Metamodelling a Formal Method:
Applying MDE to Abstract State Machines

Angelo Gargantini[1], Elvinia Riccobene[2], Patrizia Scandurra[2]

[1] Università di Bergamo, Dip. di Ingegneria Informatica e Gestionale, V.le Marconi, 5 – 24044 Dalmine, Italy
[2] Università di Milano, Dip. di Tecnologie dell'Informazione, via Bramante, 65 – 26013 Crema, Italy

Abstract This paper presents the AsmM, a *metamodel* for Abstract State Machines developed by following the guidelines of the Model Driven Engineering. The AsmM represents concepts and constructs of the ASM formal method in an abstract way, it is endowed with a standard visual notation, and it is intended easy to learn and understand by practitioners and students.

From the AsmM a concrete syntax is also proposed and a standard interchange format for a systematic integration of a number of loosely-coupled ASM tools is derived. The metamodelling advantages for tool interoperability are shown by referring to the experience in making the ATGT, an existing tool supporting test case generation for ASMs, compliant to the AsmM.

Contents

1 Introduction

The success of the Abstract State Machines (ASMs) as a system engineering method able to guide the development of hardware and software systems, from requirements capture to their implementation, is nowadays widely acknowledged [16,15]. The increasing application of the ASM formal method for academic and industrial projects 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 [45,22,26,18,54], and execution engines for simulation and testing purposes [46,10,9,17,27].

Since each tool usually covers well only one aspect of the whole system development process, at different steps modelers and practitioners would like to switch tools to make the best of them while reusing information already entered about their models. As already discussed in [43], a standard interchange format is of particular interest for the ASMs community, since ASM tools have been usually developed by individual research groups, are loosely coupled and have syntaxes strictly depending on the implementation environment (compare, for example, simulation environments like AsmGofer [46], ASM-SL [17], XASM [9], and AsmL [10]). 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.

A general framework for a wide interoperability of ASM tools, based on *metamodelling*, is suggested by the Model-Driven Engineering (MDE) [13,33,36], an emerging approach for software development and analysis where models play the fundamental role of first-class artifacts. In MDE, metamodelling is intended as a modular and layered way to endow a language or a 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 metamodel-based abstract syntax, an easy to learn concrete syntax, possibly graphic, a well-founded semantics, and a XML-based [57] model interchange format.

In [14], the problem of tool interoperability in the context of bug tracing systems is tackled through *meta-models* 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 about bug-tracking. Model transformations based on these metamodels to the pivot and from the pivot to the metamodels are defined and implemented. The approach allows a relatively easy addition of new tools that have to operate with the existing ones by: creating the associated metamodel; building the bridge (composed of two transformations) between this metamodel and the logical *pivot* metamodel; making the XML injector/extractor for this metamodel.

In this paper, we introduce the *Abstract State Machine Metamodel* (AsmM), a metamodel for ASMs [11]. The AsmM can be seen as the *pivot metamodel* 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.

The AsmM comprises: a metamodel definition (the *abstract syntax*) conforming to the Meta Object Facility (MOF)[2] metalanguage and representing in an abstract and visual way the ASMs related concepts and constructs (abstract machines, signatures, terms, rules, etc.) as described in [16]; an *interchange syntax*, i.e a standard XMI-based [55] format automatically derived from the AsmM for the interchange of ASM models among tools; and a *concrete syntax*, namely an EBNF (extended Backus-Naur Form) grammar derived from the AsmM as textual notation to write ASM models conforming to the AsmM. For the *AsmM semantics*, we assume the ASM semantics in [16].

Tool interoperability is only one of the benefits of using the metamodelling approach. We identify other two significant advantages. First, a metamodel could serve as standard representation of a formal notation, establishing a common terminology to discriminate pertinent elements to be discussed, and therefore, helps to communicate understandings, especially if − as in the case of the ASMs − the formal method is still evolving and the community is too much heterogeneous to easily come to an agreement on the further development of the method. Second, people often claim that formal methods are too difficult to put in practice due to their mathematical-based foundation. In this direction an abstract and visual representation, like the one provided by a MOF-compliant metamodel, delivers a more readable view of the modelling primitives offered by a formal method, especially for people, like students, who do not deal well with mathematics but are familiar with the standard MOF/UML. Indeed, the AsmM can be considered a complementary approach to [28,16] for the presentation of the ASMs.

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 defining 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, textual or graphical or both, for various scopes like graphical rendering, model interchange, standard encoding in programming languages, and so on.

The paper is organized as follows. Basic metamodelling notions are sketched out in section 2 where the

MOF and XMI standards are also presented. Section 3 introduces basic concepts underlying ASMs. Sections 4 - 10 present the AsmM in a modular and incremental way. In Section 12 some derivative artifacts obtained from the AsmM abstract syntax are introduced: the AsmM concrete syntax and the MOF-to-EBNF mapping rules applied to derive it; the AsmM-specific Java Metadata Interfaces used to access and manipulate models; the AsmM-specific XMI interchange format; the parser which processes the ASM models written in the AsmM concrete syntax and creates the corresponding AsmM instances. The role of this parser and of the XMI interchange format is discussed in Section 13. Section 14 shows how an existing tool for ASMs can be easily modified to make it AsmM-compliant. Related and future work are given in sections 15 and 16, respectively.

2 Metamodelling for Language Definition

This section outlines some basic notions on *metamodelling* and some MDE standards to allow a reader not familiar with these concepts to understand the AsmM and the derivative artifacts obtained from the AsmM. Readers expert in this area can skip the section.

In the MDE, where models are first class entities and any software artifact is a model or a model element, languages or formalisms enabling to express models are defined in terms of *metamodels*.

A metamodel describes the various kinds of model elements, and the way they are arranged, related, and constrained. Metamodel elements (or *metaelements*) provide a *typing scheme* for model elements expressed by the meta relation between a model element and its metaelement. It is said that a model element is *typed* by its metaelement. A model is said *to conform* to its metamodel if and only if each model element has its metaelement defined within the metamodel. In the same way models are defined in conformance with their metamodel, metamodels are defined by means of a common base formalism called *meta-metamodel* or *metalanguage*. A metamodel conforms to the meta-metamodel if and only if each of its elements has its metaelement defined in the meta-metamodel.

Among the objectives pursued by the MDE approach, one may list (i) the separation from system-neutral descriptions and platform dependent implementations, (ii) the identification, precise specification, separation and combination of specific aspects of a system under development with domain-specific languages (DSLs), and, more importantly, (iii) the establishment of precise *bridges* (or *projections*) between these different languages in a "global framework" to automatically execute model transformations. The importance of *metamodelling* is that it settles such a "global framework" to enable otherwise dissimilar languages (representing different domains) to be used in an interoperable manner in different technical spaces [34].

The OMG's Model Driven Architecture (MDA) [35] was historically one of the original proposals to support MDE principles. The primary goal of the MDA initiative is preserving the IT investments of companies through the constant and rapid evolution of platforms. The idea consists into keeping separate the design of the system functionality − the platform independent model (PIM), usually written in a general purpose language like UML − from its implementation on a specific platform − the platform specific model (PSM). Moreover, PSMs should be obtained by refining the original PIMs through automatic transformation bridges.

At the core of MDA, there is a set of OMG standards like the Meta-Object Facility (MOF), the UML (Unified Modeling Language)[50], XMI (XML Metadata Interchange) [55,56], the OCL (Object Constraint Language) [37], QVT (Query/Views/Transformations) [42], to name a few.

The classical OMG's metamodelling framework is based on an four-layered architecture ([2], Sect. 2.2.1), where the relation between a model and its metamodel is characterized by a *instanceOf* relation rather than the *conformsTo* relation as proposed in MDE:

M_0 (Data) The *data* (or *information*) *layer* comprises data of the real world that we wish to describe, i.e. it refers to actual instances of information.

M_1 (Model) The *model* (or *metadata*) *layer* comprises metadata that describes (in format and semantics) data in the information layer. This is the level at which system modelling takes place.

M_2 (Metamodel) The *metamodel* (or *meta-metadata*) *layer* contains the language specification, or the metamodel. It comprises the description of the structure and semantics of metadata in an abstract way.

M_3 (Meta-metamodel) The *meta-metamodel layer* comprises the description of the structure and semantics of meta-metadata. It is the common meta-language for defining different kinds of metamodels.

The MOF is the OMG proposal for metamodel definition, namely a meta-metamodel. It resides at layer M_3 of the metamodelling architecture and, as there is no higher abstraction layer, it is defined in terms of itself.

One of the best-known metamodels is the UML metamodel [50], which stays at layer M_2. The proposed AsmM lies at layer M_2 as well. Any ASM model conforms to the AsmM lies at layer M_1 and when initialized (data are supplied) it lies at layer M_0 (see Fig. 1).

The MOF is able to capture object organizations as well as other organizations. As model of a language or formalism, a metamodel is the result of a process of abstraction, classification, and generalization on the language domain, not necessarily object oriented.

Several *projections* towards other technical spaces are associated to the MOF and may be seen as additional facilities of level M_3 to handle models. Among these:
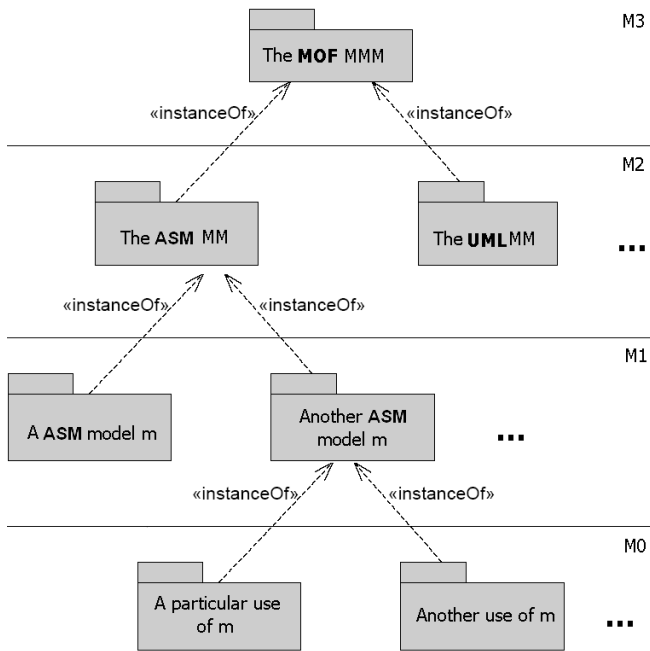
Figure 1 Four-layered architecture

– XMI (XML Metadata Interchange) [55,56] for bridging with the XML Document space for interchange of models between tools;
– JMI (Java Metadata Interface) [32] for bridging with the Java space, to access model elements in a metadata repository (where models are understood to be instances of some particular metamodel) for model construction, discovery, traversal, and update;
– CMI (CORBA Metadata Interface) [19] for bridging with the middleware CORBA space;
– HUTN (Human Usable Textual Notation) [30] and the anti-Yacc proposal by DSTC [21], for bridging with the textual flat-file technical space.

Many other standard projections could be useful as well, like the one on rendering spaces like SVG [5], for example.

2.1 The Metamodelling Process using MOF

The MOF defines a set of modelling constructs that a modeler can use to define and manipulate metamodels. The MOF concepts described below refers to the version 1.4 [2] that we used to define a metamodel for the ASMs formalism. A reader familiar with the UML is already familiar with the most important concepts of the MOF[1]. These include *types* (classes, primitive types, and type constructors), *associations*, *packages*, *constraints*.

Classes are type descriptions of MOF meta-objects [2]. Classes defined at the M_2 layer (i.e. into a metamodel)

have their instances with an object identity, state, and behaviour at the M_1 layer. Structural features of classes can be described by *Attributes* and *Operations*. Derived attributes (preceded by a "/") are determined by the object state. Operations do not actually specify the behaviour or the methods implementing that behaviour, but only the name and various signatures by which the behaviour is invoked.

Classes can inherit their structure from other classes by *Generalization* relations (the common *class inheritance* concept of the object-oriented paradigm). *Abstract classes* are classes that must not have instances. A *singleton class* is one for which only one instance may exist. A class can contain references to associations (see below). Classes can also contain *Exceptions*, *Constants*, and other elements (see [2] for more details).

Primitive types and constructors represent attributes and operation parameter values which do not have an object identity. Primitive types stand for *primitive* data like `Boolean`, `Integer`, `String`, etc. Type *constructors* allow modelers to introduce more complex types like enumeration types, structure types, collection types, and alias types.

Associations are the primary construct for expressing binary relationships between class instances. At the M_1 layer, a binary M_2 layer association defines relationships, called *links*, between pairs of instances of the related classes. Both ends of an association may specify a multiplicity which indicates the number of objects that, at run-time, may participate in the association. Common multiplicity values are the following:
– 1: exactly one object will participate;
– a number $n > 0$: exactly n objects will participate;
– 0..1: 0 or 1 object will participate;
– $n..m$: the number of participating objects is between n and m;
– *: zero or more objects will participate.

A name string, referred as *role name*, near the end of an association indicates the role played by the class attached to the association end. The role name is optional but not suppressible. Associations ends can also specify *aggregation* semantics[2] and structural constraints on the *ordering* (if the multiplicity is greater than one, then the set of related elements can be ordered or – the default case – unordered), *navigability* (whenever navigation is supported in a given direction), and *uniqueness*.

Association References An instance of a class may be defined to be "aware" of being in a relationship with other

---

[1] The MOF 1.4 (and prior versions) supports basic object oriented concepts, most of which are a subset of the UML.

[2] An association may represent an aggregation (i.e., a whole/part relationship). An aggregation can be also composite to denote a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it.

objects via an association. This awareness is expressed as an *association reference* contained as named property in the class, and results in link navigation and update operations being made available in the MOF object's interface. The referenced association end determines what ends of what kinds of links an instance is "aware" of.

Packages are MOF constructs to group elements (classes and associations) within a metamodel for partitioning and modularization purposes. Packages can be composed by importing other packages or by inheriting from them. They can also be nested, to provide information hiding.

Constraints specify consistency rules, called *well-formedness rules*, to additionally argument models described by the constructs in the metamodel. Constraints are usually written as formula in the Object Constraint Language (OCL) [37] and can be evaluated against the metamodel to decide if a given model conforms to it.

The above metamodelling constructs are sufficient to define the so called *abstract syntax* of a language, i.e. the structure of a language, separated from its *concrete notation*. To be precise, a metamodel-based language definition is in general articulated into the specification of:

- an *abstract syntax*, defined by a MOF-compliant metamodel, for the definition of the modelling constructs of the language, plus well-formedness rules in OCL to capture the *static semantics* of the language;
- a *concrete syntax* in which to write models, generally based on a diagrammatic notation (shapes, connectors, layout, etc.) or a textual syntax, or both;
- the *semantics*, i.e. the abstract logical space in which models find their meaning[3].

2.2 Model Constraints using OCL

The Object Constraint Language [37] is a language of typed expressions, used to specify constraints on objects in the MOF/UML. A constraint is a valid OCL expression of type Boolean that states a restriction on one or more values of (part of) an object-oriented model or system. There are a number of constraint types: class *invariants*, i.e. constraints that must always be met by all instances of a specified class, and *pre/post-conditions* on operations. The mostly used here in class diagrams are invariants.

An OCL expression can be built using the basic OCL types (Boolean, Integer, Real, ...), the MOF/UML data types, and predefined operators, as well as classes (considered as OCL types) from the MOF/UML model, associations, attributes and query operations defined inside classes, i.e. operations that simply return a value but do not change the state of the system (since all OCL expressions must be side effect free).

There are different operations to treat and analyze classes at the meta-level. The mostly used here are the following:

- the operation `o.oclIsTypeOf(T)` returns true when a given instance `o` is a direct instance of a certain class `T` (and not of one of its subclasses or of other classes);
- the operation `o.oclAsType(C)::C` casts the object `o` to an instance of class `C`;
- the operation `C.allInstances()` returns all instances of a given class `C`.

Any OCL expression can navigate through the model by following the "path" of associations. The constraint `context` is the starting point for navigating and the object(s) on the other side of the association are identified by the role name. In an OCL expression one may therefore put constraints also on the associated object(s) or on attributes of the associated object(s). To this end, there are a large number of predefined operations on collections, e.g. `isEmpty`, `size`, `includes`, `forAll`, `exists`, etc. An arrow, instead of a dot, before the operation indicates the use of one of these collection operations.

2.3 Model Interchange using XMI

The XML Metadata Interchange (XMI) [55,56] is an OMG standard developed as a tool-independent medium for exchanging models at any level of the metamodelling hierarchy. It maps the MOF to the W3C's eXtensible Markup Language (XML) [57] defining rules to represent *serialized* MOF concepts in XML tags.

In XMI, a XML Document Type Definition (DTD) or a schema can be defined at any level M_x of the metamodelling hierarchy, using rules appropriate for that layer (see Fig. 2). This DTD or schema can then be used to transport data at layer M_{x-1} in the form of XML documents that conform to the DTD or schema defined at layer M_x. A DTD or schema is defined by the OMG at the M_3 for the MOF itself, and it is used to transport metamodels at layer M_2 in XML documents conforming to the MOF DTD or schema.

Whenever a language or formalism is specified in terms of a MOF-compliant metamodel, the MOF enables a standard way to generate an interchange format for models in that language; indeed, a DTD or a schema is defined at layer M_2 for the metamodel of the given language. This metamodel-specific DTD or schema is then used for models written in the given language at layer M_1 in XML format.

---

[3] The MOF lacks – as often signaled in literature by proposals which aim at formalizing the UML semantics – of any formality to express the *dynamic semantics* of languages, which is generally given in natural language or, more formally, through the use of formal methods.
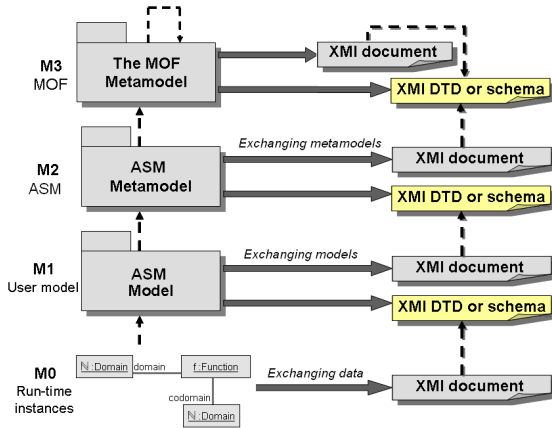
Figure 2 XMI in the four-layer metamodelling hierarchy

The capability of XMI (and of XML, in general) to communicate both metaelements (*tags*) and model elements (*element content*) in the same document, enables applications to easily recover information about models via their metaelements, making the XMI format an optimal solution for interoperability.

In Section 12.2 we present the XMI format for the AsmM, and in Section 13 we discuss its use for interchanging ASM models among tools.

2.4 Java Metadata Interface

Within the open Java Community Process (JCP) [31], a number of Java specifications are currently under development that represent formal mappings of OMG's MDA standards on Java technologies. The most important one is the *Java Metadata Interface* (JMI) [32]: a pure Java API, which provides a natural and easy-to-use mapping from the MOF to the Java programming language for creating, storing, accessing, and interchanging metadata. Applications and tools endowed with MOF-compliant metamodels, can have their JMI automatically generated. Further, metamodel and metadata interchange via XML is enabled by JMI's use of the XMI specification. Java applications can create, update, delete, and retrieve information contained in a JMI compliant metadata service.

In Section 12.1 we show how to use JMI to access AsmM data.

3 Abstract State Machines

Abstract State Machines (ASMs) are a system engineering method that guides the development of software and embedded hardware-software systems seamlessly from requirements capture to their implementation.

The three constituents of the ASM method are: the concept of *abstract state machines* for system specification, the *ground model* method to faithfully capture

informal requirements through precise but concise high-level system blueprints (*system contracts*) formulated in domain-specific terms, using an application-oriented language which can be understood by all stakeholders, and the *refinement method* for turning ground models by incremental steps into executable code in a traceable and documented way, providing explicit description of the software structure and of the major design decisions.

Even if the ASM method comes with a rigourous scientific foundation [28,16], 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*.

Within a single conceptual framework, ASMs allow a nowadays widely-requested *modelling technique* which integrates dynamic (*operational*) and static (*declarative*) descriptions. *Analysis techniques* that combine *validation* (by simulation and testing) and *verification* can be performed *at any desired level of detail*. A number of ASM tools have been developed for model simulation [46, 10,9,17], model-based testing [27], verification of model properties by proof techniques (i.e. by KIV [45] or PVS [22,26]), or model checkers [18,54].

A complete introduction on the ASM method can be found in [16], together with a presentation of the great variety of its successful application in different fields such as: definition of industrial standards for programming and modelling languages, design and re-engineering of industrial control systems, modelling e-commerce and web services, design and analysis of protocols, architectural design, language design, verification of compilation schemas and compiler back-ends, etc.

For our purposes, we quote here only the essential working definitions of the ASMs, i.e. in a form which justifies their intuitive understanding as "pseudo-code" over "abstract data", as stated in [16], and we show how to turn these definitions into metamodelling constructs.

For a more detailed mathematical definition of the semantics of the ASMs, and explanation of the ASM *ground models* and the notion of ASM *refinement* we refer the reader to [16].

4 The ASM Metamodel (AsmM)

According to the metamodel-based language definition guidelines, the specification of an *Abstract State Machines Metamodel* (AsmM) [11] comprises:

- an *abstract syntax*, i.e. a MOF-compliant metamodel and OCL constraints (see Sect. 5 - 10) representing in an abstract (and visual) way concepts and constructs of the ASM formalism as described in [16];
- a *concrete syntax* Asm^2L (AsmM Language), namely an EBNF (Extended Backus-Naur Form) grammar (see Sect. 12.3) derived from the AsmM as a textual notation to be used by modelers to effectively write ASM models in a textual form;
- an *interchange syntax*, i.e. a standard XMI-based format automatically derived from the AsmM, for the interchange of ASM models (see Sect. 13).

For the metamodel *semantics*, we adopt the ASM semantics given in [16].

In the following sections, a complete meta-level representation of ASMs concepts is formulated, based on the MOF metalanguage. We develop the metamodel in a modular way reflecting the natural classification of abstract state machines in Basic ASMs, Turbo ASMs, and Multi-Agent (Sync/Async) ASMs. 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 meaningfully connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams.

Further concepts which enrich and complete the specification of the AsmM (like particular forms of domains, special terms and derived rule schemes of an ASM) are reported in Section 11, while a standard AsmM library containing pre-defined ASM domains and functions (instances of the AsmM classes *Domain* and *Function*, see Sections 5.1 and 5.2) can be found in the AsmM website [11].

We have drawn the AsmM with the Poseidon UML (v.3.0) [41] empowered with an ancillary tool UML2MOF which transforms UML models to MOF 1.4 and is provided by the MDR Netbeans framework [4].

5 Metamodelling ASM States

ASMs extend the Finite State Machines replacing unstructured "internal" control states by states comprising arbitrary complex data. An ASM *state* models a machine state, i.e. the collection of elements and objects the machine "knows", and the functions and predicates it uses to manipulate them. Mathematically, a *state* is defined as an algebraic structure, where data come as abstract objects, i.e. as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions*) and *predicates* (attributes or relations). The set of function and domain names (declaration) is indicated as the *signature* or *vocabulary* of the ASM, while the collection of all state elements is called the *superuniverse* of the state. Part of the superuniverse is the set *reserve* which is used to increase, providing new elements whenever needed, the working space of an ASM. Usually it is supposed to be infinite and to be part of the state, but without any structure.

For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used. Without loss of generality we usually treat predicates as characteristic functions and constants as 0-ary functions. Partial functions are turned into total functions by interpreting $f(x) = undef$ with a fixed special value *undef* as $f(x)$ being undefined. The reader who is not familiar with this notion of structure may view a state as a "database of functions" (read: a set of function tables).

Pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1, \ldots, v_n)$, which is formed by a list of dynamic parameter values v_i of whatever type, are called *locations*. They represent the abstract ASM concept of basic object containers (memory units), which abstracts from particular memory addressing and object referencing mechanisms.

In this section we describe how to formalize domains, functions and terms in a MOF-compliant metamodel.

5.1 Metamodelling Domains

The abstract[4] class *Domain* represents the ASM notion of domain (or universe). The class is subclassed by *TypeDomain* and ConcreteDomain (see Fig. 3).

