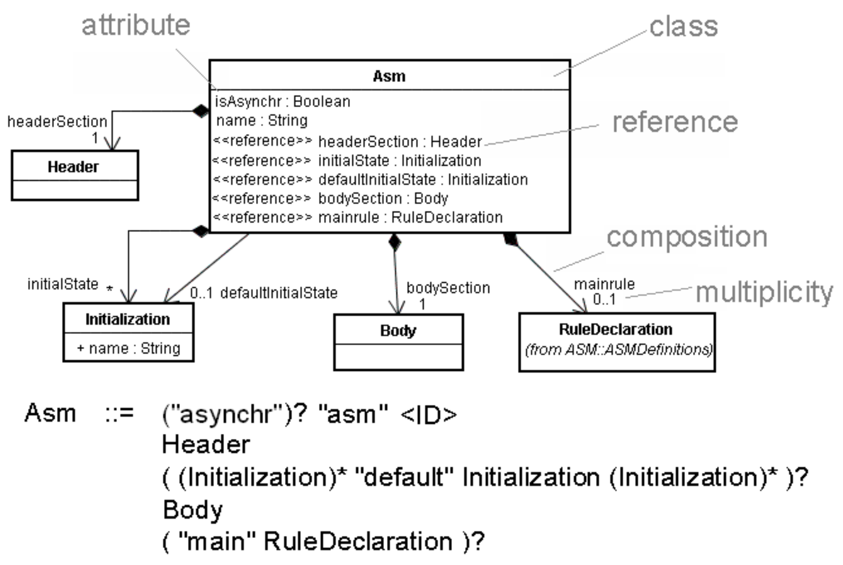
easily translated in EBNF. Table 1 describes the subset of the MOF constructs we used together with the mapping rules from MOF to EBNF we adopted to obtain the AsmM-CS. Figure 10 shows the application of the mapping rules to a fragment of AsmM.

MOF Concepts	**EBNF Concepts**
Package	–
Class	A non terminal C. The production rule for C is determined by the attributes and the relations with other classes.
Attribute (instance level, single primitive value)	
of **Boolean** type	A special keyword reflecting the name of the attribute followed by ?
of **Enum** type	A choice group of keywords reflecting the names of the enum literals
of **String** type	Ientifiers
Association	see Reference
Generalization	
from a concrete super-class	The production rules for the non terminals of the super-class and of the sub-classes are determined as usual. The properties inherited have the same representation in both the production for the non terminal of the super-class and the productions for the non terminals of the sub-classes.
from an abstract super-class A to sub-classes $A_1,\ldots,A_n$	A choice group $A ::= A_1 \mid \ldots \mid A_n$ The properties inherited have the same representation in all productions of the non terminals.
Aggregation	see Reference
Composition	see Reference
Reference	
in a **composition** or **aggregation**	A *full representation*, i.e. an occurrence of the non terminal of the class of the contained instance.
in a **simple association**	A keyword combined with either a *full representation* of the instance (an occurrence of the non terminal), or a representation *by name*, i.e. an occurrence of the identifier of the instance.
Multiplicity	**Repetition ranges**
`0..*`	*
`1..*`	+
`0..1`	? or []

Table 1. Mapping from MOF to EBNF

MOF *References* are a means for classes to be aware of class instances that play a part in an association (either simple or composite), by providing a view into the association as it pertains to the observing instance. For this reason, the representation within the production rule of a non terminal corresponding to a class instance of a reference depends in part on the nature of the association to which it refers.

Since in a composition 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 by a *full representation*,


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

```
Asm   ::=   ("asynchr")? "asm" <ID>
            Header
            ( (Initialization)* "default" Initialization (Initialization)* )?
            Body
            ( "main" RuleDeclaration )?
```

Figure 10. From MOF to EBNF: an example

i.e. as a non terminal (corresponding to the class of the contained instance) combined with other parts of the production taking into account the multiplicities. An eventual reference to the whole instance is not represented.

In a simple association (that is, the associated instance can exist outside the scope of the other instance), instead, the representation of the reference consists of a keyword reflecting the name of the reference (or the name of the association end) combined with either a full representation of the instance (an occurrence of the non terminal of the associated class), or *by name*, i.e. an occurrence of the identifier (the name attribute value of the instance) if any, taking into account the multiplicities.

