Journal of Universal Computer Science, vol. 14, no. 12 (2008), 1949-1983
submitted: 1/9/07, accepted: 1/6/08, appeared: 28/6/08 © *J.UCS*

A Metamodel-based Language and a Simulation Engine for Abstract State Machines

Angelo Gargantini
(Dip. di Ing. Informatica e Metodi Matematici - Università di Bergamo, Italy
angelo.gargantini@unibg.it)

Elvinia Riccobene and **Patrizia Scandurra**
(Dip. di Tecnologie dell'Informazione - Università di Milano, Italy
{riccobene, scandurra}@dti.unimi.it)

Abstract : In this paper, we present a concrete textual notation, called *AsmetaL*, and a general-purpose simulation engine, called *AsmetaS*, for Abstract State Machine (ASM) specifications. They have been developed as part of the ASMETA (ASMs mETAmodelling) toolset, which is a set of tools for ASMs based on the metamodelling approach of the Model-driven Engineering. We briefly present the ASMETA framework, and we discuss how the language and the simulator have been developed exploiting the advantages offered by the metamodelling approach. We introduce the language AsmetaL used to write ASM specifications, and we provide the AsmetaL encoding of ASM specifications of increasing complexity. We explain the AsmetaS architecture, its kernel engine, and how the simulator works within the ASMETA tool set. We discuss the features currently supported by the simulator and how it has been validated.

Key Words: Abstract State Machines, ASM language, ASM simulator, Model-driven Engineering, Metamodelling.
Category: D.2.1, D.2.6, F.3.1, B.2

1 Introduction

Abstract State Machines (ASMs) [Börger and Stärk, 2003] are nowadays known as a formal method successfully employed as system engineering method that guides the development of complex systems seamlessly from requirements capture to their implementation. The increasing application of the ASM formal method in 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 [Winter, 1997, Schellhorn and Ahrendt, 1997, Gargantini and Riccobene, 2000], and execution engines for simulation and testing purposes [ASML, 2001, Gargantini et al., 2003, Schmid, 2001, Ouimet and Lundqvist, 2007, Castillo, 2001, CoreAsm, 2008].

Since each tool usually covers well only one aspect of the whole system development process, at different steps developers would like to switch tools while reusing information already entered about their models. However, each tool introduces a different syntax strictly depending on the implementation environment, adopts its own internal representation of ASM models, and provides proprietary constructs which extend the basic ASM mathematical concepts. ASM tools are therefore loosely coupled and their interoperability is hard to accomplish, so preventing ASMs from being used in an efficient and tool supported manner during the system development life-cycle. Furthermore, there is no agreement around a common standard and open ASM language, and most ASM researchers still use their own ASM notation, normally not defined by a grammar but in terms of mathematical concepts. Moreover, due to the lack of abstractness of the tool languages, the process of encoding ASM models is also not always straightforward and natural, and one needs to map mathematical concepts into types and structures provided by the target language.

To achieve the goals of developing a *unified abstract notation* for ASMs, a notation independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs, and developing a *general framework for a wide interoperability and integration of tools* around ASMs, we exploited the *metamodelling* approach suggested by the Model-Driven Engineering (MDE) [Bézivin, 2004, Kent, 2002, Nytun et al., 2006].

Initially [Riccobene and Scandurra, 2004], we aimed only at defining a standard interchange format for a systematic integration of a number of loosely-coupled ASM tools by taking advantage of the metamodelling approach, which provides standard XML-based interchange formats – like the Object Management Group [OMG, 2008] standard XML Metadata Interchange (XMI) – for metamodel-based languages. We therefore defined the *Abstract State Machine Metamodel* (*AsmM*) as abstract syntax description of a language for ASMs, using the OMG metamodelling framework. From the AsmM metamodel, we obtained in a generative manner (i.e. automatically) several artifacts (an interchange format, APIs, etc..) for the creation, storage, interchange, access and manipulation of ASM models. The AsmM and the combination of these language artifacts lead to an *instantiation* of the OMG metamodelling framework for the ASM application domain, the *ASM mETAmodelling framework* (*ASMETA*) that provides a global infrastructure for the interoperability of ASM tools (new and existing ones). Then, we realized that a developer who is interested in developing a new tool for ASMs can completely base, with minimal effort, the tool development on the ASMETA framework and exploit all technologies provided by ASMETA in terms of specification language, abstract storage (i.e. the model repository), standard model access and manipulation APIs, interchange format, etc.

In this paper, we present a concrete textual notation, called *AsmetaL*, and a general-purpose simulation engine, called *AsmetaS*, for ASM specifications. Both are based on the ASMETA framework. This paper is an extended version of the work presented in [Gargantini et al., 2007a]. We here emphasize the description of the AsmetaL language, and, through examples of ASM specifications of increasing complexity, we show the leanness of the language and the simplicity of encoding mathematical ASM models in AsmetaL. We then introduce the architecture of AsmetaS, which has been developed as a simulator of AsmetaL programs. We describe the execution engine of the simulator, how AsmetaS works within the ASMETA tool set, and how the simulator has been validated. With respect to the previous work in [Gargantini et al., 2007a], we here discuss in much more detail the features supported by the simulator, how they allow a simulator customization, how they can be exploited to speed-up the model execution, and, by concrete examples, we also discuss how they can be used for model validation. For validation purposes, we also consider the clustering algorithm as described in terms of ASMs in [Jensen et al., 2007]. This algorithm was taken by the authors in [Jensen et al., 2007] as case study to compare different ASM specifications in AsmL, CoreASM and Java.