In practical applications, the superuniverse of an ASM state can be thought as partitioned into smaller universes. In the AsmM, these smaller super domains are called *type-domains* and they are represented by the abstract class *TypeDomain*. All user-named sub-domains of type-domains are represented by the class ConcreteDomain. The association end with role name typeDomain between classes *TypeDomain* and ConcreteDomain binds a concrete domain to its corresponding type-domain. A concrete domain must be explicitly declared in the signature of a given ASM, and the boolean attribute isDynamic specifies if the domain is *static* (never changes), or *dynamic* (its content can change during the computation by effect of transition rules). By default, a concrete domain is considered static (isDynamic=false).

---

[4] The name of an abstract class is shown in *Italics*. An *abstract class*, in contrast to an ordinary (concrete) class, cannot be instantiated. Typically, instances are created from its concrete sub-classes. The generalization relationship is shown as a line with an hollow triangle between the sub-class and the super-class with the arrowhead pointing to the super-class.
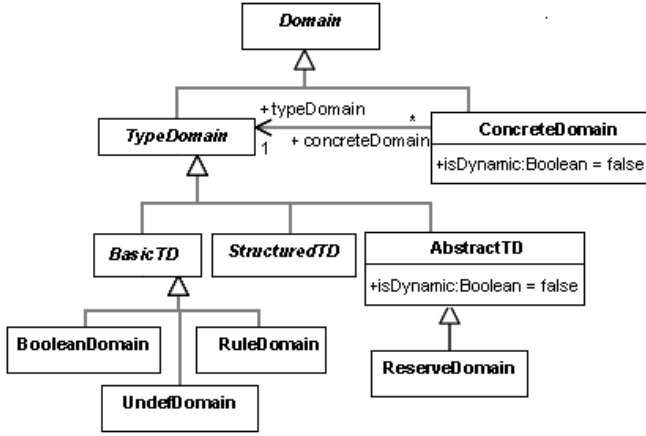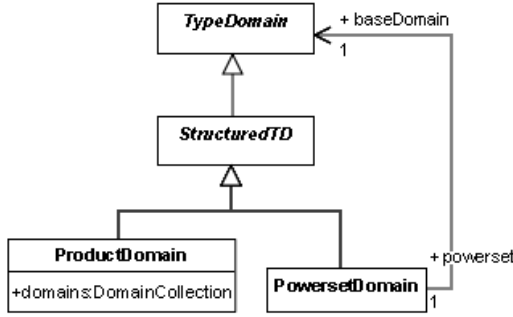
Figure 3 Domains (Part 1)

Figure 4 Domains (Part 2) – Structured domains

```
         context BooleanDomain inv:
D1:          allIstances->size()=1
         context RuleDomain inv:
D2:          allIstances->size()=1
         context UndefDomain inv:
D3:          allIstances->size()=1
         context ReserveDomain inv:
D4:          allIstances->size()=1
D5:          isDynamic = true
```

Table 1 Domain Constraints

Figure 5 Functions

The class *TypeDomain* is further classified in: *BasicTypeDomain*, for primitive data values; *StructuredTypeDomain*, representing type-domain constructors for structured data (like finite maps, sets and tuples); AbstractTypeDomain, modeling user named domains whose elements have no precise structure. Abstract type domains can be (by the boolean attribute isDynamic, which is false by default) *static*, i.e. never change, or *dynamic*, i.e. new elements can be added during the computation by effect of transition rules (see the *extend rule* in Sect. 6).

As basic type-domains, we have: BooleanDomain, representing the set $\{true, false\}$; RuleDomain, whose unique instance is the universe of all transition rules; and UndefDomain having as instance the set $\{undef\}$. This last set is assumed to be subset of every other type-domain or concrete-domain. All these three classes are singleton, and the OCL constraints D1, D2, and D3 in Tab. 1 assure this.

The subclass ReserveDomain of AbstractTypeDomain is a singleton class (D4) whose unique instance represents the notion of ASM *reserve*. It is dynamic, since we assume it is updated automatically upon (and only by) execution of an *extend rule* (see Sect. 6) as stated by the OCL constraint D5.

The *StructuredTypeDomain* is subclassed as shown in Fig. 4. The class ProductDomain represents the Cartesian product of two or more type-domains listed in th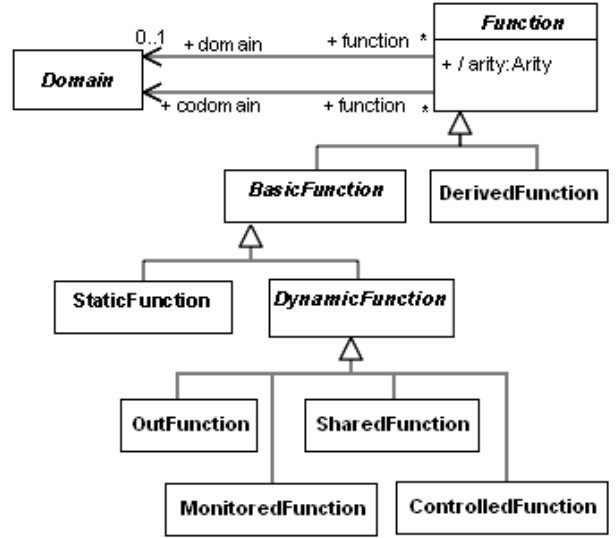e attribute domains; the class PowersetDomain represents the powerset of a type-domain specified by the association end baseDomain.

Section 11.0.1 introduces several other domains: basic type-domains for primitive data values like reals, integers, naturals, strings, etc., structured type-domains for sequences, bags and maps, and enum-domains.

5.2 Metamodelling Functions

The abstract class *Function* and its hierarchy (see Fig. 5) models the notion of function and function classification in ASMs. Functions are systematically distinguished between *basic* functions which are taken for granted (typically those forming the basic signature of an ASM) and derived ones (auxiliary functions coming with a specification or computation mechanism given in terms of basic functions), together with a classification of basic functions into static and *dynamic* ones and of the dynamic ones into monitored (only read), controlled (read and write), shared and output (only write) functions.

Static Functions never change during any run of the machine so that their values for given arguments do not depend on the states of the machine. Whether

the meaning of these functions is determined by a mere signature definition, or by axiomatic constraints, or by an abstract specification, or by an explicit or recursive definition, depends on the degree of information-hiding the specifier wants to realize. `Dynamic Functions` may change as a consequence of agent actions (or *updates*, see definition in Sect. 6) or by the *environment*, so that their values may depend on the states of the machine. Static 0-ary functions represent *constants*, whereas with dynamic 0-ary functions one can model *variables* of programming (not to be confused with logical variables, see Sect. 5.3).

`Controlled Functions` are dynamic functions which are directly updatable by and only by the machine instructions (better called *transition rules*, see Sect. 6). Therefore, these functions are the ones which constitute the internally controlled part of the dynamic state of the machine; they are not updatable by the environment (or more generally by another agent in the case of a multi-agent machine).

`Monitored Functions` (also called *in* functions) are dynamic functions which are read but not updated by a machine and directly updatable only by the environment (or more generally by other agents). These monitored functions constitute the externally controlled part of a machine state. As with static functions, the specification of monitored functions is open to any appropriate method. The only (but crucial) assumption made is that in a given state the values of all monitored functions are determined.

Combinations of internal and external control are captured by interaction or `Shared Functions` that can be read and are directly updatable by more than one machine (so that typically a protocol is needed to guarantee consistency of updates).

`Out Functions` are updated but not read by a machine and are typically monitored by other machines or by the environment.

Function domain and codomain are defined by the association ends `domain` and `codomain` to *Domain*. The `arity` of a function is derived from the function definition (Tab. 2). A constant is a function which is not linked to any instance of the class *Domain* via the association end `domain`, and its arity is 0 (F1). If the function domain is a Cartesian product of n domains then the arity of the function is equal to n (F2). If the function domain is a concrete domain with type-domain a Cartesian product of n domains, then the arity of the function is again equal to n (F3). Otherwise the arity of the function is 1 (F4).

5.3 Metamodelling Terms

As first-order structures, ASMs admit the use of predicate logic terms (variables, constants and function applications) and formulae. Terms can be interpreted in an ASM state if a *variable assignment* is provided, i.e. if ele-
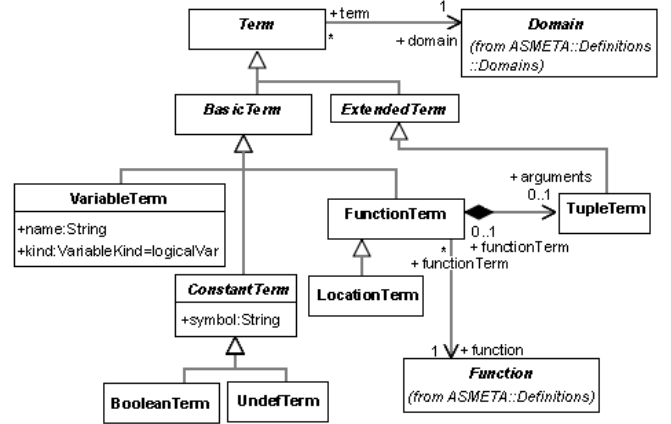

Figure 6 Terms

ments of the super-universe are assigned to the variables of the terms.

In the metamodel, we introduce the class *Term* (see Fig. 32) to model the concept of first-order logic terms. The association between classes *Term* and *Domain* specifies the domain to which the value of a term belongs.

We define an OCL query `compatible()` (Tab. 3) between two domains D_1 and D_2 to check wether the substitution of a term with domain D_1 by a term with domain D_2 is syntactically correct. A domain *self* is *compatible* with another domain d if and only if they are equals, or d is the `UndefDomain`[5], or if *self* is a concrete domain with type domain d or vice-versa, or if both *self* and d are both `PowersetDomain` and their `baseDomains` are compatible, or if they are `ProductDomain` and their component domains are compatible.

Two terms are *compatible* if and only if their domains are compatible (T1 in Tab. 4).

The class *Term* is subclassed by *BasicTerm*, for representing variables, constants and functions terms, and *ExtendedTerm*, for structured terms like sets, tuples, etc. (see Fig. 32).

The subclass *ConstantTerm* of *BasicTerm* models *constants*. It is slit in `BooleanTerm`, for the constant boolean terms *true* and *false* (T2, T3,T4), and `UndefTerm`, a singleton class whose unique instance represents the value *undef* (T5,T6,T7).

The class `FunctionTerm`, subclass of *BasicTerm*, represents a function application $f(t_1, \ldots, t_n)$ where the function f is given by the association end `function` and $(t_1, \ldots, t_n)$ is a tuple of terms specified by the association end `arguments` with the `TupleTerm` class (see below).

For function terms, we need to guarantee that the domain of a function term is equal to the associated function codomain (T8), and that if the associated function arity is greater than 0, the function term is linked to a `TupleTerm` instance which specifies the actual arguments of the function application (T9). Note that, the

---

[5] Note that if *self* is *undef*, then d must be *undef* as well to be equal; i.e. the relation is not symmetric.

```
     context Function inv :
F1:  if domain->isEmpty() then arity = 0
F2:  else if domain.oclIsTypeOf(ProductDomain)
        then arity = domain.oclAsType(ProductDomain).domains.items->size()
F3:     else if domain.oclIsTypeOf(ConcreteDomain) and
                 domain.oclAsType(ConcreteDomain).typeDomain.oclIsTypeOf(ProductDomain)
            then arity =
               domain.oclAsType(ConcreteDomain).typeDomain.oclAsType(ProductDomain).domains.items->size()
F4:             else arity = 1
     endif endif endif
```

Table 2 Function Arity

```
context Domain def: let compatible(d : Domain): Boolean =
-- self and d are equals or d is Undef
  self = d or d.oclIsTypeOf(UndefDomain) or
-- one is a ConcreteDomain and the other is its type-domain
  ( self.oclIsTypeOf(ConcreteDomain) and d.oclIsTypeOf(TypeDomain) and
     d = self.oclAsType(ConcreteDomain).typeDomain ) or
  ( d.oclIsTypeOf(ConcreteDomain) and self.oclIsTypeOf(TypeDomain) and
     self = d.oclAsType(ConcreteDomain).typeDomain ) or
-- two PowersetDomain: their base domain must be compatible
  ( self.oclIsTypeOf(PowersetDomain) and d.oclIsTypeOf(PowersetDomain) and
     self.oclAsType(PowersetDomain).baseDomain.compatible(d.oclAsType(PowersetDomain).baseDomain)) or
-- two ProductDomain: their size must be equal and their domains must be compatible
  ( self.oclIsTypeOf(ProductDomain) and d.oclIsTypeOf(ProductDomain) and
   (let selfDoms:DomainCollection = self.oclAsType(ProductDomain).domains in
    let dDoms:DomainCollection = d.oclAsType(ProductDomain).domains in
     let nDoms = selfDoms.items->size()in nDoms = dDoms.items->size() and
        Sequence1..nDoms->forAll(i:Integer | selfDoms.items->at(i).compatible(dDoms.items->at(i)))))
```

Table 3 Domain Compatibility Definition

relationship between a `FunctionTerm` and its `arguments` of `TupleTerm` class is modelled (graphically shown by a solid filled diamond) as *composite aggregations*[6].

`FunctionTerm` has subclass `LocationTerm` to represent location terms, i.e. function applications $f(t_1, \ldots, t_n)$ with f a *dynamic* (T10) function fixed by the signature of the ASM.

`VariableTerm` class models *logical variables* which should not be confused with variables of programming defined as dynamic 0-ary functions in the ASM signature. Logical variables are typically used as formal parameters in function definitions or initializations, quantified expressions, and in rule declarations (see Sect. 7).

In ASMs, we have to distinguish logical variables from *location variables*, which also appear as formal parameters in a rule declaration, but can be used only in the rule body on the left-hand side of an update rule, and *rule variables* used at places where transition rules are expected (we better explain the role of these variables in Sect. 6). The attribute `kind` of type `VariableKind`

---

[6] A composition or composite aggregation is a special form of binary association that specifies a whole-part relationship between the aggregate (whole) and a component part. It requires that a part instance is included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts.

```
    context Term::compatible(t: Term) : Boolean
T1:  body: domain.compatible(t.domain)

    context BooleanTerm inv:
T2:  domain.oclIsTypeOf(BooleanDomain)
T3:  BooleanTerm.allInstances()->size()=2
T4:  BooleanTerm.allInstances()->exists(symbol
   = 'true') and BooleanTerm.allInstances()->
  exists(symbol = 'false')

    context UndefTerm inv:
T5:  domain.oclIsTypeOf(UndefDomain)
T6:  UndefTerm.allInstances()->size()=1
T7:  UndefTerm.allInstances()->
      exist(symbol = 'undef')

    context FunctionTerm inv:
T8:  domain = function.codomain
T9:  if function.arity = 0
     then arguments->isEmpty()
     else arguments->notEmpty() and
      arguments.domain.compatible(function.domain)
     endif

    context LocationTerm inv:
T10:  function.oclIsTypeOf(DynamicFunction)

    context VariableTerm inv:
T11:  domain.oclIsTypeOf(RuleDomain) =
       (kind = VariableKind::ruleVar)
```
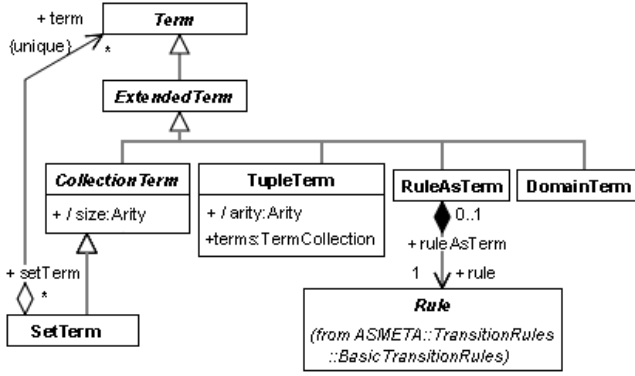
Table 4 Term Constraints

Figure 7 Extended Terms

allows to set the nature of a `VariableTerm` instance. `VariableKind` has been defined as enumeration class of values `locationVar` for location variables, `ruleVar` for rule variables, and `logicalVar` for logical variables. Location variables can be replaced only by location terms, rule variables only by rule terms, and logical variables by all kind of terms, except location terms and rule terms. These constraints are guaranteed by suitable OCL rules on classes `MacroCallRule` and `TurboCallRule` (see Sect. 6).

The variable kind is determined by the context in which the variable is assigned to a term, except for rule variables. In fact, if (and only if) the domain of a `VariableTerm` is the `RuleDomain`, then the variable is a rule variable (T11).

The class *ExtendedTerm* is the abstract root-class of the class hierarchy (see Fig. 7) introduced to represent special terms like tuples, collections, set terms, etc.

The class `TupleTerm` represents terms which are tuples of terms (listed in `terms`). The `arity` of a tuple is a derived attribute equal to the number (greater than 0 by definition of `TermCollection`) of composing terms (E1 in Tab. 5). If the arity of a tuple is 1, then the domain of the tuple is that of the component term (E2); otherwise, if the arity is greater than 1, the domain of the tuple is the Cartesian product of the domains of the corresponding component terms (E3).

Another subclass of *ExtendedTerm* is `RuleAsTerm`. It models special terms, called *rule terms*, used to represent a transition rule (linked by `rule`) where a term is expected (e.g as actual parameter in a rule application to represent a transition rule, see Sect. 6). The domain of a rule term must be `RuleDomain` (E4).

The class `DomainTerm` represents a term that appears where the identifier (or name) of a domain D fixed by the ASM signature, or the expression of a structured type-domain is expected. Usually, this term is combined with other terms to construct more complex terms to be used, for example, in a domain definition or initialization (see Sect. 7). The domain of a `DomainTerm` dt is a powerset (E5) whose `baseDomain` is the domain dt represents.

The abstract class *CollectionTerm* models types which are collection of terms, as set, bag, sequence, and map. The subclass `SetTerm` represents a mathematical set $\{t_1, \ldots, t_n\}$ where $t_1, \ldots, t_n$ are terms of the same type. A set does not contain duplicate elements, and has no order defined on it. The derived attribute `size` of a set term is the number of terms t_i it contains (E6). The domain of a set term must be a powerset domain over the domain D associated to all terms t_i (E7). Note that, if the set is empty, then D can be any type domain. D is generally set during the creation of the empty set depending on the type of the elements the set is going to contain.

In order to enrich the language of terms, Section 11.0.2 introduces other terms including numerical terms, collection terms (maps, sequences and bags), condition and case terms, and variable binding terms.

6 Metamodelling ASM Transition Rules

As extension of Finite State Machines, the ASMs are transition systems. The transition relation is specified by *rules* which update abstract states, namely they describe the modification of (dynamic) functions from one state to the next.

The *computation* of an ASM is modelled through a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an *initial state* and each S_{n+1} is obtained from S_n by firing simultaneously all rules which are enabled in S_n.

The abstract class *Rule* (see Fig. 8) models transition rules. We describe here one of its subclasses, the class *BasicRule* which represents the rule constructors for the basic model of ASMs, whereas more complex forms of ASM transition rules are later introduced to model Turbo ASM (see Sect. 8).

Let $l = (f, (v_1, \ldots, v_n))$ be a location and v be a value. Pairs (l, v) are called *updates* and they represent the basic form of state change. To fire an *update (rule)*

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

where f is a function symbol and $t_1, \ldots, t_n, t$ are terms, first all parameters t_i, t are evaluated in the current state of the ASM yielding to values v_i and v, respectively, then the value of f at $v_1, \ldots, v_n$ is updated to v which represents the value of $f(t_1, ..., t_n)$ in the next state.

The class `UpdateRule` (see Fig. 8) represents an update rule as in $l := t$, where the right-hand side t is a generic term and it is linked by the association end `updatingTerm`, while the left-hand side l is linked by the association end `location`. Term l must be compatible with t (R1 in Tab. 6).

The term l is either a *location term* $f(t_1, \ldots, t_n)$ or a *location variable term* x (R2). Note that in the latter case, the update $x := t$ occurs inside a rule $R(x)$ having x as parameter. When $R(x)$ is called as $R(l)$, with l a

```
      context TupleTerm inv:
       let size:Integer = terms.items->size() in
E1:   arity = size and
E2:   if size = 1 then domain = terms.items->at(1).domain
E3:   else domain.oclIsTypeOf(ProductDomain) and domain.oclAsType(ProductDomain).domains.items->size() =
          size and Sequence{1..size}->forAll(i:Integer |
            domain.oclAsType(ProductDomain).domains.items->at(i) = terms.items->at(i).domain) endif
E4:   context RuleAsTerm inv: domain.oclIsTypeOf(RuleDomain)
E5:   context DomainTerm inv: domain.oclIsTypeOf(PowersetDomain)

      context SetTerm inv:
E6:   size = term->size()
E7:   domain.oclIsTypeOf(PowersetDomain) and
          term->forAll(t:Term| domain.oclAsType(PowersetDomain).baseDomain = t.domain)
```

Table 5 Extended Term Constraints

location term, then x is replaced by l which is updated as explained before.

An update rule is the basic form of a transition rule. There are some rule constructors which are represented by subclasses of the class *BasicRule* as shown in Fig. 8 and 9.

Typically, an ASM transition system appears as a set of guarded updates

if *cond* **then** *updates*

All function *updates* are simultaneously executed when the condition *cond* is true.

A more general schema is the *conditional rule* (modelled by the subclass `ConditionalRule`) of the form

if φ **then** R_1 **else** R_2 **endif**

where φ, the `guard`, is a term representing a boolean condition (R3), R_1 (`thenRule`) and R_2 (`elseRule`) are transition rules. The meaning is: if the value resulting from the evaluation of the guard φ is *true* then execute R_1, otherwise execute R_2. If R_2 is omitted (since it is optional), from a semantic view it is assumed that $R_2 \equiv skip$, where *skip* is the empty rule whose meaning is: do nothing. The *skip rule* is represented by the subclass `SkipRule`.

If a set of transition rules have to be executed simultaneously, a *block rule* is used (modelled by `BlockRule`). It has the form

par $R_1 \ldots R_n$ **endpar**

and the meaning is: execute in parallel the transition rules $R_1 \ldots R_n$, listed in the `rules` attribute. In case of *inconsistency* (e.g. a set of updates with clashing elements) the computation does not yield the next state and an error message should be reported by the executing engine.

To construct new elements and to add them to domains, we use an *extend rule* of the form

extend D **with** $x_1, \ldots, x_n$ **do** R

where D is the (usually user-defined) domain to extend, $x_1, \ldots, x_n$ are logical variables which are bound to the new elements imported in D from the *reserve*, and R is a transition rule. The meaning is: choose the elements $x_1, \ldots, x_n$ from the reserve, delete them from the reserve, add them to the domain D and execute R. The class `ExtendRule` models an extend rule. The domain to extend (`extendedDomain`) must be dynamic (R4) and the variables x_i (linked by the association end `boundVar`) must be logical variables ranging in D (R5).

Non determinism is a convenient way to abstract from details of scheduling of rule executions. It can be expressed by a *choose rule* of the form

choose x_1 **in** D_1, ..., x_n **in** D_n **with** $\varphi(x_1, \ldots, x_n)$
do R_1 **ifnone** R_2

where $x_1, \ldots, x_n$ are variables, $D_1, ..., D_n$ are terms representing the domains where variables x_i take their value, φ is a term representing a boolean condition over the variables x_i, R_1 is a transition rule containing occurrences of the variables x_i, and R_2 is a transition rule. The meaning is: choose arbitrary x_1 in D_1, ..., x_n in D_n satisfying the condition φ and then execute R_1. If no such a set of variables exists execute R_2. If R_2 is omitted, it is assumed $R_2 \equiv skip$.

The subclass `ChooseRule` models a choose rule. Variables x_i are all different and are linked via the association end `variable`. D_i are listed in the attribute `ranges` (in which the same domain may occur several times). We assume that each D_i is a term: it may be either a domain (as `DomainTerm`) or a generic set (including but not limited to a `SetTerm` or a `ComprehensionTerm` introduced in Section 11.0.6). The domain of D_i must be a `PowersetDomain` (R6) whose base domain must be equal to the domain of x_i (R7). The term φ denoting the selection criteria must be boolean (R8). The number of variables x_i is equal to the number of terms D_i (R9).