The grammar obtained, of course, may need to be further optimized and concretized for the purpose of construction of a parser/compiler. Respect to the derivation process presented here, for example, the AsmM-CS has been already customized a bit, 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 well-known functions on basic domains (like plus, minus, mult, etc.), the use of the two keywords **`asm`** and **`module`** to better distinguish an ASM from an ASM module, etc..

5 Related Work

Concerning the definition of a concrete language for ASMs, other previous proposals exist. The Abstract State Machine Language (AsmL) [14] developed by the Foundation Software Engineering group at Microsoft is the greatest effort in this respect. AsmL is a rich executable specification language, expression and object oriented, based on the theory of Abstract State Machines and fully integrated into the .NET framework and Microsoft development tools; 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” [26]. Adopting a terminology currently used in the MDA [11] approach, AsmL is a powerful platform-specific modelling language (PSM) for the .NET type system. A similar consideration can be made also for the AsmGofer language [19]. An AsmGofer specification can be seen, in fact, as a specific PSM for the Gofer environment. Other specific languages for the ASMs, no longer maintained, are ASM-SL [6], which adopts a functional style being developed in ML and which has inspired us in the language of terms, and XASM [1] which is integrated in Montages, an environment generally used for defining programming language semantics and grammar.

A platform-independent modelling (PIM) language for ASMs could allow to define precise *transformation bridges* in order to automatically map an ASM PIM into an AsmGofer-PSM, or into an AsmL-PSM, and so on. In the same manner, we may “compile” ASMs models into programming languages such as C++, C#, Java and so on, to provide efficient code generation capabilities and *round trip engineering* facilities as well.

Concerning the metamodelling technique for the definition of languages, we can mention the official metamodels supported by the OMG [15] for MOF itself [12], for UML [22], for OCL [13], for CWM [4], etc. Academic communities like the Graph Transformation community [21,16] and the Petri Net community [17,7], have also started to settle their tools on general metamodels and XMI/XML-based formats.

Recently, a metamodel for the ITU language SDL-2000 [20] was developed [9,10]. The authors present a semi-automatic *reverse engineering* methodology that allows the derivation of a metamodel from a formal syntax definition of an existing language. The SDL metamodel has been derived from the SDL grammar using this methodology. Their method is complementary with the derivation process presented in this paper (see Sect. 4 above), since our approach has to be considered a *forward engineering* process consisting in deriving a concrete textual notation from an abstract metamodel.

Other more complex MOF-to-text tools, capable of generating text grammars from specific MOF-based repositories, exist [8,5]. These tools render the content of a MOF-based 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.

6 Conclusions and Future Directions

Within the language engineering area, metamodels provide a standardized visual representation easy to learn and supported by a number of tools to design, implement and document languages.

We propose a metamodelling-based definition of a language, the AsmM, for the ASM theory described in [3]. The AsmM has been defined using two metalanguages:

the MOF [12] and EBNF. The MOF is used to describe the abstract syntax, while the EBNF is used to describe the (textual) concrete syntax.

A prior version of the AsmM and also of an XML/XMI -based interchange syntax was presented in [18]. In this paper, we present the AsmM concrete syntax and we describe the MOF-to-EBNF mapping rules applied to derive the concrete syntax from the abstract syntax.

The all AsmM is still evolving because, in order to make it a "standard", one of our main goals is to modify existing constructs and add (or even remove) concepts to meet the needs of the ASM community. We also plan to develop a parser for the AsmM language based on our syntax definition to link it to a proper modelling environment, and to provide support for "transformations" (text-To-MOF, MOF-To-text, XMI-To-MOF, MOF-To-XMI, MOF-To-ANY, etc.). In this way, a designer could write her/his ASM specification in the textual form of AsmM-CS, transform it in the XMI interchange format easily readable and modifiable by tools.

Benefits provided by a standardized notation for the ASMs may contribute to increase the practical use of the ASMs method and provide an efficient interaction among ASM tools for a higher quality design based on the ASM formalism.