The paper is organized as follows. Sect. 2 presents the overall development process of the ASMETA framework and the ASMETA tool set. Sect. 3 provides a brief description of the AsmM metamodel. Sect. 4 presents the AsmetaL language and reports some examples of ASM models in AsmetaL. Sect. 5 describes the simulator and its features. Related and future work are given in sections 6 and 7, respectively.

2 The ASMETA framework

ASMETA is an *instantiation* of the OMG metamodelling framework for the ASM application domain. It has been obtained by applying to the AsmM metamodel standard or proprietary bridges (or projections) towards other *technical spaces*[Kurtev et al., 2002], namely working contexts with a set of associated concepts, knowledge, tools, skills, and possibilities.

In the context of (software) language engineering, metamodelling is to be intended as a way to define the abstract syntax of a language or formalism in terms of a (usually object-oriented) model, called metamodel. A metamodel-based abstract syntax definition has the great advantage of being suitable to derive from the same metamodel in a generative manner (through standard 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. Therefore, a metamodel could serve as *standard interlingua* for a specific domain of interest and it allows settling a flexible object-oriented infrastructure on which tools development and their interoperability should be based.

Furthermore, metamodelling allows settling a “global framework” to enable otherwise dissimilar languages of possibly different domains to be used and coordinated in an interoperable manner in various technical spaces. Indeed, it allows the establishment of precise bridges among the metamodels of these different domain-specific languages to automatically execute model transformations.

In the remainder of this section, we explain the process of developing the ASMETA framework and then we briefly present a set of tools – the ASMETA toolset – around ASMs developed and/or integrated by exploiting the ASMETA facilities (in terms of derivatives, libraries, interchange format, APIs, etc.).

2.1 The ASMETA development process

The overall process of developing a tool set around the ASM formal method exploiting the metamodelling approach consisted of the following steps:

1. Choice of a metamodelling framework and supporting technologies;
2. Definition of the metamodel, AsmM, for ASM concepts;
3. Metamodel derivatives development as additional facilities to handle ASM models:
   (a) an XMI-based interchange syntax for serializing ASM models;
   (b) APIs (like the Sun Java Metadata Interface (JMI) technology [JMI, 2002]) to access and manipulate ASM models in a model repository;
   (c) one or more ASM concrete syntaxes (textual, graphical, or mixed) for human-use with their associated parsers for the conformance-checking of ASM models against the AsmM metamodel;
4. Development of tools based on the chosen metamodelling framework and of software artifacts to integrate existing ASM tools with the metamodelling framework;
5. Validation of the AsmM metamodel and its derivatives.

Step 1. – We first chose the OMG metamodelling platform even though many other implementations of the MDE principles exist, like the AMMA metamodelling platform [AMMA, 2005], the Xactium XMF Mosaic [XMF Mosaic, 2007] initiative, the Microsoft Software Factories [Microsoft DSL Tools, 2005], the Model integrated Computing (MIC) [Sztipanovits and Karsai, 1997] and the Generic Modeling Environment [GME, 2006] for domain-specific modelling, etc. We use the OMG's Meta Object Facility (MOF) as meta-language, i.e. as language to define metamodels. In particular, we adopt the MOF 1.4 MDR (Model Driven Repository) of NetBeans [MDR, 2003] as model repository, the Poseidon UML tool as metamodel editor, the XMI 1.2 format and JMIs as generated by the MDR framework, and the OCL support provided by the OCLE [OCLE, 2005][1] to define and validate the OCL constraints defined over the metamodel.

[1] Note that we had to use two different tools (one for the metamodel and one for the OCL constraints) because current UML tools present several limitations regarding the OCL support [Cabot and Teniente, 2006].

It should be noted that thanks to well-mastered model transformations, like the ATL-KM3 plugin [Jouault et al., 2006] which allows to move forward and backward in the EMF and MOF modelling spaces, the choice of a specific metamodelling framework does not prevent the use of models in other different modelling spaces.

Step 2. and **step 3.** lead to the instantiation of the OMG metamodelling framework for the ASM domain, ASMETA. We started by defining the AsmM metamodel (see Sect. 3), and then we automatically derived from the metamodel some additional facilities to handle models: an XMI (XML Metadata Interchange) [OMG, 2008] interchange format for ASM models; JMI (Java Metadata Interfaces) APIs for the creation, storage, access and manipulation of ASM models in a MOF-based model repository; a concrete textual notation, called AsmetaL (ASMETA Language) (see Sect. 4), and its parser to effectively edit ASM models conforming to the AsmM metamodel.

Details on the development of the AsmM metamodel and its derivatives using the OMG framework can be found in [Gargantini et al., 2006b, AsmM, 2006]. AsmM and ASMETA are also available in the meta-environments AMMA/KM3 [Jouault and Bézivin, 2006] and in EMF/Ecore [EMF, 2008]. This allows us to look at the ASMETA framework as a *family of metamodels*.

Step 4. resulted in the ASMETA toolset (see Sect. 2.2) and software artifacts to integrate external and existing tools [Gargantini et al., 2007b].

Step 5. – The validation process applied both to the metamodel and to its derivatives. Since the metamodel represents the abstract notation of a specification language, one may validate the metamodel by validating the expressive power of languages derived from the metamodel. We validated the AsmM metamodel and the AsmetaL notation to asses their usability and capability to encode ASM models, namely to test if AsmetaL is suitable to encode non trivial ASM specifications and if the encoding process of mathematical models is natural and straightforward. To this purpose, we have asked a non ASM expert for porting some specifications from [Börger and Stärk, 2003] and other ASM case studies to AsmetaL. The task was completed within three man-months, by encoding almost all ASM specifications provided in [Börger and Stärk, 2003]. The encoding of ASM models was really straightforward, since the AsmM metamodel and, therefore, the AsmetaL language, have been designed to represent ASM concepts and constructs as described in [Börger and Stärk, 2003] balancing expressiveness and simplicity. Up to now we have about 150 AsmetaL specifications (including those for testing purposes) available at the ASMETA website [AsmM, 2006].