Simultaneous execution allows to abstract from sequentiality where it is irrelevant for an intended design. This synchronous parallelism is expressed by a *forall rule* which has form

```
        context UpdateRule inv:
 R1:    location.compatible(updatingTerm)
 R2:    location.oclIsTypeOf(LocationTerm) or ( location.oclIsTypeOf(VariableTerm) and
                   location.oclAsType(VariableTerm).kind = VariableKind::locationVar )

 R3: context ConditionalRule inv: guard.domain.oclIsTypeOf(BooleanDomain)

        context ExtendRule inv:
 R4:    extendedDomain.isDynamic=true
 R5:    boundVar->forAll(v:VariableTerm | v.domain = extendedDomain and v.kind=VariableKind::logicalVar)

        context ChooseRule inv:
        Sequence{1..variable->size()}-> forAll(i:Integer |
 R6:       ranges.items->at(i).domain.oclIsTypeOf(PowersetDomain) and
 R7:          variable->at(i).domain = ranges.items->at(i).domain.oclAsType(PowersetDomain).baseDomain)
 R8:    guard.domain.oclIsTypeOf(BooleanDomain)
 R9:    variable->size() = ranges.items->size()

        context ForallRule inv:
        Sequence{1..variable->size()}-> forAll(i:Integer |
R10:       ranges.items->at(i).domain.oclIsTypeOf(PowersetDomain) and
R11:          variable->at(i).domain = ranges.items->at(i).domain.oclAsType(PowersetDomain).baseDomain )
R12:    guard.domain.oclIsTypeOf(BooleanDomain)
R13:    variable->size() = ranges.items->size()

        context LetRule inv:
R14:    initExpression->size() = variable->size() and Sequence{1..variable->size()}->
        forAll(i:Integer| variable->at(i).domain = initExpression->at(i).domain)

        context RuleDeclaration inv:
R15:    arity = variable->size()

        context MacroCallRule inv:
R16:    parameter->size() = calledMacro.arity
        let arity:Integer = parameters->size() in
        Sequence{1..arity}->forAll(i:Integer |
R17:  calledMacro.variable->at(i).compatible(parameters.items->at(i)) and
R18:  calledMacro.variable->at(i).kind=VariableKind::locationVar implies
        ( parameters.items->at(i).oclIsTypeOf(LocationTerm)
          or (parameters.items->at(i).oclIsTypeOf(VariableTerm) and
            parameters.items->at(i).oclAsType(VariableTerm).kind = VariableKind::locationVar))
```
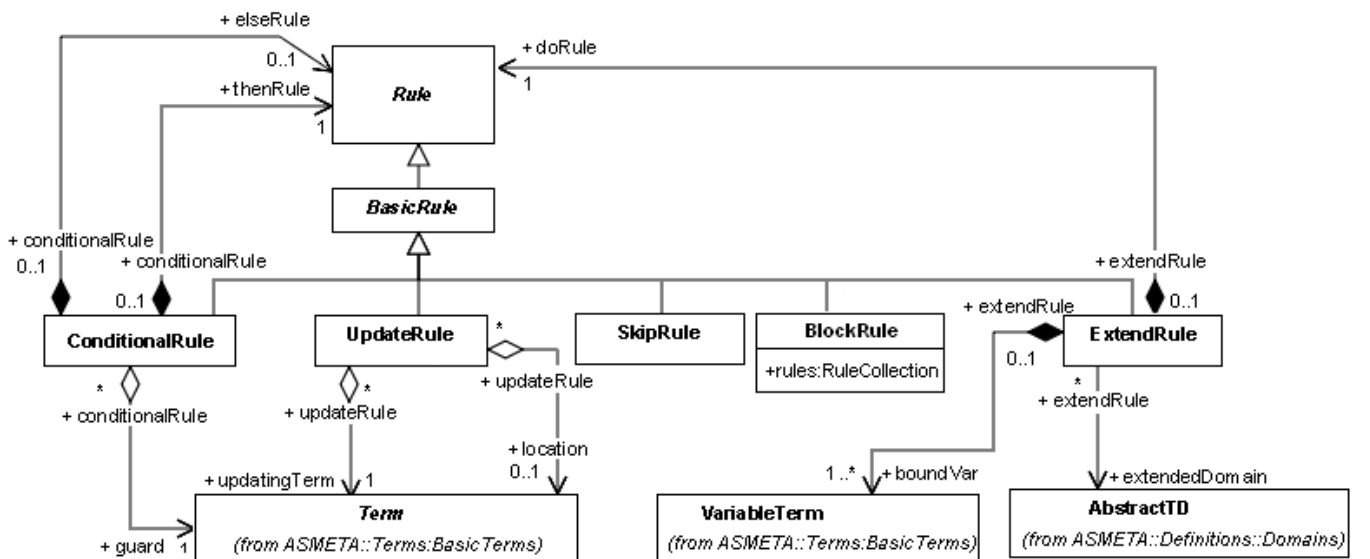
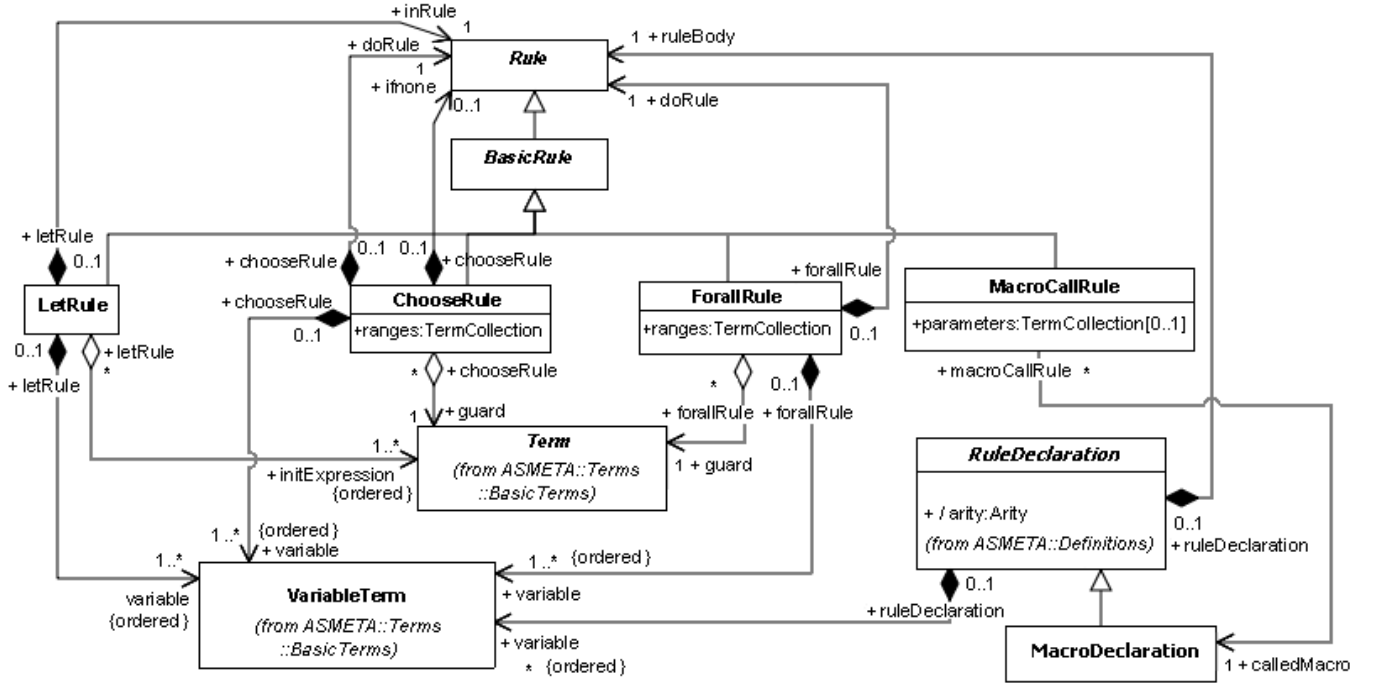**Table 6** Rule Constraints

Figure 8 Basic transition rules (Part1)

Figure 9 Basic transition rules (Part 2)

forall x_1 in D_1, ..., x_n in D_n with φ do R

where $x_1, \ldots, x_n$ are variables, $D_1, \ldots, D_n$ are terms representing the domains where variables x_i take their value, φ is a term representing a boolean condition over the variables x_i, R is a transition rule containing occurrences of the variables x_i bound by the quantifier. The meaning is: execute in parallel R for each set of variables $x_1, \ldots, x_n$ satisfying the given condition φ.

In a forall rule, modelled by the class `ForallRule`, the terms D_i are listed in the `ranges` attribute, while variables x_i are linked by the association end `variable`. The type-domain of every term D_i must be a power set domain (R10). The domain of x_i must be set accordingly (R11). The guard φ must be boolean (R12). The number of variables x_i is equal to the number of D_i (R13).

When abbreviation on terms or rules is necessary, in ASMs we can make use of *let rules* and *macros*, respectively. The class `LetRule` represents a *let rule* as in

let $(x = t)$ in R endlet

where x is a variable, t is a term, R is a transition rule which contains occurrences of the variable x. The meaning is: assign the value of the term t to the variable x and then execute R. This rule scheme can be generalized by the following reduction schema

let $(x_1 = t_1, \ldots, x_n = t_n)$ in R endlet $\equiv$
let $(x_1 = t_1)$ in let $(x_2 = t_2)$ in ...
        let $(x_n = t_n)$ in R endlet ... endlet endlet

In a let rule, terms t_i are linked by the association end `initExpression` and variables x_i are linked by the

association end `variable`. The number of terms t_i must be equal to the number of variables x_i, and the type-domain associated to each variable x_i must be equal to the one associated to each term t_i (R14).

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 part of the ASM state.

In order to structure large ASMs, one can introduce reusable rules, also called *macro*, by rule declarations modelled by the class `RuleDeclaration`. A *rule declaration* for a rule name r of arity n is an expression

$$r(x_1 \text{ in } D_1, \ldots, x_n \text{ in } D_n) = R$$

where R is a transition rule, the free variable of R are contained in the list $x_1 \ldots x_n$, and D_i are the domains where variables x_i take their value. In a `RuleDeclaration`, R is given by the association end `ruleBody`, $x_1 \ldots x_n$ are linked by the association end `variable`, r is given by the `name` attribute, and n by the `arity` derived attribute (set by the constraint R15). Formal parameters x_i can be *logical variables* or *location variables* or *rule variables* and they are the only freely occurring variables in the rule body R.

A `RuleDeclaration` refers to a *macro* rule or to a *Turbo submachine* which will be introduced in Sect. 8. The subclass `MacroDeclaration` models a rule declaration of a macro.

The class `MacroCallRule` (see Fig. 9) represents the application of a named macro rule r as in $r[t_1, \ldots, t_n]$. Its meaning is: expand $r[t_1, \ldots, t_n]$ with the body of

$$[\![\mathbf{skip}]\!]_\zeta^\mathfrak{A} = \varnothing$$

$$[\![f(t_1,\ldots,t_n) := t]\!]_\zeta^\mathfrak{A} = \{(l,v)\}$$
where $l = f\langle [\![t_1]\!]_\zeta^\mathfrak{A},\ldots,[\![t_n]\!]_\zeta^\mathfrak{A}\rangle$, $v = [\![t]\!]_\zeta^\mathfrak{A}$

$$[\![\mathbf{par}\ R_1\ R_2\ \mathbf{endpar}]\!]_\zeta^\mathfrak{A} = [\![R_1]\!]_\zeta^\mathfrak{A} \cup [\![R_2]\!]_\zeta^\mathfrak{A}$$

$$[\![\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2\ \mathbf{endif}]\!]_\zeta^\mathfrak{A} = [\![R_1]\!]_\zeta^\mathfrak{A}\ \text{if}\ [\![\varphi]\!]_\zeta^\mathfrak{A} = true$$

$$[\![\mathbf{if}\ \varphi\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2\ \mathbf{endif}]\!]_\zeta^\mathfrak{A} = [\![R_2]\!]_\zeta^\mathfrak{A}\ \text{if}\ [\![\varphi]\!]_\zeta^\mathfrak{A} = false$$

$$[\![\mathbf{let}\ (x = t)\ \mathbf{in}\ R\ \mathbf{endlet}]\!]_\zeta^\mathfrak{A} = [\![R]\!]_{\zeta[x\to a]}^\mathfrak{A}\ \text{where}\ a = [\![t]\!]_\zeta^\mathfrak{A}$$

$$[\![\mathbf{forall}\ x\ \mathbf{in}\ D\ \mathbf{with}\ \varphi\ \mathbf{do}\ R]\!]_\zeta^\mathfrak{A} = \bigcup_{v\in V} [\![R]\!]_{\zeta[x\to v]}^\mathfrak{A}$$
where $V = \{v \in [\![D]\!]_\zeta^\mathfrak{A}\ |\ [\![\varphi]\!]_{\zeta[x\to v]}^\mathfrak{A} = true\}$

$$[\![\mathbf{choose}\ x\ \mathbf{in}\ D\ \mathbf{with}\ \varphi\ \mathbf{do}\ R_1\ \mathbf{ifnone}\ R_2]\!]_\zeta^\mathfrak{A} = [\![R_1]\!]_{\zeta[x\to a]}^\mathfrak{A}$$
if $a \in \{v \in [\![D]\!]_\zeta^\mathfrak{A}\ |\ [\![\varphi]\!]_{\zeta[x\to v]}^\mathfrak{A} = true\}$

$$[\![\mathbf{choose}\ x\ \mathbf{in}\ D\ \mathbf{with}\ \varphi\ \mathbf{do}\ R_1\ \mathbf{ifnone}\ R_2]\!]_\zeta^\mathfrak{A} = [\![R_2]\!]_\zeta^\mathfrak{A}$$
if $\{v \in [\![D]\!]_\zeta^\mathfrak{A}\ |\ [\![\varphi]\!]_{\zeta[x\to v]}^\mathfrak{A} = true\} = \varnothing$

$$[\![r[t_1,\ldots,t_n]]\!]_\zeta^\mathfrak{A} = [\![body\ \tfrac{t_1,\ldots,t_n}{x_1,\ldots,x_n}]\!]_\zeta^\mathfrak{A}\ \text{if a macro definition}$$
$r(t_1,\ldots,t_n) = body$ exists

$$[\![r[\ ]]\!]_\zeta^\mathfrak{A} = [\![body]\!]_\zeta^\mathfrak{A}\ \text{if a macro definition}\ r = body\ \text{exists}$$

$$[\![\mathbf{extend}\ D\ \mathbf{with}\ x\ \mathbf{do}\ R]\!]_\zeta^\mathfrak{A} = [\![R]\!]_{\zeta[x\to a]}^\mathfrak{A} \cup \{(D\langle a\rangle, true)\}$$
$\cup\ \{(Reserve\langle a\rangle, false)\}\ \text{if}\ a \in Res(\mathfrak{A}) \setminus ran(\zeta)$,
where $Res(\mathfrak{A}) = \{a \in |\mathfrak{A}|\ |\ Reserve^\mathfrak{A}\langle a\rangle = true\}$

Table 7 Update Sets of Basic ASMs. $[\![R]\!]_\zeta^\mathfrak{A}$ is the set of updates defined by rule R in state $\mathfrak{A}$ with variable assignment ζ of range $ran(\zeta)$. Updates are pairs (l,v) of locations l and values v, to which the location is intended to be updated. Locations $l = f\langle [\![t_1]\!]_\zeta^\mathfrak{A},\ldots,[\![t_n]\!]_\zeta^\mathfrak{A}\rangle$ consist of an n-ary function name f with a sequence of length n of elements in the domain of $\mathfrak{A}$. The value $f^\mathfrak{A}\langle [\![t_1]\!]_\zeta^\mathfrak{A},\ldots,[\![t_n]\!]_\zeta^\mathfrak{A}\rangle$ is the content of the location in $\mathfrak{A}$.

r replacing the occurrences of the formal parameters $x_1,\ldots,x_n$ of r with the values of the corresponding actual parameters $t_1,\ldots,t_n$, and then execute it. The rule to apply is linked by the association end `calledMacro`. The number of actual **parameters** $t_1,\ldots,t_n$ must be equal to the arity of the rule to apply (R16) and formal parameters x_i must be compatible with the corresponding terms t_i(R17).

If a formal parameter of a macro rule declaration is a location variable, then it can be replaced only by an actual parameter which is either a location term or a location variable (R18).

The semantics of all transition Basic ASM rules is summarized by the calculus in Tab. 7 inspired by [16].

7 Metamodelling Basic ASMs

According to the working definition given in [16], a basic ASM has a *name* and is defined by a *header* (to en-

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

    context ExportClause inv:
A2:  exportedDomain->notEmpty() or
     exportedFunction->notEmpty() or
     exportedRule->notEmpty()
```

Table 8 Basic Asm Constraints (Part 1)

stablish the signature) and a *body* (to define domains, functions, and rules). Executing a basic ASM means executing its *main rule* starting from one specified *initial state*.

Fig. 10 shows the *MOF model of a Basic ASM* defined by a name, a `Header`, a `Body`, a `mainrule` and an `Initialization`.

The class `Asm` is a root element (just like a root node in a graph) from which all the other elements of the metamodel can be reached. An instance of this class represents an entire ASM specification. Note that, the composite relationships between the class `Asm` (the *whole*) and its component classes (the *parts*) assures that each part is included in at most one `Asm` instance.

Since ASM allows structured model construction, a notion of library module is also supported to syntactically structure large specifications. An ASM *module* is an ASM without a main rule and without a set of initial states [7].

The `mainrule` is a named transition rule which usually takes the same name of the ASM. It is *closed*, i.e. it does not contain free variables (A1 in Tab. 8), so that its semantics depends on the state of the machine only.

The *header*, modelled by the class `Header` (see Fig. 11), consists of some `import clause`s and an optional `export clause` to specify the names which are imported from or exported to other ASMs (or ASM modules), and of its `signature` containing the *declarations* of the ASM domains and functions. Every ASM is allowed to use only identifiers (for domains, functions and transition rules) which are defined within its signature or imported from other ASMs or ASM modules.

The class `ImportClause` provides names of domains, functions and transition rules that are imported from other ASMs. The `imported domains` and `functions` will be statically added to the signature of the ASM[8], while the `imported` transition `rules` will enrich its body interface. If an ASM (or ASM module) is imported without specifying the imported domains, functions and rules names, we assume that all the content of the export clause of the imported ASM is imported. Moreover, do-

---

[7] This definition of module differs slightly from the module concept outlined in Chap. 2 of [16]; but, it has been accorded with the authors.

[8] We assume that there are no name clashes in the signature. However, we admit function overloading provided that their domains are different.
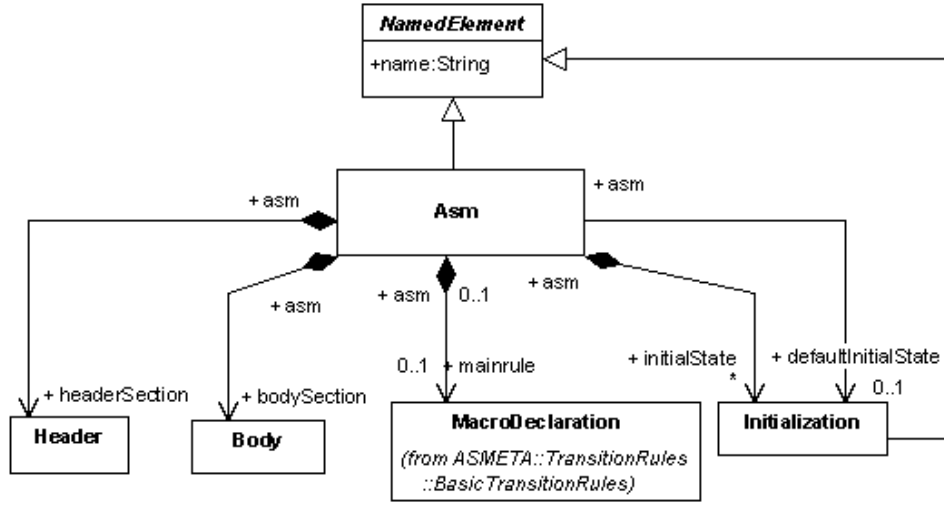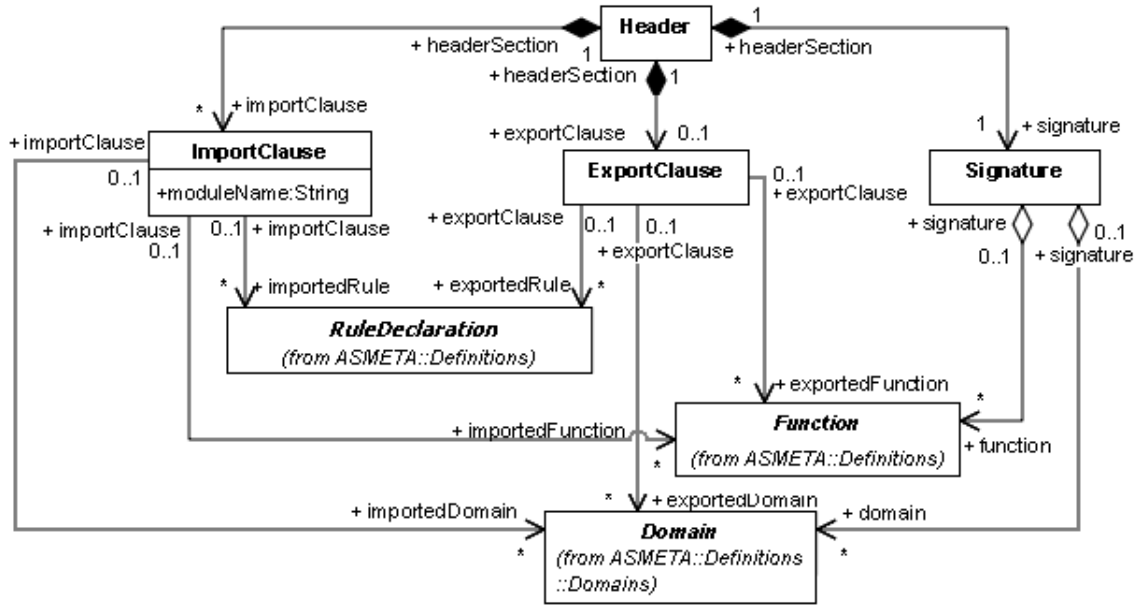
Figure 10 Backbone

Figure 11 Header

mains and codomains of imported functions are automatically added to the signature, although their names are not explicitly specified in the import clause.

The class `ExportClause` provides names of domains, functions, and transition rules that can be imported by other ASMs or ASM modules. If it exists, then it must contain at least one domain, or one function, or one rule to export (A2).

The class `Signature` is a finite collection of `function` and `domain` names (declarations).

The initialization of an ASM consists of a set of *initial states*. The class `Initialization` (see Fig. 12) models the notion of an initial state. All possible initial states are linked (see Fig. 10) to an ASM by the association end `initialstate` and one initial state is elected as *default* (designed by the association end `defaultIni-`

`tialState`). Each initial state has a `name` and is defined specifying an initial value for dynamic functions and dynamic concrete domains declared in the signature of the ASM. The class `DomainInizialization` models the initialization of a dynamic concrete domain and the class `FunctionInitialization` models the initialization of a dynamic function. An `Initialization` consists of a set of `DomainInizialization` linked by the association end `domainInizialization` and a set of `FunctionInitialization` linked by the association end `functionInitialization`. Within an initial state of an ASM there must be at least one domain or function initialization (I1 in Tab. 9), and a dynamic concrete domain and a dynamic function can be initialized only once (I2 and I3).