References

1. M. Anlauff and P. Kutter. Xasm: The Open Source ASM Language. `http://www.xasm.org`.
2. The Abstract State Machines Metamodel (AsmM) website. `http://www.dmi.unict.it/~scandurra/AsmM/`.
3. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
4. OMG, The Common Warehouse Metamodel. `http://www.omg.org/cwm/`.
5. D. Hearnden and K. Raymond and J. Steel. Anti-Yacc: MOF-to-text. In *EDOC*, pages 200–211, 2002.
6. G. Del Castillo. The ASM Workbench, a general-purpose ASM tool set based on the ASM-SL language. `http://www.uni-paderborn.de/fachbereich/AG/rammig/DE/gruppe/giusp/workbench/index.html`.
7. J. Bézivin. E. Breton. Towards an Understanding of Model Executability. In *Proc. FOIS 2001*.
8. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. `http://www.uml.org/`.
9. J. Fischer and M. Piefel and M. Scheidgen. Using Metamodels for the definition of Languages. In *Proc. of Fourth SDL And MSC Workshop (SAM04). To appear.*, 2004.
10. 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.
11. OMG, The Model Driven Architecture (MDA). `http://www.omg.org/mda/`.
12. OMG, The Meta Object Facility Specification, document formal/2002-04-03, version 1.4.
13. OMG, Response to the UML 2.0 OCL RfP (ad/2000-09-03), Document ad/2003-01-07, version 1.6.
14. Microsoft Research Foundations of Software Engineering Group. AsmL: The Abstract State Machine Language. `http://research.microsoft.com/foundations/AsmL/`.
15. The Object Managment Group (OMG). `http://www.omg.org`.

16. A. Pataricza, D. Varró, and G. Varró. Towards an xmi-based model interchange format for graph transformation systems. Technical report, Budapest University of Technology and Economics Department of Measurement and Information Systems, 2000.
17. Petri Net Markup Laguage (PNML). `http://www.informatik.hu-berlin.de/top/pnml`.
18. E. Riccobene and P. Scandurra. Towards an Interchange Language for ASMs. In Bernhard Thalheim Wolf Zimmermann, editor, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer, 2004.
19. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.
20. SDL (Specification and Description Language. ITU Reccomandation Z.100. `http://www.itu.int`.
21. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *J. Padberg (Ed.), UNIGRA 2001: Uniform Approaches to Graphical Process Speci0cation Techniques*, 2001.
22. OMG, The Unified Modeling Language (UML). `http://www.uml.org`.
23. OMG, UML 2.0 Superstructure Final Adopted specification. Document ptc/03-08-02. `http://www.uml.org/`.
24. OMG, XML Metadata Interchange (XMI) Specification, v1.2.
25. W3C, The Extensible Markup Language (XML).
26. Y. Gurevich, B. Rossman and W. Schulte. Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .

Appendix A – the AsmM-CS

In the EBNF grammar reported below, nonterminals are plain and literal symbols are enclosed in double quotes. In addition, words enclosed in angle brackets indicate a placeholder for a literal value that must be substituted with an actual value (e.g., `<DIGIT> ::= [`0" - "9"]").

The structural language

```
Asm ::= ( "asynchr" )? ( "asm" | "module"
) <ID> Header (( Initialization )* "default" Initialization (
Initialization )*)? Body ( "main" RuleDeclaration )?

Header ::= ( ImportClause )* ( ExportClause )? Signature

ImportClause ::= "import" <MOD_ID> (  "(" ( <ID_DOMAIN> |
<ID_FUNCTION> | <ID_RULE> ) ( "," ( <ID_DOMAIN> | <ID_FUNCTION> |
<ID_RULE> ) )* ")"  )?

ExportClause ::= "export" (  ( <ID_DOMAIN> | <ID_FUNCTION> |
<ID_RULE> ) ( "," ( <ID_DOMAIN> | <ID_FUNCTION> | <ID_RULE> ) )*
) | "*"

Signature ::= "signature" ":" ( Domain )* ( Function )*

Initialization ::= "init" <ID> ":" ( DomainInitialization )*
                           ( FunctionInitialization )*
                           ( AgentInitialization )*

FunctionInitialization ::= "function" <ID_FUNCTION> ( "("
VariableTerm "in" ( <ID_DOMAIN> | StructuredTD ) ( ","
VariableTerm "in" ( <ID_DOMAIN> | StructuredTD ) )* ")" )? "="
Term

DomainInitialization ::= "domain" <ID_DOMAIN> "=" Term

AgentInitialization ::= "<" ID_AGENT ">" ":" "agent" "<" (
<ID_RULE> | Rule ) ">"

Body ::= "definitions" ":" ( DomainDefinition )*
                           ( FunctionDefinition )*
                           ( RuleDeclaration )*
                           ( Axiom )*

FunctionDefinition ::= "function" <ID_FUNCTION> ( "(" VariableTerm
"in" ( <ID_DOMAIN> | StructuredTD ) ( "," VariableTerm "in" (
<ID_DOMAIN> | StructuredTD ) )* ")" )? "=" Term

DomainDefinition ::= "domain" <ID_DOMAIN> "=" Term
```

```
RuleDeclaration ::= "rule" ( "macro" )? <ID_RULE> ( "("
VariableTerm "in" ( <ID_DOMAIN> | StructuredTD ) ( ","
VariableTerm "in" ( <ID_DOMAIN> | StructuredTD ) )* ")" )? "="
Rule

Axiom ::= "axiom" "over" ( ID_DOMAIN | ( ID_FUNCTION  "(" (
<ID_DOMAIN> | StructuredTD )? ")" ) | ID_RULE ) ( "," ( ID_DOMAIN
| ( ID_FUNCTION  "(" ( <ID_DOMAIN> | StructuredTD )? ")" ) |
ID_RULE ) )* ":"  Term
```