The validation of the metamodel derivatives consisted of the evaluation of their capability to provide the desired global infrastructure for the development of new tools, the integration of existing tools, and the tool interoperability in general. The development of the simulator was a case study toward the validation of the metamodel derivatives. In [Gargantini et al., 2006b] details can be found on how the ASM tools interoperability is achieved by the ASMETA.

The overall process turned out to be iterative: often we had to come back to previous steps and make corrections.

2.2 The ASMETA tool set

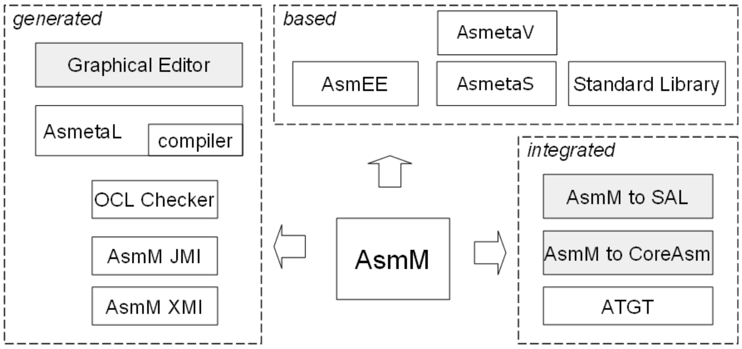

Figure 1: The ASMETA tool set

Fig. 1 shows our plan of the ASMETA tool set which: (a) provides an intuitive modelling notation having rigourous syntax and semantics, and provides a graphical view of the model; (b) allows modelling techniques which facilitate the use of the ASMs in many stages of the development process and which integrates dynamic (operational) and static (declarative) descriptions; (c) enables analysis techniques that combine validation (by simulation and testing) and verification (by model checking or theorem proving) methods at any desired level of detail; and (d) supports an open and flexible architecture to make easier the development of new tools and the integration with other existing tools.

The ASMETA tool set consists of the following components (those visualized in gray are still under development).

– The **AsmM metamodel**, the *abstract syntax*, is a complete meta-level representation of ASMs concepts based on the OMG's MOF 1.4 [MOF, 2002] as supported by the MDR environment. Currently, AsmM is also publicly available – see [AsmM, 2006] – as expressed in the meta-languages AMMA/KM3 [Jouault and Bézivin, 2006] and in EMF/Ecore [EMF, 2008].Further technical details on AsmM are provided in Sect. 3.

– The **AsmM OCL checker**, used to check if a given model is well-formed or not with respect to the OCL constraints defined over the AsmM metamodel.

– The **AsmM Java Metadata Interfaces** (JMIs) to manage the creation, storage, access, discovery, and exchange of ASM models (either at design time or at runtime) in terms of Java objects. They have been generated automatically from the AsmM metamodel by using the NetBeans MDR framework [MDR, 2003].

– The **AsmM XMI format** which is XMI 1.2 compliant and is provided in terms of an XML Document Type Definition (DTD) automatically generated from the AsmM by using MDR, for the interchange of ASM models among tools by XML serialization.

– The **AsmetaL** (ASMETA Language) textual notation for the AsmM, provided in terms of an EBNF (extended Backus-Naur Form) grammar generated from the AsmM (the abstract syntax) as a *concrete syntax* to be used by modelers to effectively write ASM models in a textual form.

– A text-to-model compiler, **AsmetaLc**, to parse ASM models written in the AsmetaL notation, check for their consistency with respect to the OCL constraints of the metamodel, and translate information about concrete models into AsmM instances in a MOF-based repository by using the AsmM JMIs.

– A **standard library**, namely a declarative collection of predefined ASM domains (*basic domains* for primitive data values like Boolean, Natural, Integer, Real, etc., and *structured domains* over other domains like finite sets, sequences, bags, maps and cartesian products) and functions implementing a set of canonical operations on domains.

– A **graphical notation**, generated from the AsmM as an alternative *concrete syntax* to be used by modelers to effectively write ASM models in a graphical form. We have been investigating for this scope, the Eclipse Graphical Modeling Framework (GMF) technology [GMF, 2008], which provides a generative component and runtime infrastructure for developing graphical editors based on EMF/GEF [EMF, 2008], and the MIC/GME [GME, 2006] which offers similar features. Both frameworks follow a novel approach, which suggests to derive modelling tools, like graphical model editors, from metamodels.

– The **AsmetaS** (ASMETA Simulator) simulator to make AsmM models executable; essentially, it is an interpreter which navigates through a model repository where ASM specifications are stored (as instances of the AsmM metamodel) to make its computations.

– The **ATGT** (ASM Tests Generation Tool) [ATGT, 2008], an existing tool for test case generation from models, which has been made AsmM-compliant.

– The **AsmetaV** (ASMETA validator) for *scenario-based validation* of ASM models. It is based on the AsmetaS simulator and on the **Avalla** (ASMETA Validation Language) language. This last is another metamodel-based language which provides constructs to express execution scenarios in an algorithmic way as interaction sequences consisting of actions committed by the design actor.

– The **AsmM-to-CoreAsm** and **AsmM-to-SAL** components which export ASMETA models to the CoreASM [CoreAsm, 2008] simulator and the SAL verifier [SAL, 2008].