```
     context Initialization inv:
I1:    domainInitialization->notEmpty() or functionInitialization->notEmpty()
I2:    domainInitialization->forAll(d1,d2:DomainInitialization | d1<>d2 implies
           d1.initializedDomain<>d2.initializedDomain)
I3:    functionInitialization->forAll(f1,f2:FunctionInitialization|f1<>f2 implies
            f1.initializedFunction<>f2.initializedFunction)
     context DomainInizialization inv:
I4:    initializedDomain.isDynamic = true
I5:    body.domain.oclIsTypeOf(PowersetDomain) and
           body.domain.oclAsType(PowersetDomain).baseDomain = initializedDomain.typeDomain
     context FunctionInitialization inv:
I6:    variable->size() = initializedFunction.arity
       let size:Integer = self.variable->size() in let D:Domain = self.initializedFunction.domain in
I7:    if size = 1 then variable->at(1).domain = D
       else if size > 1 then
I8:      (D.oclIsTypeOf(ProductDomain) and Sequence{1..size}-> forAll(i:Integer |
           variable->at(i).domain = D.oclAsType(ProductDomain).domains.items->at(i) )) or
I9:      (D.oclIsTypeOf(ConcreteDomain) and
           D.oclAsType(ConcreteDomain).typeDomain.oclIsTypeOf(ProductDomain) and Sequence{1..size}->
           forAll(i:Integer | variable->at(i).domain = D.oclAsType(ConcreteDomain).typeDomain.
                   oclAsType(ProductDomain).domains.items->at(i)) ) endif endif
I10:   initializedFunction.codomain.compatible(body.domain)
```

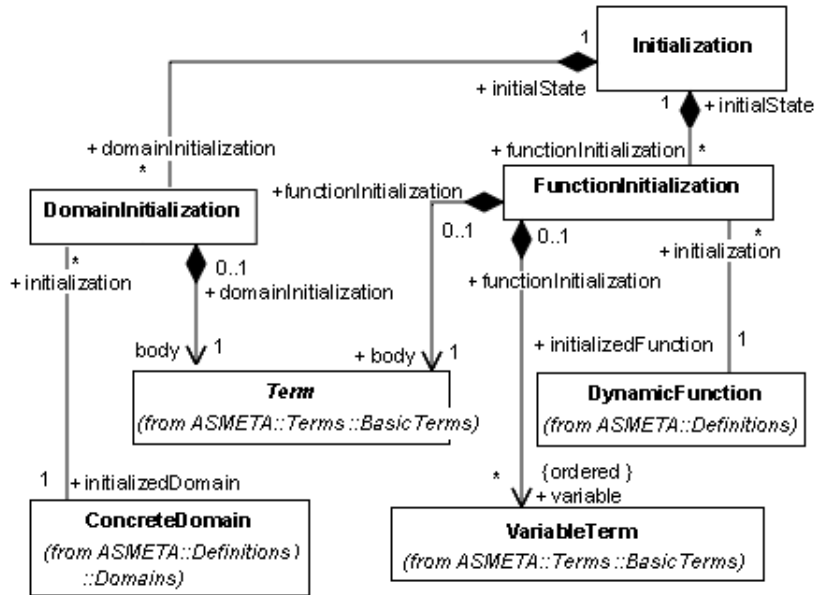**Table 9** Initialization Constraints

Figure 12 Initialization

The concrete domain initialized by a `DomainInizialization` is linked by the association end `initializedDomain` and must be dynamic(I4). The term `body` specifies the elements initially belonging to the domain. The domain of `body` must be a powerset domain over D, where D is the type-domain of the concrete domain to initialize (I5).

The function initialized by a `FunctionInitialization` is linked by the association end `initializedFunction` and must be dynamic. A function initialization, has form

$$f(x_1 \text{ in } D_1, \ldots, x_n \text{ in } D_n) = body$$

where $x_1, \ldots, x_n$ are the formal parameters of the function linked by the association end `variable`, $D_1, \ldots, D_n$ are the domains where $x_1, \ldots, x_n$ take their value[9], and `body` is the term defining the function. A dynamic function can be initialized either by declaring the associations between elements of $\text{domain}(f)$ with elements of $\text{codomain}(f)$ by means of a map term (see Sect. 11.0.2), or by specifying an initialization law which has not to

---

[9] The association end `domain` in Fig. 32 on page 38 links the variable x_i to its domain D_i

be intended as a function law, like for a static function, but a concise form for function initialization.

In a function initialization, the number n of formal parameters x_i must be equal to the arity of the function to initialize (I6). If the function to initialize has:

- only one formal parameter x, then the domain D where x takes its value must be equal to the domain D_f of the function (I7).
- more than one formal parameter $x_1, .., x_n$ taking value in $D_1, .., D_n$, then the domain D of the function is either directly a product domain $D_1 \times \ldots \times D_n$ (case I8), or D is a concrete domain whose type-domain is a product domain $A_1 \times \ldots \times A_n$, and each domain A_i is equal to D_i for $i = 1..n$ (case I9).

For the function body, which is a generic term, we require that its domain is compatible with the function codomain (I10).

The `Body` (see Fig. 13) of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations of transition rules and definitions of axioms for constraints one wants to assume for some domains, functions, and transition rules of the ASM.

The class `DomainDefinition` defines a concrete static domain (B1 in Tab. 10 on the next page) . The term `body` specifies the elements of the concrete domain. The `body` domain must be the power set domain over the domain D, being D the type-domain associated to the concrete domain (B2).

The class `FunctionDefinition` represents the definition of a static/derived function (B3) declared in the ASM signature. A function definition, which specifies the virtual table that associates domain elements to codomain elements, is similar to function initialization. Therefore, it has form

$$f(x_1 \text{ in } D_1, \ldots, x_n \text{ in } D_n) = body$$

where `variables` $x_1, \ldots, x_n$ are the formal parameters of the function, $D_1, \ldots, D_n$ are the domains where $x_1, \ldots, x_n$ take their value, and `body` is the term defining the function. If the function is static, it can be defined either explicitly declaring the associations between elements of `domain`(f) with elements of `codomain`(f) by means of a map term, or specifying the function law. A derived function is always given specifying its law since it is defined in terms of other functions.

Besides the constraint B3, the `FunctionDefinition` class has OCL constraints similar to those defined for the `FunctionInitialization` class above.

The class `Axiom` is detailed in Fig. 14. It represents a constraint one wants to assume for functions (typically dynamic monitored functions), domains fixed by the signature, and rules. The expression of an axiom is given by a term, specified by the association end `body`, which yields a boolean value (B4) when valuated in a state of the ASM. An axiom must refer to at least one rule, or one function or one domain (B5).

Note that axioms, domains, functions, and rule declarations are all represented by sub-classes of the *abstract* class `Classifier` (see Fig. 14 on the facing page).

8 Metamodelling Turbo ASMs

The characteristics of basic ASMs (simultaneous execution of multiple atomic actions in a global state) come at a price, namely the lack of direct support for practical composition and structuring principles. *Turbo ASMs* offer as building blocks sequential composition, iteration, and parameterized (possibly recursive) sub-machines extending the macro notation used with basic ASMs. They capture the sub-machine notion in a black-box view hiding the internals of subcomputations by compressing them into one step.

A Turbo ASM can be obtained from basic ASMs by applying finitely often and in any order the operators of *sequential composition, iteration* and *submachine call*. Therefore, we need to extend the abstract syntax of Basic ASMs (Sect. 7) by introducing in the metamodel Turbo ASM operators, shown in Fig. 15 and 16.

The class `SeqRule` (see Fig. 15) denotes the sequential composition **seq** R_1 R_2 **endseq** of two transition rules R_1 and R_2. The semantics is: first execute R_1 in the current state $\mathfrak{A}$ and then R_2 in the resulting state $\mathfrak{A} + U$ (if defined), where U is the set of updates produced by R_1 in $\mathfrak{A}$. Note that R_2 may overwrite a location which has been updated by R_1 (merging of two update sets). Moreover, this definition implies that a sequential computation gets stuck to U once an inconsistency is encountered in its first part R_1. This semantics can be straightforwardly generalized to a sequential composition of n rules as in **seq** R_1 $\ldots$ R_n **endseq**. Rules $R_1 \ldots R_n$ are listed in the attribute `rules` which may contain the same rule several times.

The class `IterateRule` (see Fig. 15) denotes an iterated rule application of form **iterate** R **enditerate**. Its semantics can be given in terms of the semantics of a sequence rule according to the following scheme: $R^0 = skip$ and $R^{n+1} = $ **seq** R^n R **endseq**. Denote by $\mathfrak{A}_n$ the state (if defined) obtained by firing the update set produced by R^n in state $\mathfrak{A}$. There are two natural stop situations for iterations without a priori fixed bound, namely when the update set produced by R^n becomes empty (the case of *successful termination*) and when it becomes inconsistent (the case of *failure*). The rule R to be iterated is denoted by the association end `rule`.

The submachine concept is defined by means of *turbo submachines* which are named parameterized rule calls $r(t_1, \ldots, t_n)$ with actual parameters $t_1, \ldots, t_n$, coming with a rule definition of the form $r(x_1, \ldots, x_n) = R$, where R is a rule called *body*. The subclass `TurboDeclaration` of the class `RuleDeclaration` (see Fig. 9) models
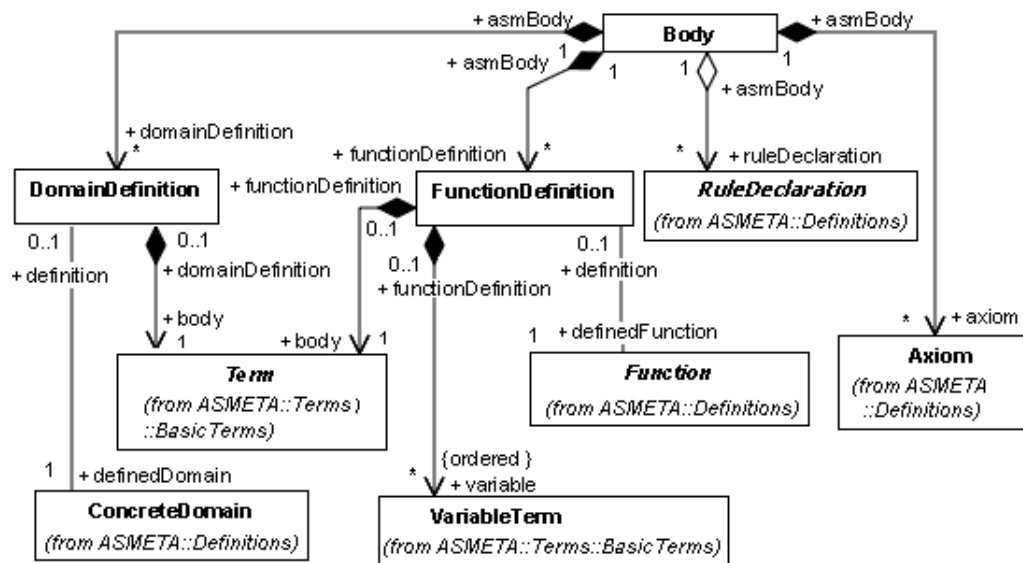
Figure 13 Body

```
      context DomainDefinition inv:
B1:     definedDomain.isDynamic = false
B2:     body.domain.oclIsTypeOf(PowersetDomain) and
           body.domain.oclAsType(PowersetDomain).baseDomain = definedDomain.typeDomain
      context FunctionDefinition inv:
B3:     definedFunction.oclIsTypeOf(StaticFunction) or definedFunction.oclIsTypeOf(DerivedFunction)
        -- constraints I6,I7, I8, I9, and I10
      context Axiom inv:
B4:     body.domain.oclIsTypeOf(BooleanDomain)
B5:     constrainedRule->notEmpty() or constrainedFunction->notEmpty() or constrainedDomain->notEmpty()
```

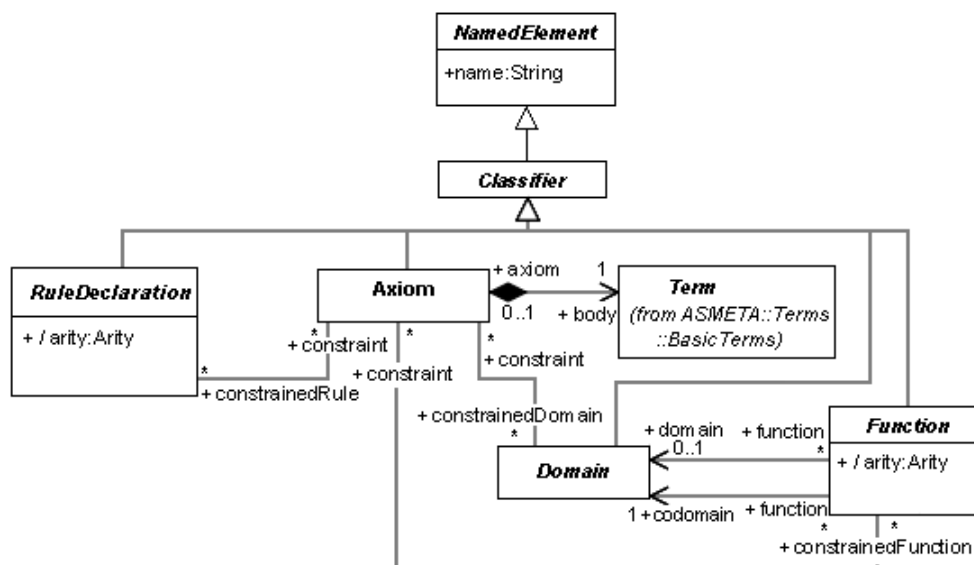**Table 10** Basic Asm Constraints (Part 2)

Figure 14 Common ASM definitions

the declaration of a *Turbo submachine* whose semantics is different from the one of a macro. A macro rule, as presented in Sect. 6, represents a reusable unit which is replaced by its definition whenever used. A Turbo submachine implies internal hidden subcomputations which are compressed in one step executable in parallel with the other rules. The declaration of a turbo submachine may specify also the domain of the returned result (if any) in the association end `resultType`.

We distinguish a *macro rule application* $r[t_1, \ldots, t_n]$ (see Sect. 6) from a *turbo rule application* $r(t_1, \ldots, t_n)$. The class `TurboCallRule` (see Fig. 15) represents the application $r(t_1, \ldots, t_n)$ of a named rule r (`calledRule`) with a *turbo submachine* semantics to express in abstract form the usual imperative calling mechanism, including recursive invocations of rules. Nested calls of a recursive rule r are unfolded into a sequence $R_1, R_2, \ldots$ of rule incarnations, where each R_i may trigger one more execution of r, by relegating the interpretation of possibly yet another call of r to the next incarnation R_{i+1}. This may produce an infinite sequence. If, however, a basis for the recursion does exist, say R_n, it yields a well-defined value for the semantics of r through the chain of successive calls of R_i; otherwise the semantics of the rule application is undefined. The rule body is executed substituting the occurrences of the formal parameters x_i with the actual arguments t_i in a *call-by-name* fashion, i.e. the formal parameters are substituted by the actual parameters' expressions which are evaluated only later when they are used (not in the current state when the rule is applied). A *call-by-value* evaluation, however, can be achieved (as suggested in [16], Sect. 4.1.2) by combining the rule r with a *let rule* as follows:

$r(y_1, \ldots, y_n) =$
$\quad\quad$ `let` $(x_1 = y_1, \ldots, x_n = y_n)$ `in` *body* `endlet`

In a `TurboCallRule` the number n of actual **parameters** $t_1, \ldots, t_n$ must be equal to the arity of the rule (U1 in Table 11), and each actual parameter t_i must be compatible with the corresponding formal parameter x_i (U2). Furthermore, location variables which appear as formal parameters of a rule declaration can be replaced only by actual parameters which are location terms or location variables (U3).

The class `TryCatchRule` (see Fig. 16) represents a try-catch rule of form **try** P **catch** $l_1, \ldots, l_n$ Q, where P and Q (denoted in the diagram by the association ends `tryRule` and `catchRule`, respectively) are rules, and l_i are locations terms or location variables (denoted by the association end `location`). A try-catch rule is used to separate error handling from normal execution. Producing an inconsistent update set has to be considered here as an abstract form of throwing an exception. The execution of a try-catch rule results in the update set of P, if this update set is consistent on the locations l_i, otherwise – if P fails but without inconsistency over l_i or if P succeeds without inconsistency – it removes the effects of executing P and yields the resulting update set
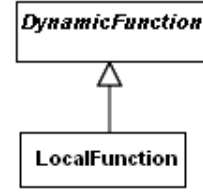

Figure 17 Local Function

of Q. Note that, since the rule enclosed by the try-block is executed either completely or not at all, there is no need for a sort of *finally clause* to remove trash.

For the class `TryCatchRule` we impose the constraint that at least one location or location variable must occur in the location set $\{l_1, \ldots, l_n\}$ and the location set $\{l_1, \ldots, l_n\}$ contains only location variables or location terms (U4).

The class `TurboReturnRule` (see Fig. 16) represents the special assignment $l \longleftarrow r(t_1, \ldots, t_n)$ where l is a location (linked by the association end `location`) and $r(t_1, \ldots, t_n)$ is a turbo submachine (linked by the association end `updateRule`) which returns a value to be assigned to l. The location l in which to store the intended return value can be either a location variable term or a location term (U5), and the domain of l must be compatible with the domain of the returned value (U6). A turbo-return rule has the overall effect of executing the body of r, where the location l has been substituted for a reserved 0-ary function **result**, which acts as placeholder for the location in which to store the return value. A good encapsulation discipline should take care that r does not modify the values of terms appearing in l, since they contribute to determining the location where the caller expects to find the return value.

The class `TurboLocalStateRule` (see Fig. 16) represents a turbo submachine with locally visible parts of the state. These submachines are obtained from named turbo rules by allowing some dynamic functions to be declared locally to the rule. In order to model *local* dynamic functions, the classification of dynamic functions shown in Fig. 5 must be modified by adding the class `LocalFunction` (see Fig. 17). Furthermore we must add a constraint to the class `Signature` to guarantee that it does not contain declarations of local functions (U7), since function names declared in it have a global visibility within the scope of the entire ASM.

A named turbo submachine $r(x_1, \ldots, x_n)$ with local state can be defined as:

$r(x_1, \ldots, x_n) = $ **local** $f_1 : D_1 \to C_1[Init_1]$
$\quad\quad\quad\quad \ldots$
$\quad\quad\quad\quad$ **local** $f_k : D_k \to C_k[Init_k]$
$\quad\quad\quad\quad$ *body*

where *body* and $Init_i$ are rules linked by the association ends `body` and `init`, f_i are local dynamic functions from domain D_i to domain C_i and linked by the association end `localFunction`. The number k of functions
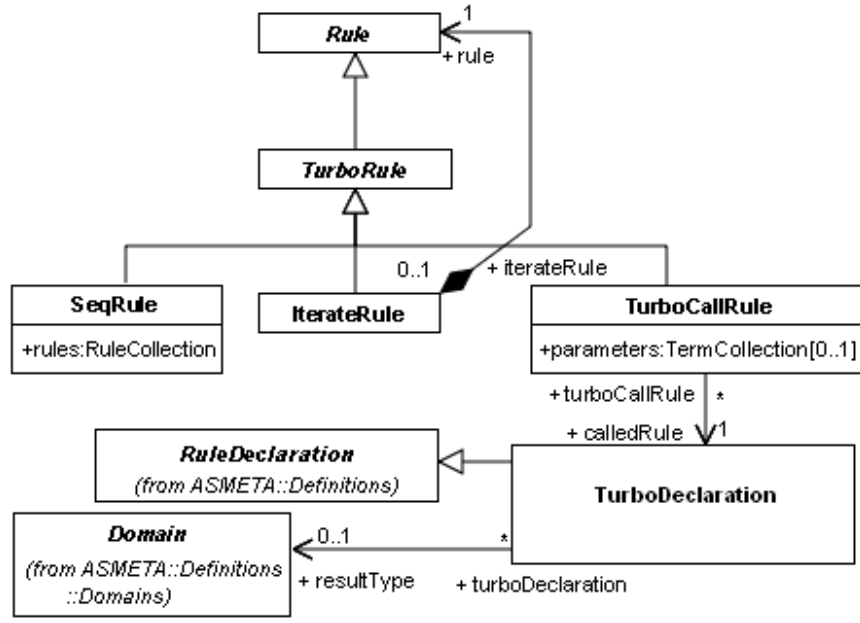
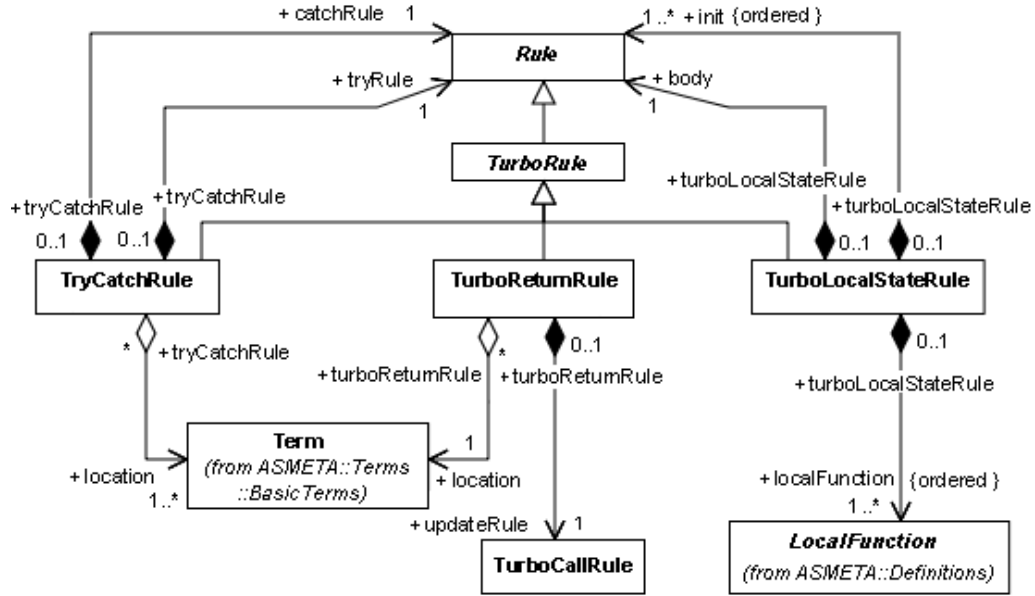**Figure 15** Turbo transition rules (Part 1)

Figure 16 Turbo transition rules (Part 2)

declared locally must be equal to the number of the initializing rules (U8). Each application of r works with its own incarnation of the local dynamic functions f_i. Each local function f_i is initialized by the corresponding rule $Init_i$ before the *body* rule is executed. The update set for the local functions are collected together with all other function updates made in *body*, including the updates of the initializing rules $Init_i$. The restriction of the scope of the local functions to the rule definition is obtained then by removing from the update set available after the execution of *body* the set of updates concerning the local functions f_i.

The calculus Table 12 inspired by [16] summarizes the semantics of all transition rules for Turbo ASM.

9 Metamodelling Multi-agent ASMs

A *multi-agent synchronous ASM* is defined as a set of agents executing their own ASMs in parallel, synchronized using an implicit global system clock. Semantically a sync ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component.

An *multi-agent asynchronous ASM* is given by a family of pairs $(a, ASM(a))$ of pairwise different agents, ele-