The definitional language

```
Domain ::= (ConcreteDomain | TypeDomain)

ConcreteDomain ::= ( "dynamic" )? "domain" <ID_DOMAIN> "subsetof"
( <ID_DOMAIN> | StructuredTD )

TypeDomain ::= ( StructuredTD | EnumTD | AbstractTD | BasicTD  |
"anydomain" <ID_DOMAIN> )

BasicTD ::= "basic" "domain" <ID_DOMAIN>

AbstractTD ::= "abstract" "domain" <ID_DOMAIN>

EnumTD ::= "enum" "domain" <ID_DOMAIN> "=" "{" EnumElement ( "|"
EnumElement )* "}"

EnumElement ::= <ID_ENUM>

StructuredTD ::= ( ProductDomain | SequenceDomain | PowersetDomain
| BagDomain | MapDomain )

ProductDomain ::= "Prod" "(" ( <ID_DOMAIN> | StructuredTD ) ( ","
( <ID_DOMAIN> | StructuredTD ) )+ ")"

SequenceDomain ::= "Seq" "(" ( <ID_DOMAIN> | StructuredTD ) ")"

PowersetDomain ::= "Powerset" "(" ( <ID_DOMAIN> | StructuredTD )
")"

BagDomain ::= "Bag" "(" ( <ID_DOMAIN> | StructuredTD ) ")"

MapDomain ::= "Map" "(" ( <ID_DOMAIN> | StructuredTD ) "," (
<ID_DOMAIN> | StructuredTD ) ")"

Function ::= ( BasicFunction | DerivedFunction )

BasicFunction ::= ( StaticFunction | DynamicFunction )
```

```
DerivedFunction ::= "derived" <ID_FUNCTION> ":" ( ( <ID_DOMAIN> |
StructuredTD ) "->" )? ( <ID_DOMAIN> | StructuredTD )

StaticFunction ::= "static" <ID_FUNCTION> ":" ( ( <ID_DOMAIN> |
StructuredTD ) "->" )? ( <ID_DOMAIN> | StructuredTD )

DynamicFunction ::= ( "dynamic" )? ( "monitored" | "controlled" |
"shared" | "out" | "local" ) <ID_FUNCTION> ":" ( ( <ID_DOMAIN> |
StructuredTD ) "->" )? ( <ID_DOMAIN> | StructuredTD )
```