– A graphical front-end called **ASMEE** (ASM Eclipse Environment) which acts as IDE to edit, manipulate, and export ASM models by using all tools/artifacts listed above. This environment is implemented as an Eclipse plug-in.

All the above artifacts/tools are classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying appropriate MOF projections to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the ASMETA environment and related derivatives; an example of such a tool is the simulator (see Sect. 5). Integrated artifacts/tools are external and existing tools that are connected to the ASMETA environment.

All available material on the ASMETA tool set (including source code, binaries, documentation and a great variety of ASM specifications) can be found in [AsmM, 2006], under GNU General Public License (GPL).

3 The AsmM metamodel

The AsmM metamodel was developed in a modular and bottom-up way. We started separating the ASM static part represented by the *state*, namely domains, functions and terms, from the dynamic part represented by the *transition system*, namely the ASM rules. Then, we proceeded to model Basic ASMs, Turbo ASMs, and multi-Agent ASMs, so reflecting the natural classification of ASMs.

Metamodelling representation results into class diagrams developed using the MOF modelling constructs (classes, associations, generalization relationships, packages, etc.). Each class is also equipped with a set of relevant *constraints*, OCL invariants written to fix how to meaningful connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams. The complete AsmM metamodel is organized in one package called `ASMETA` containing 115 classes, 114 associations, and 150 OCL class invariants, approximatively. The `ASMETA` package is further divided into four packages as shown in Fig. 2. Each package covers different aspects of the ASMs. The dashed gray ovals in Fig. 2 denote packages representing the notions of *State* and *Transition System*, respectively. The `Structure` package defines architectural constructs (modules and machines) required to specify the backbone of an ASM model. The `Definitions` package contains all basic constructs (functions, domains, constraints, rule declarations, etc..) which characterize algebraic specifications. The `Terms` package provides all kinds of syntactic expressions which can be evaluated in a state of an ASM. The `TransitionRules` package contains all possible transition rules schemes of Basic and Turbo ASMs. All derived transition rules are contained in the `DerivedTransitionRules` package. All relations between packages are of type *uses*.

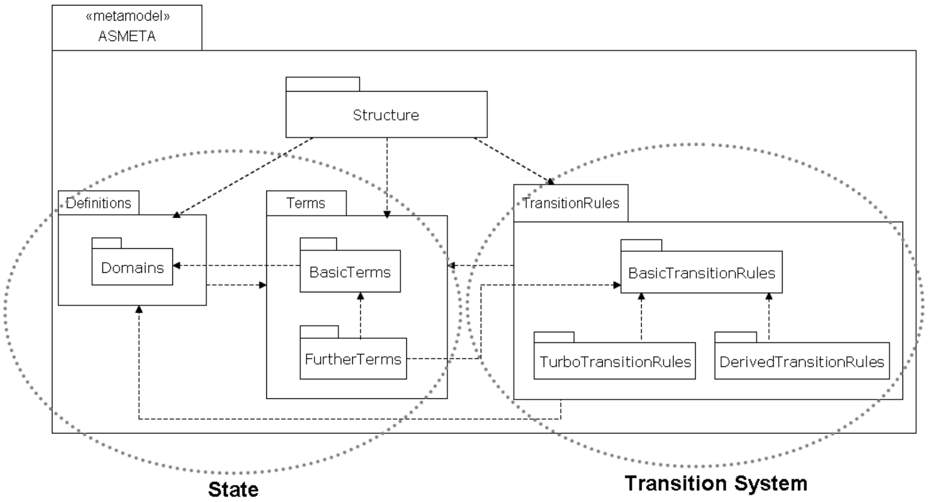

Figure 2: Package structure of the AsmM metamodel

We present here only a very small fragment of the AsmM whose complete description can be found in [Gargantini et al., 2006b, AsmM, 2006].

Fig. 3 shows the backbone of an *ASM*. An instance of the root class `Asm` represents an entire ASM specification. According to the *working definition* of an ASM model given in [Börger and Stärk, 2003], a basic ASM has a `name` and is defined by a `Header` (to establish the signature), a `Body` (to define domains, functions, and rules), a `main rule`, and a set of initial states (instances of the `Initialization` class). All possible initial states are linked to an ASM by the association end `initialState` and one initial state is elected as *default* (see the association end `defaultInitialState`).

ASM rule constructors are represented by subclasses of the class *`Rule`*. Fig. 4 shows a subset of basic forms of a transition rule under the class hierarchy rooted by the class *`BasicRule`*: update rule, conditional rule, skip, do in parallel (block rule), extend, etc.

To define the *semantics* of the metamodel, we established a semantic mapping from AsmM to a semantic domain where AsmM constructs take their meaning. The semantic domain is the first- order logic extended with a logic for function updates and for transition rule constructors formally defined in [Börger and Stärk, 2003]. The semantic mapping relating syntactic AsmM con-