```
     context  TurboCallRule inv:
U1:   parameters->size() = calledRule.arity
         Sequence{1..parameters->size()}->forAll(i:Integer |
U2:       calledRule.variable->at(i).compatible(parameters.items->at(i))) and
U3:       (calledRule.variable->at(i).kind=VariableKind::locationVar implies
          (parameters.items->at(i).oclIsTypeOf(LocationTerm)
          or ( parameters.items->at(i).oclIsTypeOf(VariableTerm)
            and parameters.items->at(i).oclAsType(VariableTerm).kind = VariableKind::locationVar)) )
     context TryCatchRule inv:
U4:    location->forAll(t:Term | t.oclIsTypeOf(LocationTerm) or ( oclIsTypeOf(VariableTerm) and
          t.oclAsType(VariableTerm).kind=VariableKind::locationVar))
     context TurboReturnRule inv:
U5:    location.oclIsTypeOf(LocationTerm) or ( location.oclIsTypeOf(VariableTerm) and
    location.oclAsType(VariableTerm).kind=VariableKind::locationVar))
U6:      and (location.domain.compatible(updateRule.calledRule.resultType))
U7: context Signature inv: function->forAll(f:Function| not f.oclIsTypeOf(LocalFunction))
U8: context TurboLocalStateRule inv: localFunction->size() = init->size()
```

Table 11 Turbo ASM Constraints

$[\![\mathbf{seq}\ R_1\ R_2\ \mathbf{endseq}]\!]^{\mathfrak{A}}_\zeta = [\![R_1]\!]^{\mathfrak{A}}_\zeta \oplus [\![R_2]\!]^{\mathfrak{A}+[\![R_1]\!]^{\mathfrak{A}}_\zeta}_\zeta$ where the notation $\oplus$ indicates the merging of two update sets $U \oplus V = \{(l,v) \in U \mid l \notin locs(V)\} \cup V$, if U is consistent, $U \oplus V = U$, otherwise

$[\![\mathbf{iterate}\ R\ \mathbf{enditerate}]\!]^{\mathfrak{A}}_\zeta = lim_{n\to+\infty}[\![R^n]\!]^{\mathfrak{A}}_\zeta$, if for some $n \geq 0$ it holds that $[\![R]\!]^{\mathfrak{A}^n}_\zeta = \varnothing$ or $[\![R]\!]^{\mathfrak{A}^n}_\zeta$ is inconsistent

$[\![r(t_1,\ldots,t_n)]\!]^{\mathfrak{A}}_\zeta = [\![body\ [t_1/x_1,\ldots,t_n/x_n]\ ]\!]^{\mathfrak{A}}_\zeta$ if a turbo submachine definition $r(t_1,\ldots,t_n) = body$ exists

$[\![r(\ )]\!]^{\mathfrak{A}}_\zeta = [\![body]\!]^{\mathfrak{A}}_\zeta$ if a turbo submachine definition $r = body$ exists

$[\![\mathbf{try}\ P\ \mathbf{cacth}\ l_1,\ldots,l_n\ Q]\!]^{\mathfrak{A}}_\zeta = [\![P]\!]^{\mathfrak{A}}_\zeta$, if $[\![P]\!]^{\mathfrak{A}}_\zeta$ is consistent on the locations determined by the set $\{l_1,\ldots,l_n\}$

$[\![\mathbf{try}\ P\ \mathbf{cacth}\ l_1,\ldots,l_n\ Q]\!]^{\mathfrak{A}}_\zeta = [\![Q]\!]^{\mathfrak{A}}_\zeta$, if $[\![P]\!]^{\mathfrak{A}}_\zeta$ is not consistent on the locations determined by the set $\{l_1,\ldots,l_n\}$

$l \leftarrow [\![r(t_1,...,t_n)]\!]^{\mathfrak{A}}_\zeta = [\![body\ [l/\mathbf{result},t_1/x_1,...,t_n/x_n]\ ]\!]^{\mathfrak{A}}_\zeta$ if a turbo submachine definition $r(t_1,...,t_n) = body$ exists

$[\![r(t_1,\ldots,t_n)]\!]^{\mathfrak{A}}_\zeta = [\![(\{Init_1 \ldots Init_k\}\ \mathbf{seq}\ body)\ [t_1/x_1,...,t_n/x_n]\ ]\!]^{\mathfrak{A}}_\zeta \setminus U$ where U is the update set concerning the local functions f_i, if a turbo submachine definition
$r(x_1,\ldots,x_n) =$
    $\mathbf{local}f_1 : [D_1] \to C_1[Init_1]$
    $\ldots$
    $\mathbf{local}f_k : [D_k] \to C_k[Init_k]$
    $body$
exists

Table 12 Update Sets of Turbo ASMs. $[\![R]\!]^{\mathfrak{A}}_\zeta$ is the set of updates defined by rule R in state $\mathfrak{A}$ with variable assignment ζ. Updates are pairs (l,v) of locations l and values v, to which the location is intended to be updated. Locations $l = f\langle[\![t_1]\!]^{\mathfrak{A}}_\zeta,\ldots,[\![t_n]\!]^{\mathfrak{A}}_\zeta\rangle$ consist of an n-ary function name f with a sequence of length n of elements in the domain of $\mathfrak{A}$.

ments of a possibly dynamic finite set *Agent*, each executing its (possibly the same but differently instantiated) basic, structured or sync ASM *ASM(a)*. Similar to the use of *this* in OO programming to denote the object for which the currently executed instance method has been invoked, the relation between global and local states is supported by the use of a reserved word **self** in functions and rules to denote the agents.

A multi-agent ASM with synchronous agents has *quasi-sequential runs*, namely a sequence of states where each state is obtained from the previous state by firing in parallel the rules of all agents.

A multi-agent ASM with asynchronous agents has *partially ordered runs*, namely a partially ordered set $(M, <)$ of moves m (read: rule applications) of its agents satisfying the following conditions: (a) *finite history*: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m' \mid m' < m\}$ is finite; (b) *sequentiality of agents*: the set of moves $\{m \mid m \in M, a$ performs $m\}$ of every agent $a \in Agent$ is linearly ordered by $<$; (c) *coherence*: each finite initial segment (downward closed subset) X of $(M, <)$ has an associated state $\sigma(X)$ – think of it as the result of all moves in X with m executed before m' if $m < m'$ – which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$. In this section the abstract syntax presented in Sect. 7 is extended to support multi-agent ASMs.

As shown in Fig. 18, the Asm class is endowed with a further attribute isAsynchr of type Boolean (*false* by default) which indicates whether the machine is asynchronous multi-agent or not. For single-agent ASMs it has no meaning.

Agents are identified with the elements of a predefined abstract domain *Agent* represented (see Fig. 18) by the class AgentDomain, which is a singleton class (M1 in
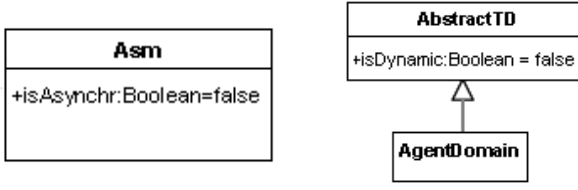
Figure 18 Refinements for Multi-agent ASMs

Tab. 13) whose unique instance is *Agent* and is dynamic (M2).

Two predefined functions, $id : Agent \rightarrow String$ and $program : Agent \rightarrow Rule$, are defined (as instances of the class `Function` in the AsmM standard library) to indicate the *id* name of the agent and the *program*, i.e a transition rule, respectively. The transition rule associated as program to an agent can be either a named transition rule or an unnamed transition rule of the ASM (M3).

Within transition rules, each agent can identify itself by means of a special reserved 0-ary function $self : Agent$ (defined in the AsmM standard library as instance of the class `Function`), which is interpreted by each agent a as a. Usually, for a function f defined from X to Y, both the expressions (function application terms) $f(self, x)$ and $self.f(x)$ denote the private version of $f(x)$ of the agent $self$.

An initial state of a multi-agent ASM should also specify an initial set of running agents. This is done via some initialization clauses represented in the metamodel by instances of the class `AgentInitialization` (see Fig. 19). In case the initial state of the ASM does not specify these clauses, it is assumed that the ASM is single-agent, and that the name and the main rule of the ASM are respectively the identifier and the program of the single running agent.

Executing a multi-agent ASM means executing its main rule starting from one specified initial state. If the ASM has no main rule, by default, the ASM is started executing in parallel the agents' programs as specified by the agent initialization clauses. If neither the main rule nor the agent initialization clauses are provided, the ASM is considered a library module.

The OCL constraint I1 within the context `Initialization` defined in Sect. 7 must be redefined as in M4 to guarantee that an ASM initial state contains at least one domain initialization, or function initialization, or agent initialization.

10 Metamodel Architecture

The complete metamodel is organized in one package called `ASMETA`, which is further divided into four packages as shown in Fig. 20. Each package covers different aspects of the ASMs. The dashed gray ovals in Fig. 20
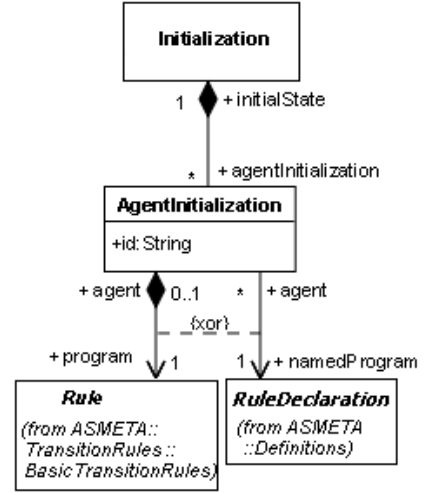

Figure 19 Agent Initialization

```
    context AgentDomain inv:
M1:    AgentDomain.allIstances->size() = 1
M2:    AgentDomain.isDynamic = true
    context AgentInitialization inv:
M3:    program->notEmpty() xor
    namedProgram->notEmpty()
    context Initialization inv:
M4:    domainInitialization->notEmpty() or
    functionInitialization->notEmpty() or
    agentInitialization->notEmpty()
```

Table 13 Multi Agent ASM Constraints

denote the packages representing the notions of *State* and *Transition System*, respectively.

- the `Structure` 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 root class `Asm`.
- the `Definitions` package (or *the definitional language*) contains all basic constructs (functions, domains, constraints, rule declarations, etc..) which characterize algebraic specifications. The content of this package is mainly reported in Fig. 14 (Fig. 3 and 4 report other specialized sub-classes representing specific domains). Under the root *abstract* class *Classifier*, here we find classes *Domains*, *Function*, *RuleDeclaration*, `Axiom`. The hierarchy of classes rooted by the class *Domain* is contained in the sub-package `Domains`.
- the `Terms` package (or *the language of terms*) provides all kinds of syntactic expressions which can be evaluated in a state of an ASM. It is divided in two packages: `BasicTerms` and `FurtherTerms`. The first sub-package contains the hierarchy of classes rooted by the class *Term* for all elementary terms presented in Section 5.3. The second sub-package contains, instead, other special terms (see next section) like numerical terms, collection terms (maps, sequences and
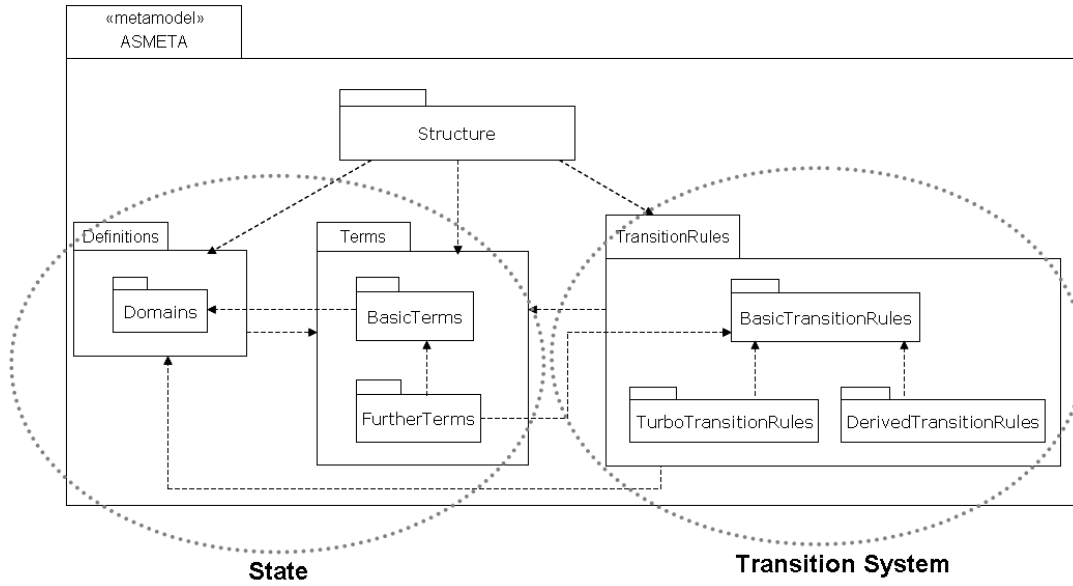
Figure 20 Package structure of the ASM Metamodel

bags), conditional and case terms, and variable binding terms.

– the `TransitionRules` package (or *the language of rules*) contains all possible transition rules schemes of Basic and Turbo ASMs. It is divided in tree packages: `BasicTransitionRules`, `TurboTransitionRules` and `DerivedTransitionRules`. The `BasicTransition-Rules` package contains the abstract class *Rule* and the hierarchy of classes rooted by the subclass *BasicRule*. The `TurboTransitionRules` package contains the hierarchy of classes rooted by the class *TurboRule*, also sub-class of *Rule*. Finally, all derived transition rules are contained in the `DerivedTransitionRules` package under the hierarchy of the class *DerivedRule*, sub-class of *Rule*; this last package is described in the next section.

Further MOF types are used to define the AsmM itself (for typing classes attributes), in particular to extend or simply rename some MOF data types. These types are (see Fig. 10): the enumeration `VariableKind`, the class `Arity` (which represents non-negative integer values) used to type the attribute indicating the arity of a function or rule, and the classes `DomainCollection`, `TermCollection`, and `RuleCollection` representing ordered MOF collections of type-domains, terms and rules, respectively. Note that, these data types do not represent the type system of the ASMs at model level.

11 Further Concepts

This section provides concepts which serve to enrich and complete the AsmM specification. New forms of type-domains, terms and derived rules are introduced as new classes in the corresponding packages of the metamodel.

11.0.1 The Definitions::Domains Package   We complete here the domain classification providing new type-domains: *basic type-domains* (supported by the AsmM standard library) for primitive data values like reals, integers, naturals, strings, etc.; *structured type-domains* for sequences and bags; and *enum-domains* to allow the user to introduce new concepts of type through finite enumerations. In addition, we introduce a notion of *generic type-domain* (see the class `AnyDomain` below) i.e. a domain which stands for any other type-domain.

The class model reflecting the complete domain classification is shown in Fig. 32 and 23. A detailed description of the new classes follows.

`ComplexDomain` is a singleton class (X1 in Tab. 14) whose unique instance represents the universe of all complex numbers. Similarly, the class `RealDomain` represents the universe of all real numbers; the class `IntegerDo-main` represents the universe of all integer numbers; the class `NaturalDomain` represents the universe of all natural numbers; the class `CharDomain` represents the universe of all characters ASCII and Unicode; the class `StringDomain` represents the universe of all strings over a standard alphabet (like ASCII or Unicode). All these classes are singleton too, and this is stated by OCL rules similar to the rule X1, not reported here for the sake of brevity.

An instance of the class `EnumTD` represents a type-domain defined by the user in terms of a finite enumeration. Elements of an enumeration are represented by instances of the class `EnumElement`. For example, one may define the enumeration $Color = \{Red, Green, Blue\}$ to introduce the new concept of "color". The enumeration elements *red*, *blue* and *green* are `EnumElement` instances.

An instance of the class `SequenceDomain` represents the set of all sequences over a specified type-domain D,
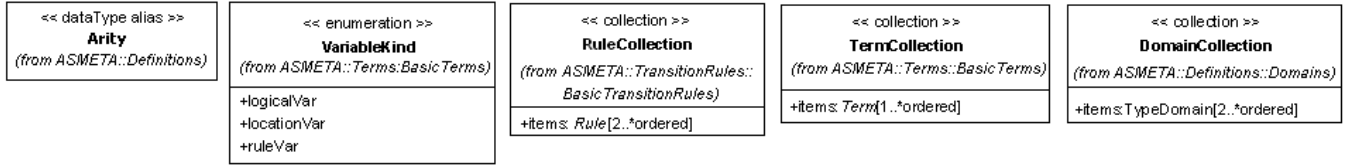
Figure 21 Extended MOF datatypes for the AsmM
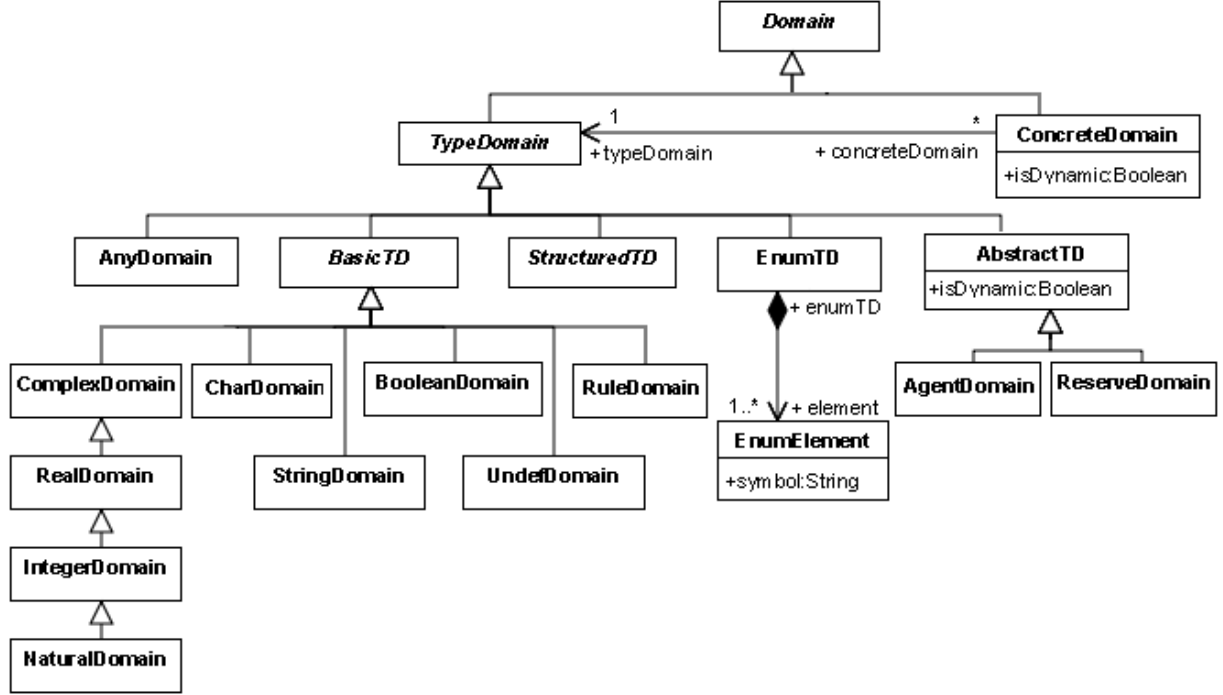
Figure 22 Domains (Part 1)
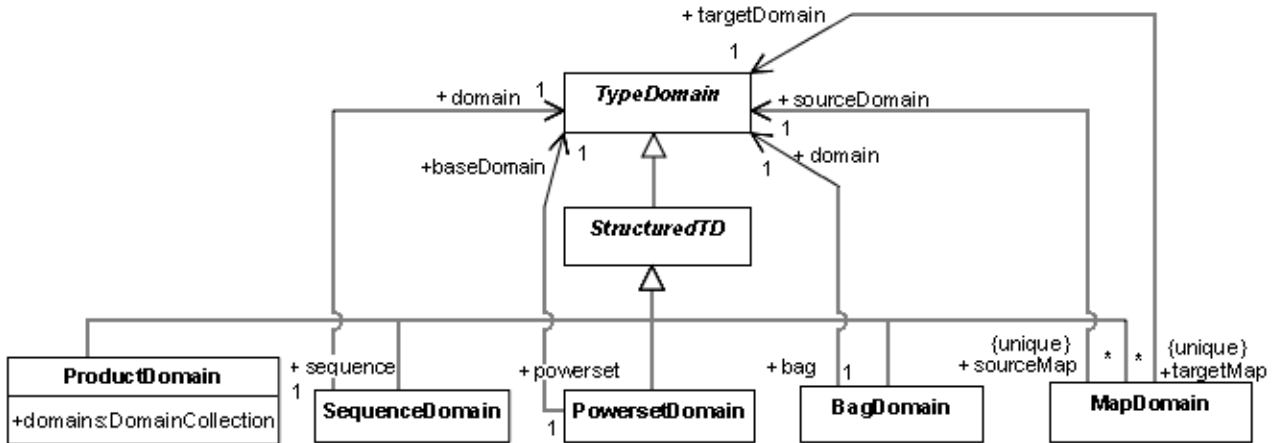
Figure 23 Domains (Part 2) – Structured domains

i.e. the set D^* of all sequences over D. A sequence is here considered like a mathematical ordered set which may contain duplicates.

An instance of the class `BagDomain` represents the domain of all finite bags over a specific type-domain D. A bag is like a mathematical set, which may contain duplicates; that is, the same element may be in a bag twice or more.

The class `MapDomain` represents the set of all finite maps between two specified type-domains (from `source-Domain` to `targetDomain`).

```
     context ComplexDomain inv:
X1:   allIstances->size() = 1
     ...
```

Table 14 Basic Domain Constraint

```
     context AnyDomain inv:
A1:   allIstances->exist(e|e.name='Any')
     context SetTerm inv:
A2:  if size = 0
     then domain.oclAsType(PowersetDomain).
          baseDomain.oclIsTypeOf(AnyDomain)
```

Table 15 Any Domain Constraints

An instance of the class `AnyDomain` represents a generic domain, i.e. a domain which stands for any other type-domain. A predefined `AnyDomain` instance named `Any` (defined in the AsmM standard library) is always instantiated (A1 in Tab. 15) and it is considered, among all type-domains, the most generic one. This genericity notion is used to abstract from the structure of specific domains when one wants to declare and use, for example, just one function for expressing a concept which applies to objects of various types, or also to specify the type-domain (which is mandatory) of empty collections like the empty set (A2 in Tab. 15). This abstraction mechanism is used to define many functions of the standard AsmM library. To declare, for example, a function *at* which returns the i-th element of a tuple, we can write as follows:

$static\ at : Prod(Prod(D_1, D_2), Natural) \rightarrow Any$

where D_1 and D_2 are two different generic domains, while *Any* stands for any domain (including D_1 and D_2). The introduction of new domains causes the redefinition of the query `compatible()` originally presented in Tab. 3 on page 10. Tab. 16 reports the new constraints to be added in order to deal with the domains introduced in this section. The compatibility is recursively checked taking into account the containment hierarchy of the structured type-domains (if any). The basic type-domains, enum type-domains, abstract type-domains, and instances of the `AnyDomain` class are considered as basis of the recursion. It is assumed, in particular, that the `Any` instance of the AnyDomain class is compatible to any domain.

During the parsing process all instances of the `Any-Domain` class are turned in specific type domain instances. These controlled side effects are carried out by the linguistic analyzer after checking the compatibility. For example, the compatibility test for the two domains $Prod(Powerset(D), D)$ and $Prod(Powerset(Integer), Integer)$ would yield true with D set to *Integer* at the end.

11.0.2 The `Terms::FurtherTerms` Package
Other constant terms, collection terms (maps, sequences, and bags),

```
     context ComplexTerm inv:
T10:  domain.oclIsTypeOf(ComplexDomain)
     context RealTerm inv:
T11:  domain.oclIsTypeOf(RealDomain)
     context IntegerTerm inv:
T12:  domain.oclIsTypeOf(IntegerDomain)
     context NaturalTerm Inv:
T13:  domain.oclIsTypeOf(NaturalDomain)
     context CharTerm inv:
T14:  domain.oclIsTypeOf(CharDomain)
     context StringTerm inv:
T15:  domain.oclIsTypeOf(StringDomain)

     context EnumTerm inv
T16:  domain.oclIsTypeOf(EnumTD)
T17:  domain.oclAsType(EnumTD).element ->
         exist(e:EnumElement | e.symbol = symbol)

     context ConditionalTerm inv:
T18:  self.compatible(thenTerm) and
        (elseTerm -> notEmpty() implies
         self.compatible(elseTerm))
T19:  guard.domain.oclIsTypeOf(BooleanDomain)

     context CaseTerm inv:
T20:  comparingTerm->size() = resultTerm->size()
T21:  resultTerms->forAll(t:Term|self.compatible(t))
        and self.compatible(otherwiseTerm)
T22:  comparingTerm->forAll(t:Term |
              comparedTerm.compatible(t))
```

Table 17 Term Constraints (Part 2)

variable-binding terms (let-terms, comprehension terms, and finite quantification terms), etc., are here provided as concrete subclasses of the *Term* class.

11.0.3 Further Constant Terms
An instance of the class `ComplexTerm` (see Fig. 24) represents a constant term which is interpreted as a complex number and belongs to the `ComplexDomain` (T10 in Tab. 17). Similarly, an instance of the class `RealTerm` is a constant term interpreted as a real number, an instance of the class `IntegerTerm` is a constant term interpreted as an integer number, an instance of the class `NaturalTerm` is a constant term interpreted as a natural number, an instance of the class `CharTerm` is a constant term interpreted as a character (ASCII or Unicode), and an instance of the class `StringTerm` is a constant term interpreted as a string over a standard alphabet (ASCII or Unicode).

An instance of the class `EnumTerm` is a constant term denoting an element of an enumeration type-domain (the interpretation of such a term yields the denoted element of that enumeration). The symbol denoted by an enum constant term is an element of the enumeration type-domain associated to the enum constant term (T17).

11.0.4 Further Extended Terms
The class `Conditional-Term` (see Fig. 25) represents a conditional term of form "**if** φ **then** t_{then} **else** t_{else} **endif**", where φ (denoted by the association end `guard`) is a term represent-