The language of terms

```
Term ::= ( Expression | ExtendedTerm )

Expression ::= or_xorLogicExpr ( "implies" or_xorLogicExpr | "iff"
or_xorLogicExpr )*

or_xorLogicExpr ::= andLogicExpr ( "or" andLogicExpr | "xor"
andLogicExpr )*

andLogicExpr ::= notLogicExpr ( "and" notLogicExpr )*

notLogicExpr ::= ( "not" includesExpr | includesExpr )

includesExpr ::= relationalExpr ( "in" relationalExpr | "notin"
relationalExpr )?

relationalExpr ::= additiveExpr ( ( "=" additiveExpr | "!="
additiveExpr | "<" additiveExpr | "<=" additiveExpr | ">"
additiveExpr | ">=" additiveExpr ) )*

additiveExpr ::= multiplicativeExpr ( "+" multiplicativeExpr | "-"
multiplicativeExpr )*

multiplicativeExpr ::= powerExpr ( "*" powerExpr | "/" powerExpr |
"mod" powerExpr )*

powerExpr ::= unaryExpr ( "^" unaryExpr )* unaryExpr ::= ( "-" "("
basicExpr ")" | basicExpr )

basicExpr ::= ( BasicTerm | DomainTerm | "(" Expression ")" )

BasicTerm ::= ( ConstantTerm | VariableTerm | FunctionTerm )

FunctionTerm ::= ( ID_AGENT "." )? <ID_FUNCTION> ( TupleTerm )?

LocationTerm ::= ( ID_AGENT "." )? <ID_FUNCTION> ( TupleTerm )?

VariableTerm ::= <ID_VARIABLE>
```

```
ConstantTerm ::= ( ComplexTerm | RealTerm | IntegerTerm |
NaturalTerm | CharTerm | StringTerm | BooleanTerm | UndefTerm |
EnumTerm )

ComplexTerm ::= <COMPLEX_NUMBER>

RealTerm ::= (  "+" | "-"  )? <REAL_NUMBER>

IntegerTerm ::= ( "+" | "-" ) <NUMBER> NaturalTerm ::= <NUMBER>

CharTerm ::= <CHAR_LITERAL> StringTerm ::= <STRING_LITERAL>

BooleanTerm ::= ( "true" | "false" ) UndefTerm ::= "undef"

EnumTerm ::= <ID_ENUM>

ExtendedTerm ::= ( ConditionalTerm | CaseTerm | TupleTerm |
CollectionTerm | VariableBindingTerm | RuleAsTerm | DomainTerm )

ConditionalTerm ::= "if" Term "then" Term ( "else" Term )? "endif"

CaseTerm ::= "switch" Term ( "case" Term ":" Term )+ ( "otherwise"
Term )? "endswitch"

TupleTerm ::= "(" Term ( "," Term )* ")"

CollectionTerm ::= ( SequenceTerm | SetTerm | MapTerm | BagTerm )

SequenceTerm ::= "[" ( Term ( ( "," Term )+  |  ( ".." Term ( ","
( <REAL_NUMBER> | <NUMBER> ) )? ) )? )? "]"

SetTerm ::= "{" ( Term ( ( "," Term )+  |  ( ".." Term ( "," (
<REAL_NUMBER> | <NUMBER> ) )? ) )? )? "}"

MapTerm ::= "{" ( "->" | ( Term "->" Term ( "," Term "->" Term )*
) ) "}"

BagTerm ::= "<" ( Term ( ( "," Term )+  |  ( ".." Term ( "," (
<REAL_NUMBER> | <NUMBER> ) )? ) )? )? ">"

VariableBindingTerm ::= ( LetTerm | FiniteQuantificationTerm |
ComprehensionTerm )

FiniteQuantificationTerm ::= ( ForallTerm | ExistUniqueTerm |
ExistTerm )

ExistTerm ::= "(" "exist" VariableTerm "in" Term ( ","
VariableTerm "in" Term )* ( "with" Term )? ")"
```

```
ExistUniqueTerm ::= "(" "exist" "unique" VariableTerm "in" Term (
"," VariableTerm "in" Term )* ( "with" Term )? ")"

ForallTerm ::= "(" "forall" VariableTerm "in" Term ( ","
VariableTerm "in" Term )* ( "with" Term )? ")"

LetTerm ::= "let" "(" VariableTerm "=" Term ( "," VariableTerm "="
Term )* ")" "in" Term "endlet"

ComprehensionTerm ::= ( SetCT | MapCT | SequenceCT | BagCT )

SetCT ::= "{" Term "|" VariableTerm "in" Term ( "," VariableTerm
"in" Term )* ( "with" Term )? "}"

MapCT ::= "{" Term "->" Term "|" VariableTerm "in" Term ( ","
VariableTerm "in" Term )* ( "with" Term )? "}"

SequenceCT ::= "[" Term "|" VariableTerm "in" Term ( ","
VariableTerm "in" Term )* ( "with" Term )? "]"

BagCT ::= "<" Term "|" VariableTerm "in" Term ( "," VariableTerm
"in" Term )* ( "with" Term )? ">"

DomainTerm ::= ( <ID_DOMAIN> | StructuredTD )

RuleAsTerm ::= "<<" Rule ">>"
```