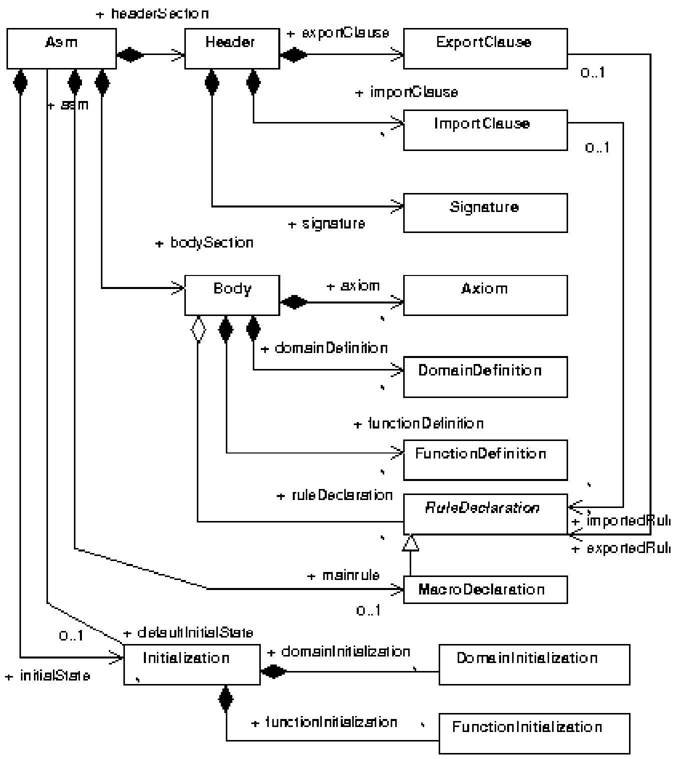

Figure 3: Backbone

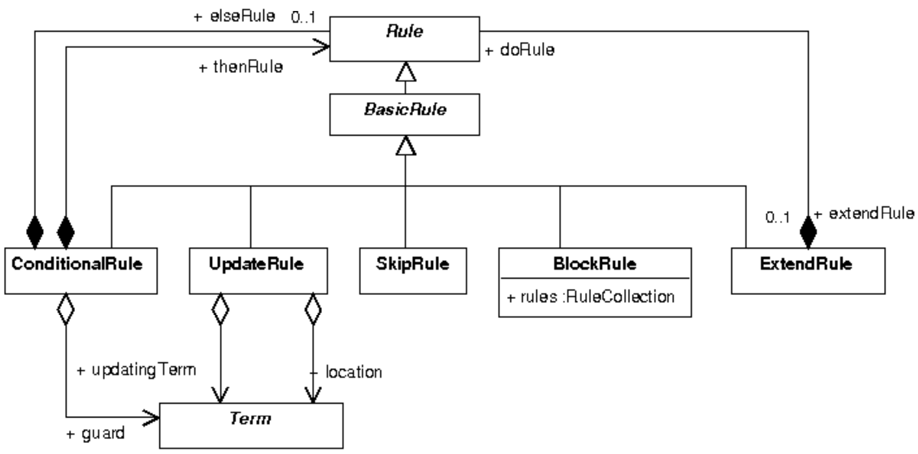

Figure 4: Basic rule forms (a subset)

cepts to those of the semantic domain has been implemented to develop the AsmetaS interpreter.

4 The AsmetaL Language

The ASMETA Language (or AsmetaL) consists of four sub-languages reflecting the packages of the ASM matamodel: the *structural language* which provides the constructs describing the structure of an ASM, the *definitional language* which provides the notation to define basic ASM elements such as functions, domains, rules, and axioms, the *language of terms* to represent syntactic expressions to be evaluated in an ASM state, and the *language of rules* which provides a notation to specify the transition rule schemes of an ASM.

AsmetaL is a metamodel-based language: it has been defined through a MOF-to-grammar mapping applied to the AsmM metamodel to derive a concrete textual syntax compliant to the metamodel. To this goal, we initially investigated the use of tools like HUTN (Human Usable Textual Notation) [HUTN, 2004] or Anti- Yacc [Hearnden et al., 2002] which are capable of generating text grammars from specific MOF-based repositories. Nevertheless, we decided not to use them since they do not permit a detailed customization of the generated language and they provide concrete notations merely suitable for object-oriented languages. We defined [Gargantini et al., 2006a], instead, general rules on how to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we use these mapping rules to derive an EBNF grammar from the AsmM. The AsmetaL textual notation is the resulting language. It is completely independent from any specific platform and allows a natural and straightforward encoding of ASM models according to the AsmM metamodel (the abstract syntax). There are better MOF-to-grammar tools now, like xText [Efftinge, 2006] of OpenArchitectureWare or TCS of AMMA [AMMA, 2005], which we may consider to adopt in the future.

A detailed description of the set of mapping rules from MOF to EBNF that we adopted to define the AsmetaL notation can be found in [Gargantini et al., 2006a]. Essentially, 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 to appear – 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*[2] of the class according to the other rules for mapping attributes, association ends, generalization relationships, etc.. As this mapping is general enough, the same transformation rules have been reused (opportunely adapted) for the EMF-to-BNF mapping for the ASMETA porting to the Eclipse/EMF framework.

[2] A *full descriptor* is the full description of an object (all attributes, associations, etc.) including features inherited from ancestor classes.

Due to the lack of space, we report below (see Listings 1 and 2) only some fragments[3] of the AsmetaL EBNF grammar concerning the structural sub-language (corresponding to the AsmM portion shown in Fig. 3) and the language of transition rules (see Fig. 4). The complete EBNF grammar can be found in [Gargantini et al., 2006b, AsmM, 2006] and a detailed user guide is available at [AsmM, 2006].

As mentioned above, an ASM specification encoded in AsmetaL reflect the structure of the AsmM metamodel, and therefore includes (see the AsmetaL program template [4] in Table 1) four sections: a *header*, a *body*, a *main rule* and an *initialization*. The name of the ASM is specified before the header section. Since we consider an ASM *module* as an ASM without the main rule and without a set of initial states, a module is specified like an ASM but replacing the keyword `asm` with the keyword `module`.

Listing 1: EBNF portion of the AsmetaL *structural* language

Asm ::= ((**asm**|**module**) ID Header Body (**main** MacroDeclaration)?
((Initialization)∗ **default** Initialization (Initialization)∗)?