```
context Domain def: let compatible(d : Domain): Boolean =
-- as the previous version in Table 3 on page 10 plus
-- one is the predefined AnyDomain instance 'Any'
let any= AnyDomain.allIstances->select(e|e.name='Any') in self = any or d = any or
-- two SequenceDomain
(self.oclIsTypeOf(SequenceDomain) and d.oclIsTypeOf(SequenceDomain) and
      self.oclAsType(SequenceDomain).domain.compatible(d.oclAsType(SequenceDomain).domain)) or
-- two BagDomain
(self.oclIsTypeOf(BagDomain) and d.oclIsTypeOf(BagDomain) and
      self.oclAsType(BagDomain).domain.compatible(d.oclAsType(BagDomain).domain)) or
-- two MapDomain
(self.oclIsTypeOf(MapDomain) and d.oclIsTypeOf(MapDomain) and
   self.oclAsType(MapDomain).sourceDomain.compatible(d.oclAsType(MapDomain).sourceDomain) and
   self.oclAsType(MapDomain).targetDomain.compatible(d.oclAsType(MapDomain).targetDomain))
```

Table 16 Redefinition of `compatible`
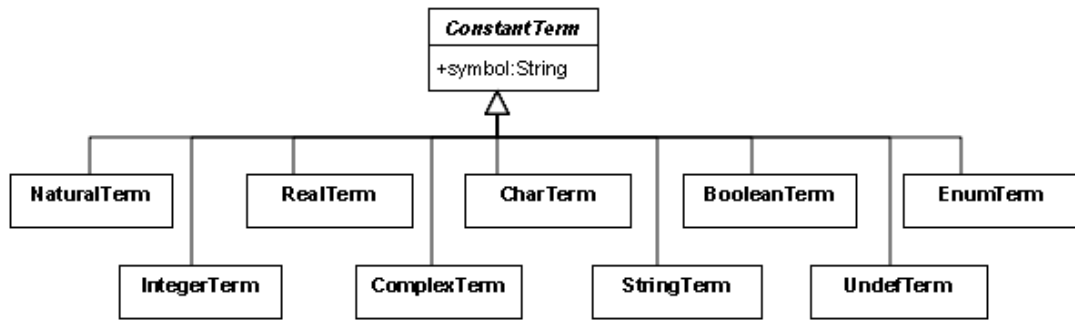
Figure 24 Constant Terms
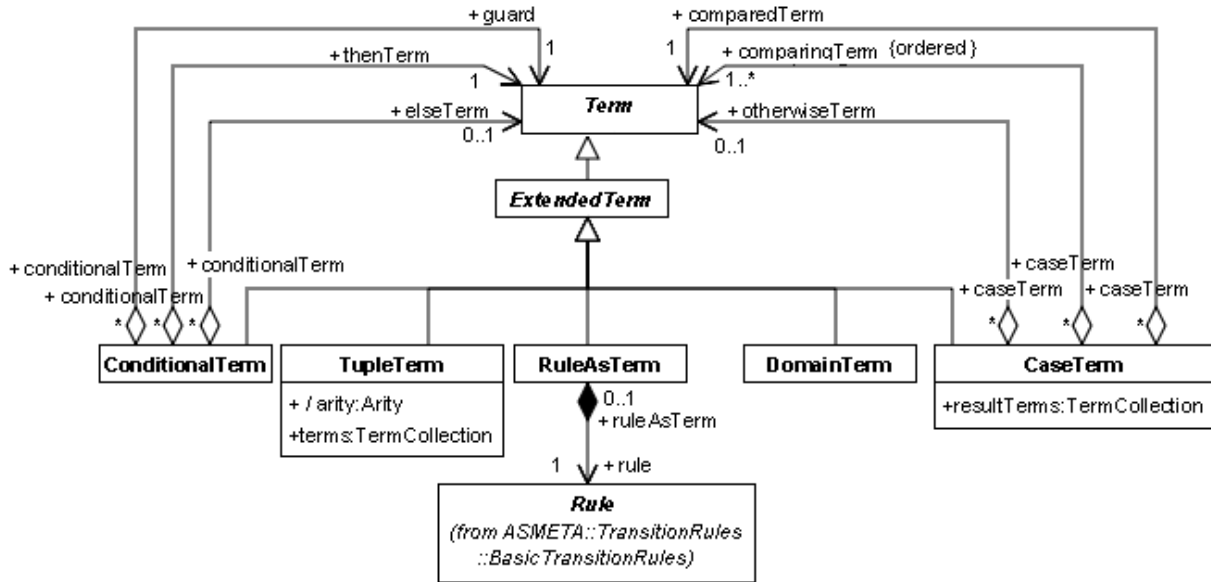
Figure 25 Extended Terms

ing the condition, and t_{then} and t_{else} are terms denoted by the association ends `thenTerm` and `elseTerm`, respectively. If t_{else} is omitted it is assumed by default that $t_{else} \equiv undef$. A conditional term must be compatible to the term t_{then}, and to t_{else} (if any) (T18), and the type-domain of the term denoting the condition is the boolean domain (T19).

The class `CaseTerm` (see Fig. 25) represents a case term of the form

> **switch** t
>   **case** $t_1 : s_1$ ... **case** $t_n : s_n$ **otherwise** s_{n+1}
> **endswitch**

where t (denoted by the association end `comparedTerm`), $t_1, \ldots, t_n$ (denoted by the association end `comparingTerm`),

$s_1, \ldots, s_n$ (denoted by the attribute `resultTerms`), and s_{n+1} (denoted by the association end `otherwiseTerm`) are terms. If s_{n+1} is omitted it is assumed that $s_{n+1} \equiv undef$. The number of left-hand side terms t_i of the case-clauses must be equal to the number of right-hand side terms s_i (T20). Moreover, a case term must be compatible to the right-hand side terms s_i of the case-clauses, and to the term s_{n+1} of the otherwise-clause (T21), and the term t to match is compatible to the left-hand terms t_i of the case-clauses (T22).

11.0.5 Further Collection Terms   The class `BagTerm` (see Fig. 26) represents a bag $< t_1, \ldots, t_n >$ where $t_1, \ldots, t_n$ (denoted by the association end `term`) are terms of the same type representing the elements of the bag. A bag may be empty and may contain duplicates. The size of a bag term is the number of terms t_i (E10 in Tab. 18). A bag term has domain a `BagDomain` (E11). Let D the domain of the `BagDomain` associated to a bag term. D must be compatible with the domain of every bag item t_i (E12). Note that, if the bag is empty, then D can be any type domain.

The class `SequenceTerm` (see Fig. 26) represents a sequence $[t_1, .., t_n]$, i.e. a collection of elements $t_1, .., t_n$ (denoted by the association end `term`) of the same type with some order defined on it. A sequence may be empty and may contain duplicates. The size of a sequence is the number of terms t_i (E13). A sequence term has domain a `SequenceDomain` (E14). Let D the domain of the `Se-quenceDomain` associated to a sequence term. D must be compatible with the domain of every sequence item t_i (E15). Note that, if the sequence is empty, then D can be any type domain.

The subclass `MapTerm` represents a map term $\{s_1 \to t_1, \ldots, s_n \to t_n\}$ as composition of `pairs` (s_i, t_i) of terms. Every term denoting the association $s_i \to t_i$ is a `Tu-pleTerm` and must be a pair (E16). The size of the map term is the number of pairs (s_i, t_i) (E17), and its domain is a `MapDomain` (E18).

In a map term, all terms $s_1, \ldots, s_n$ are of the same type, as well as all terms $t_1, \ldots, t_n$. Let the Cartesian product $S_i \times T_i$ be the domain of the i-th pair (s_i, t_i) of a map term. The domain of the map term is a map domain from a type-domain D_S to a type-domain D_T, where D_S is compatible to S_i and D_T is compatible to T_i for every pair within the map (E19). If the map term is empty, D_S and D_T can be any type domain.

Note that a function initialization `body` (see Fig. 12) can be a `MapTerm`. If the *body* is a map term from domain M_1 to domain M_2, and the function to initialize is not 0-ary, M_1 must be compatible to the function domain, and M_2 must be compatible to the function codomain (E20). A similar constraint must be introduced for the `FunctionDefiniton` class.

11.0.6 VariableBindingTerm   The abstract class *Finite-QuantificationTerm* (see Fig. 27) models the basic struc-

```
  context FiniteQuantificationTerm inv:
Q1: domain.oclIsTypeOf(BooleanDomain)
    Sequence{1..variable->size()}->forAll(i:Integer|
Q2: ranges.items->at(i).oclIsTypeOf(PowersetDomain)
    and
Q3:  variable->at(i).domain = ranges.items->at(i).
          oclAsType(PowersetDomain).baseDomain)
Q4: if self.guard->notEmpty() then
    guard.domain.oclIsTypeOf(BooleanDomain) endif
```

Table 19 Finite Quantification Term Constraints

ture of quantification terms $\exists$, $\exists!$, and $\forall$. Basically, a quantification term is characterized by a list of variables x_i (all different and denoted by the association end `variable`), by a collection of terms D_i (listed in the `ranges` attribute) representing the sets in which x_i varies, and an optional term $\varphi_{x_1, \ldots, x_n}$ (denoted by the association end `guard`) representing a boolean-valued expression with occurrences of the variables x_i. If the condition $\varphi_{x_1, \ldots, x_n}$ is omitted, it is assumed that $\varphi_{x_1, \ldots, x_n} \equiv true$.

A finite quantification term is a boolean (Q1 in Tab. 19). The domain of each term D_i is a powerset (Q2) and the domain of x_i is set to the `baseDomain` of D_i (Q3). The domain of the term $\varphi_{x_1, \ldots, x_n}$ representing the selection criteria (if any) must be the boolean domain (Q4).

The concrete subclasses `ExistTerm`, `ExistUnique-Term` and `ForallTerm` inherit the structure and the constraints of the class *FiniteQuantificationTerm* to represent finite existential quantifications and finite universal quantifications of the form, respectively:

exist x_1 **in** D_1, ..., x_n **in** D_n **with** $\varphi_{x_1, \ldots, x_n}$,
exist unique x_1 **in** D_1, ..., x_n **in** D_n **with** $\varphi_{x_1, \ldots, x_n}$,
forall x_1 **in** D_1, ..., x_n **in** D_n **with** $\varphi_{x_1, \ldots, x_n}$.

The class `LetTerm` (see Fig. 27) represents a let-term. The basic form of such a term is "**let** $(x = t)$ **in** t_x **endlet**". It is used to allow the evaluation of the term t_x (given by the association end `body`) with the given variable x (denoted by the association end `variable`) bound to the value resulting from the interpretation of the term t (denoted by the association end `assignment-Term`). This semantics can be easily generalized to a let-term of the form

let $(x_1 = t_1, \ldots, x_n = t_n)$ **in** $t_{x_1, \ldots, x_n}$ **endlet**,
according to the following reduction rule

let $(x_1 = t_1, \ldots, x_n = t_n)$ **in** $t_{x_1, \ldots, x_n}$ **endlet** $\equiv$
let $(x_1 = t_1)$ **in let** $(x_2 = t_2)$ **in**
... **let** $(x_n = t_n)$ **in** $t_{x_1, \ldots, x_n}$ **endlet** ... **endlet endlet**.

The type-domain of a let-term is the type-domain of the term $t_{x_1, \ldots, x_n}$ (L1 in Tab. 20). Moreover, the number of terms t_i must be equal to the number of variables x_i (L2), and each variable x_i must be compatible to each term t_i (L3).
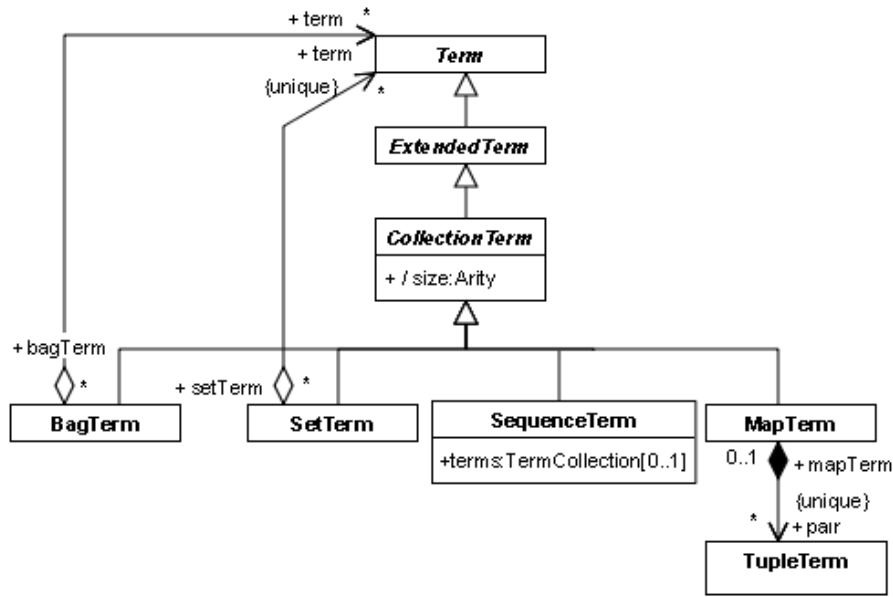
Figure 26 Collection Terms

```
      context BagTerm inv:
E10:    size = term->size()
E11:    domain.oclIsTypeOf(BagDomain)
E12:    if size = 0
        then domain.oclAsType(BagDomain).domain.oclIsTypeOf(AnyDomain)
        else term->forAll(t:Term | t.domain.compatible(domain.oclAsType(BagDomain).domain) )
        endif

      context SequenceTerm inv:
E13:    size = term->size()
E14:    domain.oclIsTypeOf(SequenceDomain)
E15:    if size = 0
        then domain.oclAsType(SequenceDomain).domain.oclIsTypeOf(AnyDomain)
        else term->forAll(t:Term | t.domain.compatible(domain.oclAsType(SequenceDomain).domain) )
        endif

      context MapTerm inv:
E16:    pair->forAll(p:TupleTerm | p.term->size()= 2)
E17:    size = pair->size()
E18:    domain.oclIsTypeOf(MapDomain)
E19:    if size = 0
        then domain.oclAsType(MapDomain).sourceDomain.oclIsTypeOf(AnyDomain) and
             domain.oclAsType(MapDomain).targetDomain.oclIsTypeOf(AnyDomain)
        else pair->forAll(p:TupleTerm | domain.oclAsType(MapDomain).sourceDomain.
             compatible(p.domain.oclAsType(ProductDomain).domain->at(1)) and
             domain.oclAsType(MapDomain).targetDomain.compatible(p.domain.oclAsType(ProductDomain).
             domain->at(2)))
        endif

     context FunctionInitialization inv:
E20:   let M: Domain = body.domain in
         if (M.oclIsTypeOf(MapDomain)) then initializedFunction.domain->notEmpty() and
         M.oclAsType(MapDomain).sourceDomain.compatible(initializedFunction.domain) and
         M.oclAsType(MapDomain).targetDomain.compatible(initializedFunction.codomain)
         endif
```

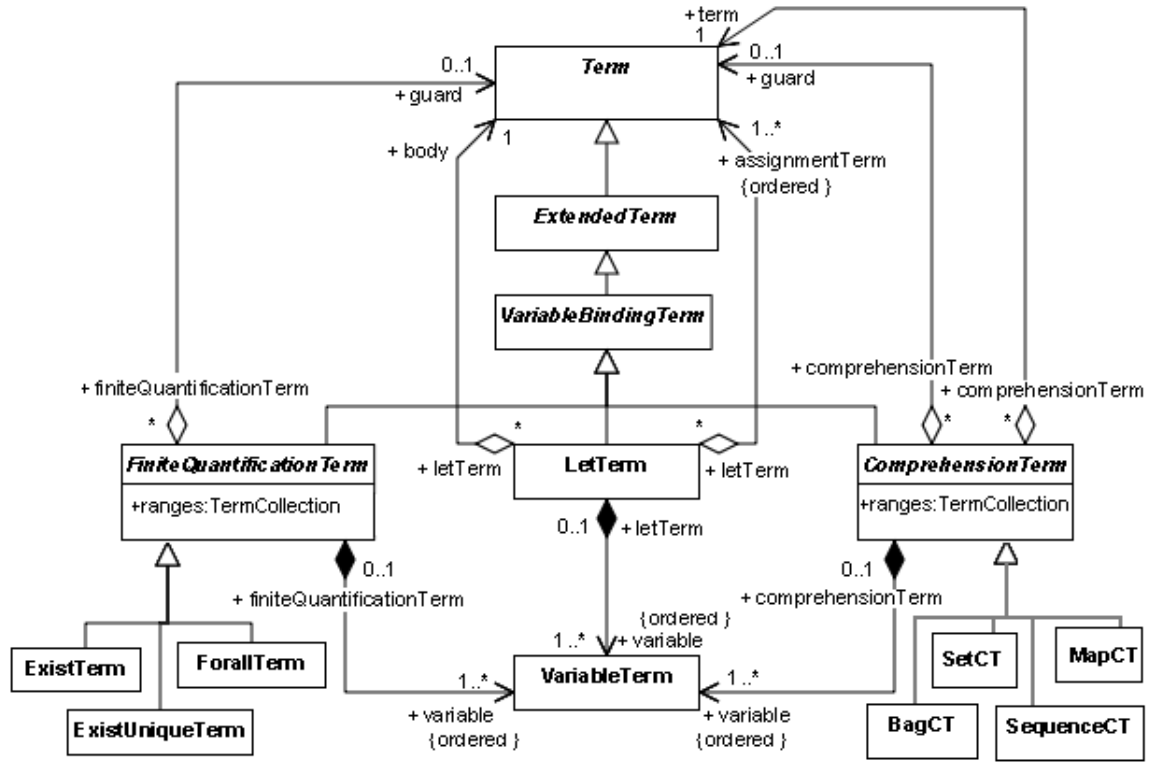**Table 18** Collection Term Constraints

Figure 27 Variable Binding Terms

```
    context LetTerm inv:
L1: domain = body.domain
L2: assignmentTerm->size() = variable->size()
L3: Sequence1..variable->size()->forAll(i:Integer|
    variable->at(i).compatible(assignmentTerm->at(i))
```

Table 20 LetTerm Constraints

To represent a collection of elements in a given reciprocal relationship and satisfying a certain property, we introduce *comprehension terms* (borrowing several constructs already used in ASM-SL [17]). The class `ComprehensionTerm` (see Fig. 27) is abstract. Its concrete subclasses `SetCT`, `BagCT`, `SequenceCT` and `MapCT` inherit its structure and constraints to represent comprehension terms respectively for sets, bags, sequences and maps. A comprehension term has the following forms:

$\{t_{x_1,...,x_n} \mid x_1 \textbf{ in } D_1, ..., x_n \textbf{ in } D_n \textbf{ with } \varphi_{x_1,...,x_n}\}$
   denotes a set comprehension term, which yields a set;
$< t_{x_1,...,x_n} \mid x_1 \textbf{ in } B_1, ..., x_n \textbf{ in } B_n \textbf{ with } \varphi_{x_1,...,x_n} >$
   denotes a bag comprehension term, which yields a bag;
$[t_{x_1,...,x_n} \mid x_1 \textbf{ in } S_1, ..., x_n \textbf{ in } S_n \textbf{ with } \varphi_{x_1,...,x_n}]$
   denotes a sequence comprehension term, which yields a sequence;
$\{s_{x_1,...,x_n} \rightarrow t_{x_1,...,x_n} \mid x_1 \textbf{ in } D_1, ..., x_n \textbf{ in } D_n \textbf{ with } \varphi_{x_1,...,x_n}\}$
   denotes a map comprehension terms, which yields a map.

The main term $t_{x_1,...,x_n}$ (determined by the association end `term`) denotes the relationship existing among elements of the collection. It contains occurrences of the variables x_i, which are linked by the association end `variable`. D_i, B_i, and S_i are terms whose interpretation yields the collections in which variables vary and are listed in the `ranges` attribute, and $\varphi_{x_1,...,x_n}$ is a term (denoted by the association end `guard`) representing a boolean-valued expression with occurrences of the variables x_i. If $\varphi_{x_1,...,x_n}$ is omitted it is assumed that $\varphi_{x_1,...,x_n} \equiv true$.

The following OCL constraints must hold for the abstract class `ComprehensionTerm` (and therefore for any sub-class). The type-domain of the term $\varphi_{x_1,...,x_n}$ representing the selection criteria is the boolean domain (H1 in Tab. 21). The number of variables x_i is equal to the number of terms D_i in `ranges` (H2).

The type-domain of a set (resp. bag, resp. sequence) comprehension term is the powerset domain (resp. bag domain, resp. sequence domain) over the type-domain of the main term $t_{x_1,...,x_n}$ (H3) (resp. (H5), resp. (H7)).

We assume (as in [17]) that the type of the collection terms in `ranges`, D_i, B_i, and S_i, is equal to the type of the comprehension term: D_i is a set for a set comprehension term and its domain is therefore the `Powerset` (H4), B_i is a bag for a bag comprehension term and its domain is therefore the `BagDomain` (H6), and S_i is a sequence for a sequence comprehension term and its domain is therefore the `SequenceDomain` (H8). The domains of variables x_i must be set accordingly.

```
     context ComprehensionTerm inv:
H1:  if guard->notEmpty() then guard.domain.oclIsTypeOf(BooleanDomain) endif
H2:  variable->size() = ranges.items->size()

     context SetComprehensionTerm inv:
H3:  domain.oclIsTypeOf(PowersetDomain) and domain.oclAsType(PowersetDomain).domain = term.domain
H4:  Sequence{1..variable->size()}-> forAll(i:Integer |
         ranges.items->at(i).oclIsTypeOf(PowersetDomain) and
          variable->at(i).domain = ranges.items->at(i).oclAsType(PowersetDomain).baseDomain )

     context BagComprehensionTerm inv:
H5:  domain.oclIsTypeOf(BagDomain) and domain.oclAsType(BagDomain).domain = term.domain
H6:  Sequence{1..variable->size()}-> forAll(i:Integer |
         ranges.items->at(i).oclIsTypeOf(BagDomain) and
          variable->at(i).domain = ranges.items->at(i).oclAsType(BagDomain).domain )

     context SequenceComprehensionTerm inv:
H7:  domain.oclIsTypeOf(SequenceDomain) and domain.oclAsType(SequenceDomain).domain = term.domain
H8:  Sequence{1..variable->size()}-> forAll(i:Integer |
         ranges.items->at(i).oclIsTypeOf(SequenceDomain) and
          variable->at(i).domain = ranges.items->at(i).oclAsType(SequenceDomain).domain )

     context MapComprehensionTerm inv:
H9:  term.oclIsTypeOf(TupleTerm) and term.oclAsType(TupleTerm).term->size()=2
H10: domain.oclIsTypeOf(MapDomain) and
     domain.oclAsType(MapDomain).sourceDomain = term.domain.oclAsType(ProductDomain).domains->at(1) and
     domain.oclAsType(MapDomain).targetDomain = term.domain.oclAsType(ProductDomain).domains->at(2)
H11: Sequence{1..variable->size()}-> forAll(i:Integer |
         ranges.items->at(i).oclIsTypeOf(PowersetDomain) and
          variable->at(i).domain = ranges.items->at(i).oclAsType(PowersetDomain).baseDomain )

     context TupleTerm inv:
H12: terms -> forAll( t: Term | not t.domain.oclIsTypeOf(FiniteQuantificationTerm) and
         not t.domain.oclIsTypeOf(LetTerm) and
         not t.domain.oclIsTypeOf(ConditionalTerm) and not t.domain.oclIsTypeOf(CaseTerm)
```

Table 21 Comprehension Terms Constraints

For a map comprehension term, $s_{x_1,\ldots,x_n}$ and $t_{x_1,\ldots,x_n}$ are modelled by an unique tuple term which is still referenced by the association end `term` and which must be a pair of terms (to represent the two terms $s_{x_1,\ldots,x_n}$ and $t_{x_1,\ldots,x_n}$, respectively) (H9). Let the Cartesian product $A \times B$ be the type-domain of the pair $(s_{x_1,\ldots,x_n}, t_{x_1,\ldots,x_n})$. The type-domain of a map comprehension term is therefore a map domain from A to B (H10). Terms in `ranges`, D_i, representing where variables x_i vary, must be powersets and the domains of variables x_i must be set accordingly (H11).

The only terms allowed in a tuple term are basic terms, collection terms, and extended terms except `ConditionalTerms`, `CaseTerms`, `FiniteQuantificationTerm`, and `LetTerm`. The only terms allowed in map terms, sequence terms, set terms, bag terms, and comprehension terms (bags, sets, sequences, and maps) are basic terms, collection terms, and extended terms except `ConditionalTerms`, `CaseTerms`, and all `VariableBindingTerms`. For this reason we write several OCL constraints as the following one given for a `TupleTerm` (H12).

11.0.7 The `TransitionRules::DerivedTransitionRules` Package Other transition rule schemes, derived from the basic and the turbo ones, are defined (see Fig.

28) as subclasses of two abstract classes called *DerivedRule* and *DerivedTurboRule*, respectively. Although they could be easily expressed at model level in terms of other existing rule schemes, we decided to include them in the metamodel for their wide spread use.

The class `CaseRule` models a case rule as in

switch t
    case $t_1 : P_1, \ldots,$ **case** $t_n : P_n$
    otherwise P_{n+1}
endswitch

where $t, t_1, \ldots, t_n$ are terms (linked by the association ends `term` and `caseTerm`) and $P_1, \ldots, P_n, P_{n+1}$ are rules (listed in `caseBranches`). If P_{n+1} is omitted, then it is assumed that $P_{n+1} \equiv skip$. This rule scheme allows to select the rule to execute according to the following: execute the rule P_i corresponding to the first term t_i, for $i = 1..n$, such that the value of t is equal to the value of t_i; if no match is found, execute P_{n+1}. A case-rule is a derived rule, since it is easily reducible to a combination of conditional rules.

In a case rule, the number of terms t_i must be equal to the number of rules P_i (K1 in Tab. 22), and the term t is compatible to every term t_i of the case-clauses (K2).

The class `IterativeWhileRule` represents a rule of form "**while** φ **do** R" where φ is a term representing a
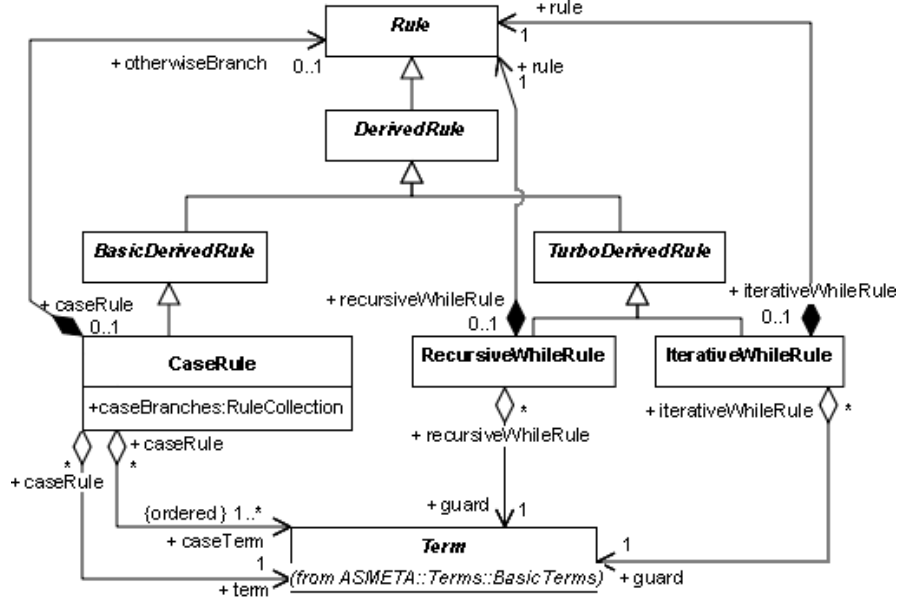
Figure 28 Derived transition rules