The language of rules

```
Rule ::= ( BasicRule | TurboRule | TermAsRule | DerivedRule )

TermAsRule ::= ( FunctionTerm | VariableTerm )

BasicRule ::= ( SkipRule | UpdateRule | MacroCallRule | BlockRule
| ConditionalRule | ChooseRule | ForallRule | LetRule | ExtendRule
)

SkipRule ::= "skip"

UpdateRule ::= ( LocationTerm | VariableTerm ) ":=" Term

BlockRule ::= "par" Rule Rule ( Rule )* "endpar"

ConditionalRule ::= "if" Term "then" Rule ( "else" Rule )? "endif"

ChooseRule ::= "choose" VariableTerm "in" Term ( "," VariableTerm
"in" Term )* "with" Term "do" Rule ( "ifnone" Rule )?

ForallRule ::= "forall" VariableTerm "in" Term ( "," VariableTerm
"in" Term )* "with" Term "do" Rule

LetRule ::= "let" "(" VariableTerm "=" Term ( "," VariableTerm "="
Term )* ")" "in" Rule "endlet"

MacroCallRule ::= <ID_RULE> "[" ( Term ( "," Term )* )? "]"

ExtendRule ::= "extend" <ID_DOMAIN> "with" VariableTerm ( ","
VariableTerm )* "do" Rule

TurboRule ::= ( SeqRule | IterateRule | TurboReturnRule |
TurboCallRule | TurboLocalStateRule | TryCatchRule )

SeqRule ::= "seq" Rule Rule ( Rule )* "endseq"

IterateRule ::= "iterate" Rule "enditerate"

TurboCallRule ::= <ID_RULE> "(" ( Term ( "," Term )* )? ")"

TurboReturnRule ::= ( LocationTerm | VariableTerm ) "<-"
TurboCallRule

TurboLocalStateRule ::= DynamicFunction "[" Rule "]" (
DynamicFunction "[" Rule "]" )* Rule

TryCatchRule ::= "try" Rule "catch" ( LocationTerm | VariableTerm
) ( "," ( LocationTerm | VariableTerm ) )* Rule

DerivedRule ::= ( BasicDerivedRule | TurboDerivedRule )

BasicDerivedRule ::= CaseRule

CaseRule ::= "switch" Term ( "case" Term ":" Rule )+ ( "otherwise"
Rule )? "endswitch"

TurboDerivedRule ::= ( RecursiveWhileRule | IterativeWhileRule )

RecursiveWhileRule ::= "whilerec" Term "do" Rule

IterativeWhileRule ::= "while" Term "do" Rule
```

Final terminals

```
ID_AGENT ::= <ID_FUNCTION>

<ID_VARIABLE> ::= "$" <LETTER>  ( <LETTER> | <DIGIT> )*

<ID_ENUM> ::= [ "A" - "Z" ] [ "A" - "Z" ] ( "_" | [ "A" - "Z" ] |
<DIGIT> )*>
```

```
<ID_DOMAIN> ::= [ "A" - "Z" ] ( "_" | [ "a" - "z" ] | <DIGIT>)* >

<ID_RULE> ::= "r_" ( <LETTER> | <DIGIT> )+

<ID_FUNCTION> ::= [ "a" - "z" ] ( <LETTER> | <DIGIT> )*

<ID> ::= <LETTER> ( <LETTER> | <DIGIT> )*

<MOD_ID> ::= ( <LETTER> | "." | "\" | "//" ) ( <LETTER> | <DIGIT>
| "." | "\" | "//" | ":" )*

<NUMBER> ::= ( <DIGIT> )+

<REAL_NUMBER> ::= ( <DIGIT> )+  "."  ( <DIGIT> )+

<COMPLEX_NUMBER> ::= ( ( ["+", "-"] )?  ( <DIGIT> )+  ( "."  (
<DIGIT> )+ )? )?  ( ["+", "-"] )?  "i"  ( ( <DIGIT> )+  ( "."  (
<DIGIT> )+ )? )?

<CHAR_LITERAL> ::=  " ' " (   ( ~[ " ' ", "\\", "\n", "\r" ] ) | (
"\\" (  [ "n", "t", "b", "r", "f", "\\", " ' ", "\"" ]
                      | [ "0"-"7" ]  (  [ "0"-"7" ] )?
                      | [ "0"-"3" ]  [ "0"-"7" ]  [ "0"-"7" ] ) ) )*  " ' "

<STRING_LITERAL> ::=  " \" " (   ( ~[ " \" ", "\\", "\n", "\r" ] )
| ( "\\" ( [ "n", "t", "b", "r", "f", "\\", " ' ", "\"" ] |
["0"-"7" ]  (  [ "0"-"7" ] )? | [ "0"-"3" ] [ "0"-"7" ] [ "0"-"7"]
) ) )*  " \" "

<LETTER> ::= [ "_" , "a" - "z" , "A" - "Z" ]

<DIGIT> ::= [ "0" - "9" ]
```