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

Body ::= **definitions** ‘‘:’’ (DomainDefinition)∗ (FunctionDefinition)∗
(RuleDeclaration)∗ (Axiom)∗

Initialization ::= **init** <ID> ‘‘:’’ (DomainInitialization)∗ (FunctionInitialization)∗

Signature ::= **signature** ‘‘:’’ (Domain)∗ (Function)∗

Listing 2: EBNF portion of the AsmetaL *rule* language

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

UpdateRule ::= Term ‘‘:=’’ Term

SkipRule ::= ‘‘skip’’

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

ExtendRule ::= **extend** <ID_DOMAIN> **with** VarTerm (‘‘,’’ VarTerm)∗ **do** Rule

The header section consists of some *import clauses* and one *export clause* which describe the ASM interface for the communication with other ASMs or

[3] We adopt the following conventions: non-terminals are plain, keywords are shown in **bold** face, and literal symbols are enclosed in double quotes. Moreover, words enclosed in angle brackets indicate a placeholder for a literal value.

[4] Keywords appear in **bold** face; a pair of square braces [] denotes that the enclosed expression is optional; a variable identifier starts always with an initial “$”; an enum literal is a string of length greater than or equal to two and consisting of upper-case letters only; a domain identifier begins always with an upper-case letter; a rule identifier always begins with the lower-case letter “r” followed by “_”; a function identifier always begins with a lower-case letter, but can not start with “r_”.

AsmM elements	Concrete syntax
ASM	(**asm**|**module**) name
Header	[**import**] m_1 [$(id_{11} \dots id_{1h_1})$] [**import**] m_k [$(id_{k1} \dots id_{kh_k})$] [**export** $(id_1 \dots id_e)$ | **export** *] **signature**: [$dom\_decl_1 \dots dom\_decl_n$] [$fun\_decl_1 \dots dom\_decl_m$]
where: - $(id_{i1} \dots id_{ih_i})$ are names for domains, functions and rules imported from module m_i (if omitted, all exported elements of m_i are imported) - $(id_1 \dots id_e)$ are names for exported domains, functions and rules (* to export all) - dom_decl_i and fun_decl_i are declarations of domains and functions	
Body	**definitions**: [**domain** $D_1 = DTerm_1$ ... **domain** $D_p = DTerm_p$] [**function** f_1 [$(p_{11}$ **in** D_{11} ... p_{1h_1} **in** $D_{1h_1})$] $= FTerm_1$ ... **function** f_q [$(p_{q1}$ **in** D_{q1} ... p_{qh_q} **in** $D_{qh_q})$] $= FTerm_q$] [$rule\_decl_1 \dots rule\_decl_r$] [$axiom\_decl_1 \dots axiom\_decl_s$]
where: - $DTerm_i$ and $FTerm_i$ are terms defining domains D_i and functions f_i - p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the function f_i - $rule\_decl_i$ and $axiom\_decl_i$ are declarations of rules and axioms	
Main rule	[**main** $rule\_decl$]
Initial state	[**default**] **init** sn: [**domain** $D_1 = DTerm_1$ ... $D_u = DTerm_u$] [**function** f_1 [$(p_{11}$ **in** D_{11} ... p_{1h_1} **in** $D_{1h_1})$] $= FTerm_1$ ... **function** f_v [$(p_{v1}$ **in** D_{v1} ... p_{vh_v} **in** $D_{vh_v})$] $= FTerm_v$] [**agent** A_1: $rule_1$ ... **agent** A_z: $rule_z$]
where: - sn is the name of the initial state - $DTerm_i$ and $FTerm_i$ specify the initial value of domains D_i and functions f_i - p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the function f_i - A_i and $rule_i$ are the agents and their associated programs	

Table 1: Template of AsmetaL programs

ASM modules. The *signature* contains the *declarations* of domains and functions occurring in the ASM. Every ASM is allowed to use only identifiers (for functions and rules) which are declared within its header's signature or imported from other modules. The imported functions will be statically added (together with their domains and codomains declarations) in the signature of the machine as completely new functions and the imported rules will enrich the module interface of the machine.

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

The main rule is a named transition rule denoted by the keyword `main`. It is closed (i.e. it does not contain free variables) so that its semantics depends only on the state of the machine. Executing an ASM means executing its main rule starting from one specified initial state.

The initialization section consists of a set of *initial states*, one of which is elected as *default*. An initial state defines an initial value for every dynamic function and every *concrete-domain* already declared in the signature of the ASM[5]. The initial state associates each *agent* domain (as subset of the predefined *Agent* domain) with its *program* (a named transition rule).

In [Gargantini et al., 2006a], we also provide guidance on how to automatically assemble a script file and give it in input to the JavaCC parser generator [JavaCC, 2008] to generate a parser for the EBNF grammar of the AsmetaL notation. This parser is more than a grammar checker: it is able to process ASM models written in AsmetaL, to check for their well-formedness with respect to the OCL constraints of the AsmM metamodel, and to create instances of the AsmM metamodel in a MDR MOF repository through the AsmM-JMI APIs. All OCL constraints have been syntactically checked by the OCLE OCL tool and implemented in Java as a well-formedness checker of AsmM models. This checker is used by the AsmetaL parser, but it can be also invoked by all tools within the ASMETA environment to check if a given model (or a subset of it, or just a model element) is well-formed or not with respect to the invariants defined over the AsmM metamodel.

We illustrate here the AsmetaL notation by three examples of increasing complexity.

4.1 The Flip-Flop device

Below, the ASM model of a Flip-Flop device is reported. The model originally presented in [Börger and Stärk, 2003, page 47] contains two rules. The first one

[5] Only dynamic (non-monitored) functions and concrete-domains need to be initialized.