```
    context CaseRule inv:
K1:  caseTerm->size() = caseBranches.items->size()
K2:  caseTerm->forAll(t:Term |term.compatible(t))

    context IterativeWhileRule inv:
K3:  guard.domain.oclIsTypeOf(BooleanDomain)

    context RecursiveWhileRule inv:
K4:  guard.domain.oclIsTypeOf(BooleanDomain)

K5: context TermAsRule inv:
     term.domain.oclIsTypeOf(RuleDomain)
```

Table 22 Rule Constraints (Part 3)

guard, which must be a boolean (K3), and R is a transition rule (linked by the association end `rule`). It means: repeat the execution of the rule R as long as it produces a non-empty update set and the condition φ holds. If the iteration of R reaches an inconsistent update set or yields an infinite sequence of consistent non-empty update sets, then the state resulting from executing the while loop is not defined (the case of divergence of the while loop). An iterative-while rule is a derived rule since it can be defined in terms of a turbo iterate rule as follows: **while** φ **do** $R \equiv$ **iterate if** φ **then** R **endif enditerate**.

The class `RecursiveWhileRule` represents a rule of form "**whileRec** φ **do** R" where φ is a term representing a `guard`, which must be a boolean (K4), and R is a transition rule (linked by the association end `rule`). It means: repeat the execution of the rule R as long as the condition φ holds. This rule leads to a termination (success) only if the condition φ eventually become false and R does not diverge. A recursive-while rule is considered a derived rule since it can be defined in terms of a seq-rule as follows:

whileRec φ **do** $R \equiv$
  if φ **then seq**
       R
      whileRec φ **do** R
    endseq
  endif.

In addition to the derived rules presented above, a special transition rule, represented by the class `TermAsRule` (see Fig. 29), has been introduced as *wrapper* for terms which are interpreted as transition rules (like a rule variable, or a function application which yields a transition rule as value) to be used in places where a rule body is expected. For example, consider the two following scenarios.

- A rule variable `r` is a formal parameter of a rule declaration as in `P(r) = body`. The variable `r`, which is a term, will eventually appear within the `body` of `P(r)` where a rule is expected. In this case it is converted from `VariableTerm` to `Rule` by a `TermAsRule`.
- In order to obtain the main rule of an agent, the function $program : Agent \rightarrow Rule$ can be applied. For an agent a, $program(a)$ is an application of the function $program$, thus it is a `FunctionTerm` whose domain is the `RuleDomain`. To use $program(a)$ as a rule, it is converted from `FunctionTerm` to `Rule` by a `TermAsRule`.

In both cases, a term `t` – a variable term `r` in the first case, and the function term $program(a)$ in the second case – is considered a *TermAsRule* rule, say R_t. A `TermAsRule` is linked to its term `t` by the association end `term` and `t` must be an actual rule, i.e. its domain must be the `RuleDomain` (K5). The effect is that, when the rule R_t is fired, it is substituted by `t`, i.e. the term rep-

resenting the actual rule to be fired − rule variable term **r**, in the first case, and *program(a)*, in the second case.

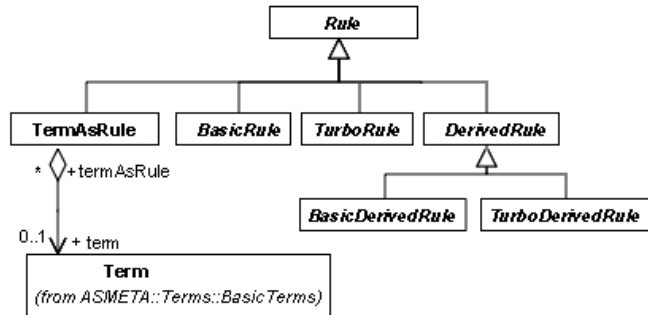Fig. 29 shows the complete classification of the ASM transition rules under the root class *Rule* .

Figure 29 All transition rules

12 AsmM derivatives

In this section we present some derivative artifacts obtained from the AsmM: the AsmM-specific Java Metadata Interfaces used to access and manipulate models; the AsmM-specific XMI interchange format; the AsmM concrete syntax and the MOF-to-EBNF mapping rules applied to derive it; the parser which processes ASM models written in the AsmM concrete syntax and creates objects of the corresponding AsmM abstract syntax in a MOF-based model repository.

12.1 Generation of specific JMI APIs

The JMI standard, already presented in Section 2.4, is based on the MOF 1.4 specification and defines a Java application program interface (API) for managing and accessing metadata. It implements a dynamic, platform-neutral infrastructure that enables the creation, storage, access, discovery, and exchange of metadata, that in our case are ASM models. It can be used in client programs written in Java which want to manipulate ASM models: to read model structure, to modify parts of the specification and create new elements. It can be used by tool developers to speed up the creation of tools supporting ASMs and by researchers to experiment algorithms over ASMs.

The JMI for the AsmM was obtained automatically by the following steps. First, we have drawn the AsmM with the Poseidon UML tool and saved it in the UML XMI format (without graphic information). Then, we have converted it to the MOF 1.4 XMI by means of the UML2MOF transformation tool provided by the Netbeans MDR project. Finally, we have loaded the MOF model in the MDR framework of Netbeans and generated the JMI interfaces from it.

MOF packages, classes and associations are mapped to JMI as specified by [32]. For every package *P* in the metamodel a *package proxy* interface **PPackage** is generated which gives access to all the metaobjects within the package *P*. It contains methods for accessing all proxy objects for all directly nested or clustered packages and contained classes and associations. In particular, the outermost package of the metamodel represents the root package which serves as entry point to access and create all the other objects. In our case, we have the root **AS-METAPackage** which can be created by calling the **getAS-META()** method of the **MDRConnector** provided by the MDR framework[10]. Through the **ASMETAPackage**, reported in Fig. 30, one can access to every sub package and through sub packages to every object in a AsmM model.

For every class *X* in the metamodel two interfaces are generated, one, **XClass**, representing a static context of the class or *class proxy* interface, and the other, **X**, representing individual instances of the class or *instance* interface. The class proxy interface **XClass** contains factory methods for creating new instances of **X** and methods for accessing/invoking classifier-level attributes/operations. The instance interface **X** contains getter and setter methods for accessing instance-level attributes and references (i.e. association ends), and invoking instance-level operations. For example, for the class **Asm** of the AsmM, JMI introduces the proxy interface **AsmClass** (which contains a factory method **createAsm()** that creates a new **Asm** object) and the **Asm** instance interface, shown in Fig. 30.

Another kind of interfaces are the *association proxy* interfaces. They are generated for each association in the metamodel and provide operations for querying and updating the links that belong to the given association. For example, the association between an **Asm** and its **main-rule** is mapped by the class **AAsmMainrule** reported in Fig. 30.The associations may be viewed (and updated) also on one or other of the ends, and there may be some form of order and cardinality constraints.

The implementation of all of these interfaces is provided by the MDR MOF repository at runtime automatically. All the metadata are stored and managed by MDR in a pluggable storage (typically a b-tree database). For more usage details and for the MOF's general computational semantics of the MOF to Java mapping, see [32].

12.2 Generation of a specific XMI format

The XMI format (see Section 2.3) has been also generated automatically from the AsmM in the MDR framework. According to the rules specified by the MOF 1.4 to XMI 1.2 mapping specification [55], a XML document type definition file, commonly named DTD, has been generated from the AsmM.

---

[10] The reference to the root package proxy is obtained in a vendor specific way, typically using some kind of lookup mechanism.

```
/** ASMETAPackage interface **/
public interface AsmetaPackage extends javax.jmi.reflect.RefPackage{
  public asmeta.terms.TermsPackage        getTerms();
  public asmeta.structure.StructurePackage     getStructure();
  public asmeta.transitionrules.TransitionRulesPackage getTransitionRules();
  public asmeta.definitions.DefinitionsPackage      getDefinitions();
}

/** Asm class proxy interface **/
public interface AsmClass extends javax.jmi.reflect.RefClass {
  public Asm createAsm();
  public Asm createAsm(String name, boolean isAsynchr);
}
/** Asm object instance interface **/
public interface Asm extends javax.jmi.reflect.RefObject{
  public java.lang.String getName();
  public void setName(java.lang.String newValue);
  /* methods to set and get the isAsynch attribute, the Header, the InitialStates, the default
     InitialState, the mainRule, and the Body */
}

/** AAsmMainrule association proxy interface. **/
public interface AAsmMainrule extends javax.jmi.reflect.RefAssociation {
 public boolean exists(asm.asmdefinitions.RuleDeclaration mainrule, asm.asmstructure.Asm asm);
 /* methods getMainrule to get the main rule of an Asm, getAsm to obtain the Asm of a main rule,
     add, and remove */
}
```

Figure 30 JMI fragments for the AsmM

The XMI representation depends on the AsmM meta-model. Document generation is based on XML element containment. For each object the element start tag is generated from the object's metaclass name, and the element attribute `xmi.id` provides a unique identifier for it in the entire document. Long names of XMI elements with several dots may seem less legible than an usual textual notation adopted to write specification; but clearly the XMI format is not for human consumption. It has not to be confused with the "concrete syntax" used by modelers to write their models. It has to be intended, instead, as an effective hard code to be automatically generated for interchanging purposes only.

A XMI-based model interchange format has more advantages than a pure XML interchange format:

– only a slight knowledge of XML is required since the ASM DTD/schema is automatically derived from the MOF-compliant metamodel;
– several XML-based technologies and frameworks already exist, and are capable of producing automatically the corresponding XMI DTD/schema from a MOF compliant metamodel, significantly reducing the time spent to design or simply update a XML DTD/schema;
– the definition of a XMI DTD or schema is guided by a more abstract and standard process with clear-cut separation of technology dependent concepts from the independent concepts;

– the graphical diagrams representing the MOF compliant metamodel provide a more faithful readable format than a purely textual XML DTD or schema, because they are written in the style of the widely used and standardized object oriented notation;
– the standard MOF has more capability to represent complex, semantically rich, hierarchical metadata.

12.3 From MOF to EBNF: derivation of a concrete syntax for the AsmM

A MOF-compliant metamodel provides an abstract syntax for a language with the advantage of deriving (through mappings or projections) different alternative concrete notations, textual or graphical. We believe that 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 specify different nodes relationships (generalizations, aggregations, compositions, and so on). 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.

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 conforming to the AsmM, they are general enough and do not rely on the specific domain language.

In the sequel, we call Asm^2L (AsmM Language) a concrete syntax derived from the AsmM as a textual notation. A preliminary version of the Asm^2L language can be found in [44], under the name of AsmM-CS (AsmM Concrete Syntax). Its EBNF specification is a set of *derivation* (or *production*) *rules* of the form <symbol> ::= <expression with symbols> where <symbol> is a nonterminal, and the expression consists of sequences of symbols and/or sequences separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are terminals, or *keywords*, which we convey to denote as literal symbols enclosed in double quotes.

For the MOF to EBNF mapping, we take into account all MOF 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-modelled, 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 Asm^2L language. This parser is more than a grammar checker: it is able to process ASM models written in Asm^2L 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 [11] 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 obtained with this process can be further optimized and enriched. For example, suitable methods were added to the Asm^2L language 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.

In the following sections, we give the set of **rules** on how to map MOF constructs into EBNF and into JavaCC.

12.3.1 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*[11] 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 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.

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.

---

[11] 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.
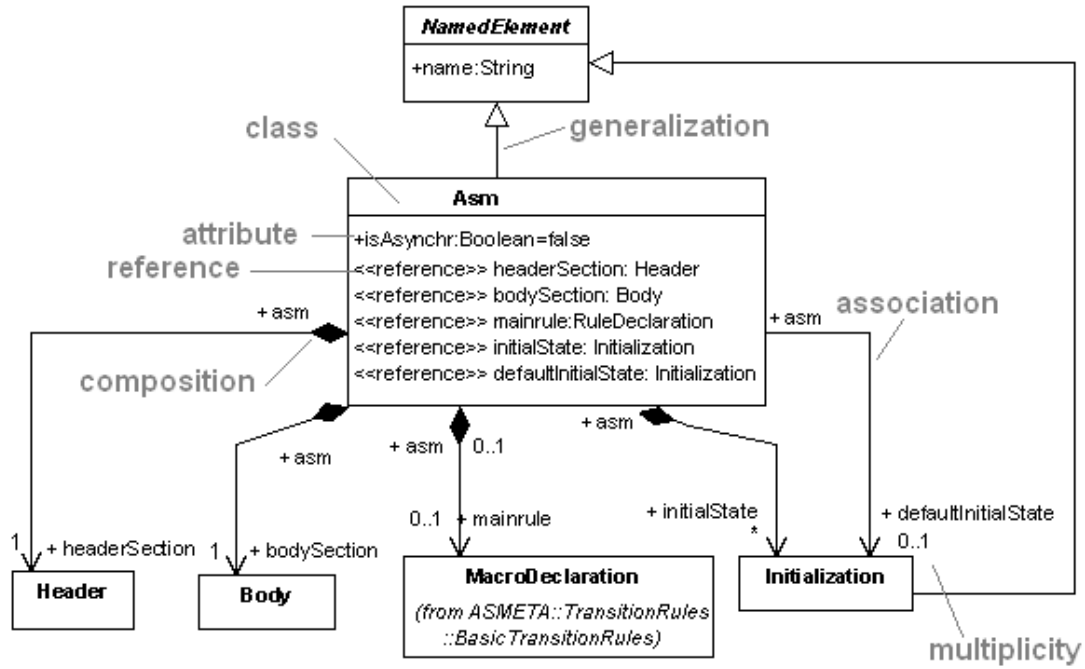
Figure 31 Example of class with attributes, references and associations

Example Fig. 31 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 `main-rule`, 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.

12.3.2 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}`.

12.4 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.

12.4.1 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. 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. 12.4.3) 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. 31, by **rule 4** and **rule 5**, we introduce the following EBNF derivation rule and JavaCC method:

Asm ::= ("asyncr")? "asm" <ID_Asm>

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

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. 32 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 =
 definitionsPack.getDomains().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*[12] 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.

12.4.2 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[13] if it is navigable, if there is no explicit MOF reference for that end within

---

[12] 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.

[13] We take this definition from the UML profile for MOF 1.4 by the MDR framework [4]. It is used to automatically imply MOF references by association ends. MOF references are implied, in fact, by each *eligible* UML association end.
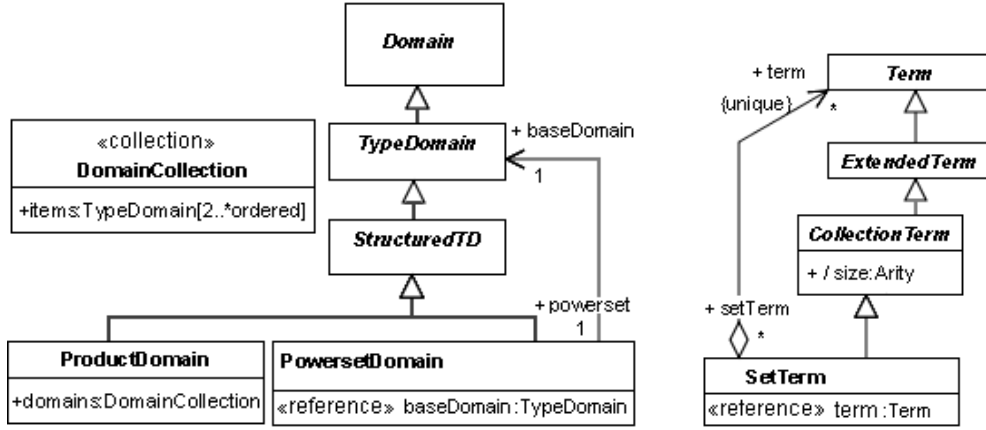
Figure 32 Example of a collection data type and of a multi-valued reference

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.

12.4.3 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.

Example 1 In Fig. 32, 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 =
definitionsPack.getDomains()
  .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 [14] 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. 32 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.

---

[14] 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.

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 = termsPack.getBasicTerms().
  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;}}
```

Example 3 By the rules above, the production rule for the Asm class in Fig. 31 can be completed as follows.

EBNF: Asm ::= ("isAsyncr")? "asm" <ID_Asm> Header Body ("main" RuleDeclaration)? (Initialization)* ("default" Initialization)?

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

12.4.4 Generalization Hierarchies of MOF classes are modeled by generalizations. We have to distinguish between a generalization from an *abstract* class and a generalization from a *concrete* class.
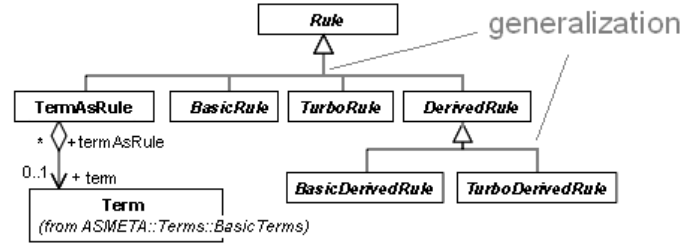

Figure 33 Example of Generalization from an abstract class

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. 33 presents an example of generalization from an abstract class. It 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;}}
```

Rule 17: In case classes $C_1, .., C_n$ inherit from a *concrete* class C, we introduce a production rule C::= C$_c$ | C$_1$|...|C$_n$ to capture the choice in the class hierarchy, and a production rule for the new non terminal symbol C$_c$ built according to the content (attributes and references) 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$_c$.

12.4.5 Constraint OCL constraints are not mapped to EBNF concepts. Appropriate parser actions, instead, 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.