(FSM) models a generic finite state machine. If in ctl_state $= i$ the condition *cond* is not satisfied, then a *persistent if-then* is achieved: the machine remains in ctl_state $= i$ until *cond* becomes true, in which case the machine proceeds to ctl_state $= j$. The second one (FLIPFLOP) instantiates the FSM for a Flip-Flop.

$$\textsc{Fsm}(i,cond,rule,j) = \textbf{if } \text{ctl\_state} = i \textbf{ and } cond \textbf{ then } \{rule, \text{ctl\_state} := j\} \textbf{ endif}$$

$$\textsc{FlipFlop} = \{\textsc{Fsm}(0,high,\textbf{skip},1),\textsc{Fsm}(1, low,\textbf{skip},0)\}$$

The rule FSM is a named parameterized rule or *macro*. A macro is used as an abbreviation to enhance the readability allowing modularization and stepwise refinement of large machines. An occurrence of such a rule, where a rule is expected (e.g. the applications of the FSM macro within the body of the FLIPFLOP rule above), stands for the corresponding rule, which is supposed to be defined somewhere else, with the parameters instantiated by legal values (data values or functions or rules) so that the resulting rule has a well-defined semantics.

Listing 3 shows the specification of the Flip-Flop device written in AsmetaL, including an example of axiom.

Listing 3: Flip-Flop Specification in AsmetaL

```
asm FlipFlop
import StandardLibrary
signature:
        domain State subsetof Integer
        dynamic controlled ctl_state: State
        dynamic monitored high: Boolean
        dynamic monitored low: Boolean
definitions:
    domain State = {0, 1}
    rule r_Fsm($i in State,$cond in Boolean,$rule in Rule,$j in State) =
        if ctl_state = $i and $cond then
            par
                $rule
                ctl_state := $j
            endpar
        endif
    axiom inv_neverBoth over high, low: not(high and low)
main rule r_Main = seq
                        r_Fsm[ctl_state, 0, 1, high, <<skip>>]
                        r_Fsm[ctl_state, 1, 0, low, <<skip>>]
                   endseq
default init s0: function ctl_state = 0
```

4.2 One-Way Traffic Light Control

This example is about a one-way traffic control. The problem description and the ASM model are taken from [Börger, 2007]. The AsmetaL specification presented

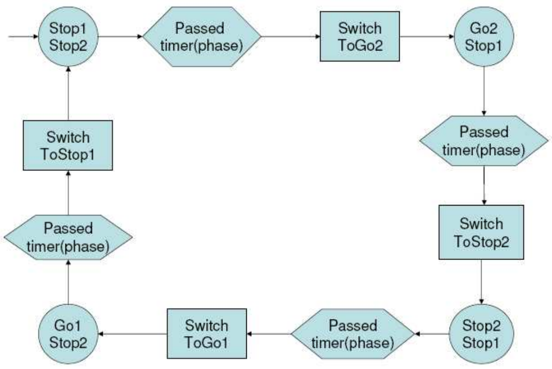

Figure 5: `oneWayTrafLightGround` Model

here is the result of a straightforward encoding of the original ASM specification, however it can be made more concise.

Problem description "... the traffic is controlled by a pair of simple portable traffic light units ... one unit at each end of the one-way section ... connect(ed) ... to a small computer that controls the sequence of lights. Each unit has a Stop light and a Go light. The computer controls the lights by emitting RPulses and GPulses, to which the units respond by turning the light on and off. The regime for the lights repeats a fixed cycle of four phases. First, for 50 seconds, both units show Stop; then, for 120 seconds, one unit shows Stop and the other Go; then for 50 seconds both show Stop again; then for 120 seconds the unit that previously showed Go shows Stop, and the other shows Go. Then the cycle is repeated."

In [Börger, 2007], a ground model is defined and shown to correctly capture the user requirements, based upon explicitly stated domain properties. A ground model refinement is then presented realizing some specific design decisions.

Ground Model. The problem is about two light units, each equipped with a $StopLight(i)$ and a $GoLight(i)$ $(i = 1, 2)$ which can be set on and off. In the ground model, the latter are controlled locations to which a value on or off can be assigned directly, abstracting from the computer emitting pulses. The ground model also abstracts from an explicit time computation and treats the passing of time by monitored locations $Passed(timer(phase))$, where the function *timer* defines the requested light regime. The monitored locations are assumed to become true in the model whenever (and only when) $timer(phase)$ has elapsed in the environment since the current phase was entered.

The sequence of lights starts with the phase $Stop1Stop2$ where for both the two light units $StopLight(i) = on$ and $GoLight(i) = off$. The overall behavior is expressed by the sequence of four phases as shown in Fig. 5 (from [Börger, 2007]) using a graphical notation for control-state ASMs. The submachine macros are defined below. After $timer(Stop1Stop2)$ has passed, the control executes a submachine `SwitchToGo2` and then enters phase $Go2Stop1$, followed upon $Passed(timer(Go2Stop1))$ becoming true by a `SwitchToStop2` to enter phase $Stop2Stop1$, then a `SwitchToGo1` to enter phase $Go1Stop2$ and finally a `SwitchToStop1` to return to phase $Stop1Stop2$.