```
asm FLIP_FLOP import STDL/StandardLibrary
signature:
 domain State subsetofNatural
 controlled ctl_state : State
 monitored high : Boolean
 monitored low : Boolean
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
default init initial_state:
 function ctl_state = 0
 function high = false
 function low = false
```

Figure 34 Flip-Flop Specification

The complete Asm^2L grammar derived from the AsmM is reported in appendix. Fig. 34 shows the specification written in Asm^2L of a Flip-Flop device. The model originally presented in [16, page 47] contains two rules: the first one (FSM) models a generic finite state machine and the second one (FLIPFLOP) instantiates the FSM for a Flip-Flop:

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

13 ASM Tool Interoperability

The combination of standards like MOF, XMI, and JMIs provides a global infrastructure for interoperability and integration of ASM tools.

The main purpose of XMI is to provide an easy interchange of data and metadata between modelling tools and metadata repositories in distributed heterogeneous environments [55]. In the ASM context, these application tools include: ASM model editors, ASM model repositories, ASM model validators, ASM model verifiers, ASM simulators, ASM-to-Any code generators, etc. Without a common metamodel for creating and accessing data, developers must hard-wire discrete interfaces between applications in order to allow model exchange and storage, thereby limiting interoperability and increasing the cost of developing and maintaining heterogeneous systems.

Fig. 35 shows a scenario of interoperability among ASM tools as suggested by our approach. According to the rules specified by the MOF-XMI mapping specification, a XML DTD has been generated from the AsmM. ASM tools (like Tool X in the figure) can exchange ASM models in the XML/XMI standard format and verify their validity with respect to the given AsmM XMI DTD. Products-specific internal representations of ASM models can remain as they are. Tool providers only need to agree on the AsmM and supply their tools with appropriate plug-ins capable of importing and/or exporting the XMI format for the AsmM (using XML-based technologies like SAX or DOM libraries or XSLT).

Some tools (like Tool Y in the figure) may keep their input data formats: in this case walkers must be developed to translate ASM models from the repository to the tool proprietary formats. From the repository, any kind of "transformation" (text-To-MOF, MOF-To-text, XMI-To-MOF, MOF-To-XMI, etc.) towards various technical spaces can be carried out.

A modeler can also start writing her/his ASM specification in the textual Asm^2L and then, trough the connection to the repository provided by the parser, transform it, for example, in the XMI interchange format.

Tools (like Tool Z in the figure) embracing the AsmM can access ASM models through the APIs (like the AsmM JMIs) in a MOF repository (like the SUN MDR [4]) where ASM models reside. XMI reader and writer provided by MDR can be used to load/save an ASM model from/into a XML file.

Currently, we have been working also to the implementation of an ASM simulator, written in Java, to make the AsmM models executable. This tool is an example of Tool Z since essentially it is an interpreter which navigates through the MOF repository where ASM models are instantiated (as instances of the AsmM metamodel) to make its computations. The main advantage of such approach is that the simulator environment including basic functionalities such as parsing, abstract syntax trees, type checking, etc., is already provided by the MOF-environment.

Mixed approaches are also possible, as the one adopted by our group in modifying the ATGT tool, as explained in Section 14.

14 The AsmM in Practice: modifying an ASM Test Generator

We have applied the AsmM in practice by modifying ATGT [12], an existing tool supporting test case generation for ASMs, originally presented in [27]. ATGT takes an ASM specification (written using the AsmGofer syntax) and produces a set of test predicates, translates the original ASM specification to Promela (the language of the SPIN model checker used to generate tests), and generates a set of test sequences by exploiting the counter example generation of the model checker.
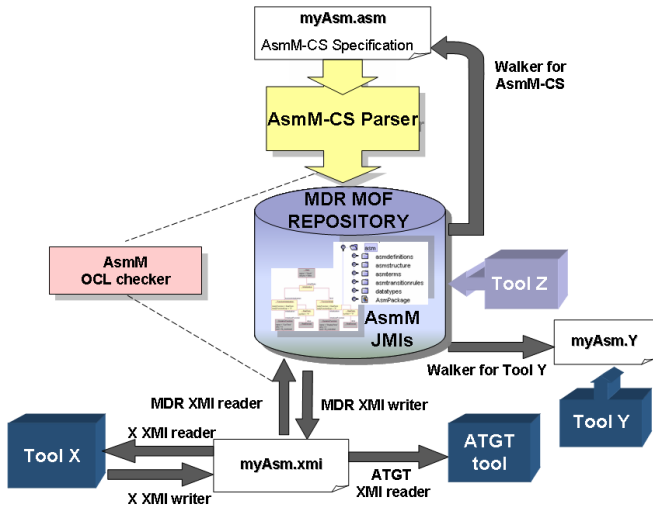
Figure 35 ASM model interchange trough XMI and APIs

We made ATGT AsmM-compliant for several reasons. First, we wanted to have a tool for test generation, able to read ASM specifications in the AsmM XMI standard format, making ATGT integrated with other tools as suggested in Sect. 13. Moreover, we wanted to test the metamodelling approach in practice, to show that it is applicable even to existing tools, since it requires a modest effort in modifying existing code. Our experience can provide guidance to people willing to write new tools or modify existing ones in order to support metamodelling approaches. Indeed, the approach presented in this section may be applied to any existing tool supporting any formal method for which there exists a MOF metamodel.

ATGT is written in Java. It already (see Fig. 36) has its own parser for AsmGofer files, which reads a specification and builds an internal representation of the model in terms of Java objects. For example, ASM rules are represented in ATGT as instances of the class `Rule`, with subclasses `Skip` and `IfThenElse` which correspond to the `SkipRule` and the `ConditionalRule` classes in the AsmM. The tool functionalities are delegated to three components (Test predicate generator, Tests generator, ASM to Promela) which read the data of the loaded ASM specification and perform their tasks.

In our approach, ATGT keeps its own data structures to represent the ASM models and other information necessary for the services it provides. In this way we do not modify the three most critical components, which continue to process data in the old representation.

To make ATGT capable of reading AsmM models, we first added a new component, the JMI/XMI reader, which is automatically derived from the metamodel by using MDR Netbeans. This JMI/XMI reader parses a XMI file containing the ASM specification the user wants to load and produces the JMI objects representing the loaded ASM. Then we added a module, called JMI queries, which queries those JMI objects and builds the equivalent model in terms of ATGT internal data. The JMI queries are

very similar to the AsmGofer parser already in ATGT, except that they read the information about the ASM model from JMI data instead of from a file.

Although we did not exploit the power of the metamodel inside ATGT and we simply made ATGT AsmM-compliant, the result is worthwhile and the effort is limited: adding this new feature to ATGT required about two man-months. If we started today from scratch to develop ATGT, we would use directly JMI to represent ASM models, since JMI offers a stable and clean interface that is derived from the metamodel. The use of JMI would avoid the burden of writing internal libraries for representing ASM models. For this reason, we have started working on making the internal representation of ASM models ATGT adopts equivalent to the JMI, in order to eventually integrate JMI directly in ATGT (work in progress in Fig. 36).

Further advances in the MDE direction [14] would be replacing the ASM to Promela and the AsmGofer parser components by model transformations from the AsmM (as pivot metamodel) to Promela metamodel and from Gofer metamodel to the AsmM, provided that such metamodels for Promela and Gofer (linked to their concrete syntax) exist.

15 Related Work

In the ASM context, no other explicit proposals exist concerning what presented in this paper. We can only cite the work in [8] as a first attempt in this respect, but unfortunately it has never been completed. This work is an attempt to realize an interchange format for ASM specifications strictly dependent on those aspects typical of functional languages, and it is technically based on the use of a pure XML format.

Concerning the definition of a concrete language for ASMs, other previous proposals exist. The Abstract State Machine Language (AsmL) [10] 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 Abstract State Machines, expression- and object- oriented, and fully integrated into the .NET framework and Microsoft development tools. 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" [58]. Adopting a terminology currently used in the MDA vision, AsmL is a platform-specific modelling language for the .NET type system. A similar consideration can be made also for the AsmGofer language [46]. An AsmGofer specification can be thought, in fact, as a PSM (platform-specific model) for the Gofer environment.

Other specific languages for the ASMs, no longer maintained, are ASM-SL [17], which adopts a functional
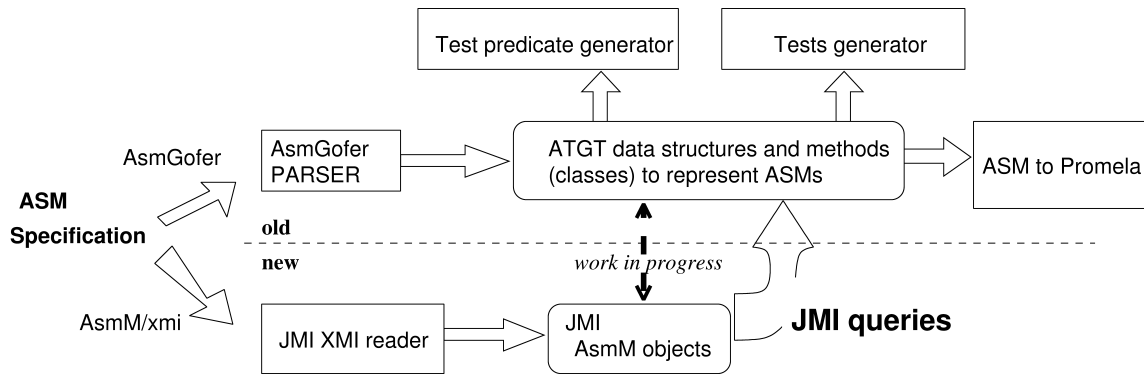
Figure 36 Adapting ATGT to the AsmM

style being developed in ML and which has inspired us in the language of terms, and XASM [9] which is integrated in Montages, an environment generally used for defining semantics and grammar of programming languages.

A platform-independent modelling language for ASMs, as the one defined by the AsmM, could allow the definition of precise *transformation bridges* in order to automatically map an ASM PIM (platform-independent model) 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 reverse engineering (or back annotation) facilities as well.

Concerning the metamodelling technique for the definition of languages, we can mention the official metamodels supported by the OMG [38] for MOF itself [2], for UML [50], for OCL [37], and for CWM [20]. Academic communities like the Graph Transformation community [29,49,51,47,52] and the Petri Net community [40,23], have also started to settle their tools on general metamodels and XML-based formats.

Recently, a metamodel for the ITU language SDL-2000 [48] was developed [25]. The authors presents also 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. A very similar method to bridge *grammarware* and *modelware* is also proposed by other authors in [7] and in [53]. These approaches are complementary to the derivation process presented in Sect. 12.3. 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 [30,21]. 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.

16 Conclusions and Future Directions

Taking advantage of the metamodel-based approach of the MDE, we propose the AsmM, a metamodel for Abstract State Machines. The AsmM delivers a standard graphical view of the ASM modelling primitives, useful specially for those people who do not deal well with "mathematics", but are familiar with the standards UML and MOF. This abstract syntax can easily match up to all the existing ASM tool's languages (or so called "concrete syntaxes") providing tool interoperability through the XMI format derived from the AsmM. From the AsmM we derived an EBNF grammar for a textual notation or concrete syntax for ASMs, called Asm^2L (AsmM Language), and developed a parser which processes Asm^2L specifications, checks for their consistency with the metamodel, and translates them into XMI format. We have applied this framework, which includes several tools and libraries, like MDR, JMI and others, to an existing ASM tool to show the viability of the proposed approach.

In future, we want to provide the Asm^2L parser introduced in Sect. 13 with a proper modelling environment which acts as front-end for the modeler. We also plan to provide the AsmM with some animation capabilities (e.g. simulation). Moreover, we intend to upgrade the AsmM to MOF 2.0 and we are evaluating the possibility to exploit other metamodelling frameworks to better support *model transformations* such as the ATL project [3], the Xactium XMF Mosaic [6], to name a few, and *model evolution activities* [24] such as code generation, reverse engineering, model refinement, model refactoring, model inconsistency management, etc. Today, only limited support is available in Model-driven development tools for these activities, but a lot of research is being carried out in this particular field to establish synergies between model-driven approaches like MDE and many other areas of software engineering including software reverse and re-engineering, Generative

techniques, generic language technology, Grammar-ware, Aspect-oriented software development, to name a few.

The AsmM specification 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 believe that benefits provided by a standard-ized notation for the ASMs may contribute to increase the practical use of the ASMs formal method and provide an efficient interoperability among ASM tools for a higher quality design based on the ASM formalism. Moreover, we believe the proposed metamodel can constitute the basics for formal methods integration [39] providing bridges from the AsmM to metamodels for other formal notations in a meta-method approach.

References

1. Java Compiler Compiler. https://javacc.dev.java.net/.
2. OMG. The Meta Object Facility Specification, document formal/2002-04-03, version 1.4.
3. The ATL model transformation language. http://www.sciences.univ-nantes.fr/lina/atl/.
4. The MDR (Model Driven Repository) for NetBeans. http://mdr.netbeans.org/.
5. The Scalable Vector Graphics home page at the World Wide Web Consortium. www.w3.org/Graphics/SVG/.
6. The Xactium XMF Mosaic tool suite. http://www.modelbased.net/www.xactium.com/.
7. M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical report, Turku Centre for Computer Science, 2003.
8. M. Anlauff, G. Del Castillo, J. Huggins, J. Janneck, J. Schmid, and W. Schulte. The ASM-Interchange Format XML Document Type Definition (ASM-DTD). http://www.first.gmd.de/~ma/asmdtd.html.
9. M. Anlauff and P. Kutter. Xasm: The Open Source ASM Language. http://www.xasm.org.
10. The Abstract State Machine Language. http://research.microsoft.com/foundations/AsmL/.
11. The Abstract State Machine Metamodel website. http://asmeta.sf.net/.
12. ATGT: ASM Tests Generation Tool. http://www.dmi.unict.it/garganti/atgt/.
13. J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
14. J. Bézivin, H. Bruneliére, F. J. Jouault, and I. Kurtev. Model Engineering Support for Tool Interoperability. In *The 4th Workshop in Software Model Engineering (WiSME'05)*, Montego Bay, Jamaica, 2005.
15. E. Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *J. UCS (Journal of Universal Computer Science)*, 8(1):2–74, Jan. 2002.
16. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
17. G. D. Castillo. The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 578–581. Springer, 2001.
18. G. D. Castillo and K. Winter. Model checking support for the ASM high-level language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 331–346, 2000.
19. OMG, The Common Object Request Broker Architecture. http://www.corba.org/.
20. OMG, The Common Warehouse Metamodel. http://www.omg.org/cwm/.
21. D. Hearnden and K. Raymond and J. Steel. Anti-Yacc: MOF-to-text. In *Proc. of EDOC*, pages 200–211, 2002.
22. A. Dold. A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universitat Ulm, July 1998.
23. J. B. E. Breton. Towards an Understanding of Model Executability. In *FOIS*, 2001.
24. T. M. et all. Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005.
25. J. Fischer, M. Piefel, and M. Scheidgen. A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In *Fourth SDL And MSC Workshop (SAM'04)*, pages 208–223, 2004.
26. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
27. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In E. Böger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer, 2003.
28. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
29. R. Holt, A. Schürr, S. E. Sim, and A. Winter. Graph eXchange Language. http://www.gupro.de/GXL/index.html.
30. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. http://www.uml.org/.
31. Java Community Process. http://java.sun.com/aboutJava/communityprocess/.
32. Java Metadata Interface Specification, Version 1.0. http://java.sun.com/products/jmi/, 2002.
33. S. Kent. Model driven engineering. In *IFM '02: Proc. of the Third International Conference on Integrated Formal Methods*, pages 286–298. Springer-Verlag, 2002.
34. I. Kurtev, J. Bézivin, and M. Aksit. Technical Spaces: An Initial Appraisal. In *CoopIS, DOA'2002, Federated Conferences, Industrial track*, Irvine, 2002.
35. OMG. The Model Driven Architecture (MDA). http://www.omg.org/mda/.
36. J. P. Nytun, A. Prinz, and M. S. Tveit. Automatic generation of modelling tools. In *Proc. of ECMDA-FA*, pages 268–283, 2006.
37. OMG. UML 2.0 OCL Specification, ptc/03-10-14.
38. The Object Managment Group (OMG). http://www.omg.org.
39. R. F. Paige. A meta-method for formal method integration. In *In Proc. of Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, volume 1313 of *LNCS*, pages 473–494. Springer, 1997.
40. Petri Net Markup Laguage (PNML). http://www.informatik.hu-berlin.de/top/pnml.
41. Poseidon UML Tool. http://www.gentleware.com.
42. OMG, Request For Proposal: MOF 2.0/QVT, ad/2002-04-10. http:www.omg.org.

43. 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.

44. P. Scandurra, A. Gargantini, C. Genovese, T. Genovese, and E. Riccobene. A Concrete Syntax derived from the Abstract State Machine Metamodel. In *12th International Workshop on Abstract State Machines (ASM'05)*, 8-11 March 2005, Paris, France, 2005.

45. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 3(4):377–413, 1997.

46. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.

47. A. Schürr, A. J. Winter, and A. Zündorf. *Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

48. SDL (Specification and Description Language. ITU Reccomandation Z.100. `http://www.itu.int`.

49. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *J. Padberg (Ed.), UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS*, 2001.

50. OMG. The Unified Modeling Language (UML). `http://www.uml.org`.

51. D. Varró, G. Varró, and A. Pataricza. Towards an XMI–based model interchange format for graph transformation systems. Technical report, Budapest University of Technology and Economics, Dept. of Measurement and Information Systems, September 2000.

52. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

53. M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Proc. of the 4th Workshop in Software Model Engineering (WiSME'05)*, Montego Bay, Jamaica, 2005.

54. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 3(5):689–701, 1997.

55. OMG, XMI Specification, v1.2. `http://www.omg.org/cgi-bin/doc?formal/2002-01-01`.

56. OMG. XMI Specification, v2.0. `http://www.omg.org/cgi-bin/doc?formal/2003-05-02`.

57. W3C, The Extensible Markup Language (XML). `http://http://www.w3.org/xml/`.

58. Y. Gurevich and B. Rossman and W. Schulte. Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .

Appendix A – the Asm^2L language

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 ::= ( <ASYNCR> )? ( <ASM> | <MODULE> ) ID Header Body ( <MAIN> MacroDeclaration )? ( ( Initialization )*
<DEFAULT> Initialization ( Initialization )* )? <EOF>
```

```
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
)*
```

```
DomainInitialization ::= <DOMAIN> ID_DOMAIN "=" Term
```

```
FunctionInitialization ::= <FUNCTION> ID_FUNCTION ( "(" VariableTerm IN getDomainByID ( "," VariableTerm IN
getDomainByID )* ")" )? "=" Term
```

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

```
Body ::= <DEFINITIONS> ":" ( DomainDefinition )* (FunctionDefinition )* ( RuleDeclaration )* ( Axiom )*
```

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

```
FunctionDefinition ::= <FUNCTION> ID_FUNCTION ( "(" VariableTerm IN getDomainByID ( "," VariableTerm IN
getDomainByID )* ")" )? "="
```

```
Term RuleDeclaration ::= ( MacroDeclaration | TurboDeclaration )
```

```
MacroDeclaration ::= ( <MACRO> )? <RULE> ID_RULE ( "(" VariableTerm IN getDomainByID ( "," VariableTerm IN
getDomainByID )* ")" )? "=" Rule
```

```
TurboDeclaration ::= <TURBO> <RULE> ID_RULE ( "(" VariableTerm IN getDomainByID ( "," VariableTerm IN
getDomainByID )* ")" )? ( IN getDomainByID )? "=" Rule
```

```
Axiom ::= <AXIOM> ( ID )? <OVER> ( ID_DOMAIN | ID_FUNCTION ( "(" (getDomainByID )? ")" )? | ID_RULE ) ( "," (
ID_DOMAIN |ID_FUNCTION ( "(" ( getDomainByID )? ")" )? | ID_RULE ) )* ":" Term
```


The definitional language

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

```
ConcreteDomain ::= ( <DYNAMIC> )? <DOMAIN> ID_DOMAIN <SUBSETOF> getDomainByID
```

```
TypeDomain ::= ( AnyDomain | StructuredTD | EnumTD | AbstractTD | BasicTD )
```

```
AnyDomain ::= <ANYDOMAIN> ID_DOMAIN
```

```
BasicTD ::= <BASIC> <DOMAIN> ID_DOMAIN
```

```
AbstractTD ::= ( <DYNAMIC> )? <ABSTRACT> <DOMAIN> ID_DOMAIN

EnumTD ::= <ENUM> <DOMAIN> ID_DOMAIN "=" "{" EnumElement ( "|" EnumElement )* "}"

EnumElement ::= ID_ENUM

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

ProductDomain ::= <PROD> "(" getDomainByID ( "," getDomainByID )+ ")"

SequenceDomain ::= <SEQ> "(" getDomainByID ")"

PowersetDomain ::= <POWERSET> "(" getDomainByID ")"

BagDomain ::= <BAG> "(" getDomainByID ")"

MapDomain ::= <MAP> "(" getDomainByID "," getDomainByID ")"

getDomainByID ::= ( ID_DOMAIN | StructuredTD )

Function ::= ( BasicFunction | DerivedFunction )

BasicFunction ::= ( StaticFunction | DynamicFunction )

DerivedFunction ::= <DERIVED> ID_FUNCTION ":" ( getDomainByID "->")? getDomainByID

StaticFunction ::= <STATIC> ID_FUNCTION ":" ( getDomainByID "->")? getDomainByID

DynamicFunction ::= ( OutFunction | MonitoredFunction | SharedFunction | ControlledFunction | LocalFunction )

ControlledFunction ::= ( <DYNAMIC> )? <CONTROLLED> ID_FUNCTION ":" ( getDomainByID "->" )? getDomainByID

SharedFunction ::= ( <DYNAMIC> )? <SHARED> ID_FUNCTION ":" (getDomainByID "->" )? getDomainByID

MonitoredFunction ::= ( <DYNAMIC> )? <MONITORED> ID_FUNCTION ":" (getDomainByID "->" )? getDomainByID

OutFunction ::= ( <DYNAMIC> )? <OUT> ID_FUNCTION ":" (getDomainByID "->" )? getDomainByID

LocalFunction ::= ( <DYNAMIC> )? <LOCAL> ID_FUNCTION ":" (getDomainByID "->" )? getDomainByID
```

The language of terms

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

Expression ::= or_xorLogicExpr ( ( <ID_FUNCTION> or_xorLogicExpr | <ID_FUNCTION> or_xorLogicExpr ) )*

or_xorLogicExpr ::= andLogicExpr ( ( <ID_FUNCTION> | <ID_FUNCTION> ) andLogicExpr )*

andLogicExpr ::= notLogicExpr ( <ID_FUNCTION> notLogicExpr )*

notLogicExpr ::= ( <ID_FUNCTION> includesExpr | includesExpr )

includesExpr ::= relationalExpr ( ( <ID_FUNCTION> relationalExpr | <ID_FUNCTION> relationalExpr ) )?

relationalExpr ::= additiveExpr ( ( <EQ> additiveExpr | <NEQ> additiveExpr | <LT> additiveExpr | <LE>
additiveExpr | <GT> additiveExpr | <GE> additiveExpr ) )*

additiveExpr ::= multiplicativeExpr ( ( <PLUS> multiplicativeExpr | <MINUS> multiplicativeExpr ) )*

multiplicativeExpr ::= powerExpr ( ( <MULT> powerExpr | <DIV> powerExpr | <ID_FUNCTION> powerExpr ) )*
```

```
powerExpr ::= unaryExpr ( <PWR> unaryExpr )*

unaryExpr ::= ( ( <PLUS> unaryExpr | <MINUS> unaryExpr ) | 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 ::= <NATNUMBER>

CharTerm ::= <CHAR_LITERAL>

StringTerm ::= <STRING_LITERAL>

BooleanTerm ::= ( <TRUE> | <FALSE> )

UndefTerm ::= <UNDEF>

EnumTerm ::= ID_ENUM

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

ConditionalTerm ::= <IF> Term <THEN> Term ( <ELSE> Term )? <ENDIF>

CaseTerm ::= <SWITCH> Term ( <CASE> Term ":" Term )+ ( <OTHERWISE>
Term )? <END_SWITCH>

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

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

SequenceTerm ::= "[" ( Term ( ( "," Term )+ | ( ".." Term ( "," (Term ) )? ) )? )? "]"

SetTerm ::= "{" ( Term ( ( "," Term )+ | ( ".." Term ( "," ( Term
) )? ) )? )? "}"

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

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

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 ::= getDomainByID

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

The language of rules

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

TermAsRule ::= ( FunctionTerm | VariableTerm )

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

SkipRule ::= <Skip>

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

BlockRule ::= <PAR> 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 | TurboCallRule | TurboLocalStateRule )

SeqRule ::= <seq> Rule ( Rule )+ <ENDSEQ>

IterateRule ::= <ITERATE> Rule <ENDITERATE>

TurboCallRule ::= ID_RULE "(" ( Term ( "," Term )* )? ")"
```

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

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

TryCatchRule ::= <TRY> Rule <CATCH> ( Term ) ( "," ( Term ) )* Rule

DerivedRule ::= ( BasicDerivedRule | TurboDerivedRule )

BasicDerivedRule ::= CaseRule

CaseRule ::= <SWITCH> Term ( <CASE> Term ":" Rule )+ ( <OTHERWISE> Rule )? <END_SWITCH>

TurboDerivedRule ::= ( RecursiveWhileRule | IterativeWhileRule )

RecursiveWhileRule ::= <WHILEREC> Term <DO> Rule

IterativeWhileRule ::= <WHILE> Term <DO> Rule
```

Final terminals

```
ID_VARIABLE ::= <ID_VARIABLE>

ID_ENUM ::= <ID_ENUM>

ID_DOMAIN ::= <ID_DOMAIN>

ID_RULE ::= <ID_RULE>

ID_FUNCTION ::= <ID_FUNCTION>

ID_AGENT ::= <ID_FUNCTION>

ID ::= ( <ID> )

MOD_ID ::= ( <MOD_ID> )

IN ::= <IN>

<ASM> ::="asm"

<MODULE> ::="module"

<ASYNCR> ::="asyncr"

<IMPORT> ::="import"

<EXPORT> ::="export"

<SIGNATURE> ::="signature"

<INIT> ::="init"

<DEFAULT> ::="default"

<AGENT> ::="agent"

<AXIOM> ::="axiom"

<OVER>  ::="over"

<DEFINITIONS> ::= "definitions"

<FUNCTION> ::="function"

<STATIC> ::= "static"

<DYNAMIC> ::= "dynamic"

<DERIVED> ::= "derived"

<MONITORED> ::= "monitored"

<CONTROLLED> ::=  "controlled"

<SHARED> ::= "shared"

<OUT> ::="out"

<DOMAIN> ::= "domain"
```

```
<ANYDOMAIN> ::= "anydomain"

<BASIC> ::= "basic"

<ABSTRACT> ::= "abstract"

<ENUM> ::= "enum"

<SUBSETOF> ::="subsetof"

<PROD> ::= "Prod"

<SEQ> ::= "Seq"

<POWERSET> ::= "Powerset"

<BAG> ::= "Bag"

<MAP> ::= "Map"

<TRUE> ::= "true"

<FALSE> ::= "false"

<UNDEF> ::= "undef"

<IF> ::= "if"

<THEN> "then"

<ELSE> ::= "else"

<ENDIF> ::="endif"

<SWITCH> ::= "switch"

<END_SWITCH> ::= "endswitch"

<CASE>  ::= "case"

<OTHERWISE> ::= "otherwise"

<ENDCASE> ::= "endcase"

<LET> ::="let"

<ENDLET> ::="endlet"

<EXIST> ::="exist"

<UNIQUE> ::="unique"

<WITH> ::= "with"

<FORALL> ::= "forall"

<Skip> ::= "skip"

<RULE> ::= "rule"

<MACRO>  ::= "macro"
```

```
<TURBO> ::= "turbo"

<MAIN>  ::= "main"

<PAR> ::= "par"

<ENDPAR> ::= "endpar"

<CHOOSE> ::= "choose"

<DO>  ::=  "do"

<IFNONE> ::="ifnone"

<EXTEND> ::= "extend"

<seq> ::= "seq"

<ENDSEQ> ::="endseq"

<ITERATE> ::="iterate"

<ENDITERATE> ::="enditerate"

<LOCAL> ::= "local"

<TRY> ::= "try"

<CATCH> ::= "catch"

<WHILE> ::= "while"

<WHILEREC> ::= "whilerec"

<IN> ::= "in"

<EQ>  ::= "="

<LT> ::= "<"

<LE> ::= "<="

<GT> ::= ">"

<GE> ::= ">="

<NEQ ::= "!="

<PLUS> ::= "+"

<MINUS> ::= "-"

<MULT> ::= "*"

<DIV> ::= "/"

<PWR> ::= "^"

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

<NATNUMBER> ::= (<DIGIT>)+"n"
```

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

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

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

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

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

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

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

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

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

<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"])
                      ) )* "\"" >

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

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