In the above requirements description, the value of $StopLight(i)$ and the value of $GoLight(i)$ are implicitly viewed as complementary; this implies their switching can be done in parallel. Thus the two submachines `SwitchToGo`i and `SwitchToStop`i are copies of one machine:

$$\begin{aligned}&\texttt{SwitchToGo}i = \texttt{SwitchToStop}i = \texttt{Switch}(GoLight(i))\\&\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{Switch}(StopLight(i))\\&\textbf{where } \texttt{Switch}(Light) = (Light := Light') \; ('\text{ for complement})\end{aligned}$$

The light regime (50,120,50,120) associates to each phase its time length, represented by the function $timer(phase)$. Following the requirements, the function is assumed to be static[6], i.e. set before running the machine as follows:

$$timer(phase) = \textbf{case } phase \textbf{ of } \begin{array}{l} Stop1Stop2 : 50sec \\ Go2Stop1 : 120sec \\ Stop2Stop1 : 50sec \\ Go1Stop2 : 120sec \end{array}$$

Listing 4 shows the oneWayTrafLightGround model in AsmetaL. The specification continues in Listing 5 with the submachines for the *phase transitions* as shown in Fig. 5: from $Stop1Stop2$ to $Go2Stop1$, from $Go2Stop1$ to $Stop2Stop1$, from $Stop2Stop1$ to $Go1Stop2$, and from $Go1Stop2$ again to $Stop1Stop2$.

The specification terminates with a main rule consisting of the parallel execution of these phase transition rules, and an initial state assuming (according to the requirements) that the sequence of lights starts with the phase $Stop1Stop2$ with for both the two light units $StopLight(i) = on$ (true) and $GoLight(i) = off$ (false).

Ground Model Refinement. This refinement step introduces the software interface feature that relates R/G pulses of the computer to turning the light units on/off. The single abstract machine `oneWayTrafLightGround` is replaced by a multi-agent ASM `oneWayTrafLight` consisting of an environmental pulse-triggered machine `Pulses` and a software machine `oneWayTrafLightCtl`, obtained from `oneWayTrafLightGround` by refining the submachines `switchTo...`i to emitting pulses:

[6] A change request to include the possibility to configure the time intervals associated to the phases would make the *timer* function dynamic and controlled by the configuration machine.

Listing 4: `oneWayTrafLightGround` in AsmetaL (Part 1)

```
asm oneWayTrafLightGround
import StandardLibrary

signature:
    abstract domain LightUnit
    enum domain Phase = {STOP1STOP2|GO2STOP1|STOP2STOP1|GO1STOP2}
    dynamic controlled phase: Phase
    static lightUnit1: LightUnit //the two light units
    static lightUnit2: LightUnit
    dynamic controlled stopLight: LightUnit -> Boolean
    dynamic controlled goLight: LightUnit -> Boolean
    static timer: Phase -> Integer
    dynamic monitored passed: Integer -> Boolean

definitions:
    function timer($p in Phase) = switch($p)
                                    case STOP1STOP2 : 50
                                    case GO2STOP1 : 120
                                    case STOP2STOP1 : 50
                                    case GO1STOP2 : 120
                                  endswitch

    //macro Switch(Light) to switch the lights
    macro rule r_switch($l in Boolean) = $l := not($l)
    //submachines switchTo...i for i=1,2
    macro rule r_switchToGo($i in LightUnit) =
         par
            r_switch[goLight($i)]
            r_switch[stopLight($i)]
         endpar
      rule r_switchToStop($i in LightUnit) = r_switchToGo[$i]
```

$$
\begin{aligned}
&\texttt{1WayTrafLightCtl} = \texttt{1WayTrafLightGround}\\
&\textbf{where forall } i \in \{1,2\}\ \texttt{switchTo}\ldots i = \texttt{Emit}(RPulse(i))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \texttt{Emit}(GPulse(i))\\
&\texttt{Pulses} = \textbf{forall } i \in \{1,2\}\\
&\qquad\qquad \textbf{upon } Event(RPulse(i))\ \textbf{do}\ \texttt{Switch}(StopLight(i))\\
&\qquad\qquad \textbf{upon } Event(GPulse(i))\ \textbf{do}\ \texttt{Switch}(GoLight(i))
\end{aligned}
$$

The link between the two agents is provided by the following *Pulse Output Assumption*, which relates the software actions to what happens in the environment: $\texttt{Emit}(RPulse(i))$ yields $Event(RPulse(i))$ to happen in the environment; and $\texttt{Emit}(GPulse(i))$ yields $Event(GPulse(i))$ to happen in the environment. Observe that each software control step of the refined $\texttt{switchTo}\ldots i$ triggers an environment step of `Pulses`, which switches the corresponding lights. Thus one ground model step is refined to two steps in the refined multi-agent machine.

The corresponding AsmetaL notation is obtained from the previous one, according to the following. First, the signature is refined by adding the two agent

Listing 5: `oneWayTrafLightGround` in AsmetaL (Part 2)

```
    rule r_stop1stop2_to_go2stop1 =
        if phase=STOP1STOP2 and passed(timer(STOP1STOP2)) then
                par
                    r_switchToGo[lightUnit2]
                    phase:=GO2STOP1
                endpar
        endif
    rule r_go2stop1_to_stop2stop1 =
        if phase=GO2STOP1 and passed(timer(GO2STOP1)) then
                par
                    r_switchToStop[lightUnit2]
                    phase:=STOP2STOP1
                endpar
        endif
    rule r_stop2stop1_to_go1stop2 =
        if phase=STOP2STOP1 and passed(timer(STOP2STOP1)) then
                par
                    r_switchToGo[lightUnit1]
                    phase:=GO1STOP2
                endpar
        endif
    rule r_go1stop2_to_stop1stop2 =
        if phase=GO1STOP2 and passed(timer(GO1STOP2)) then
                par
                    r_switchToStop[lightUnit1]
                    phase:=STOP1STOP2
                endpar
        endif
main rule r_Main = par
                        r_stop1stop2_to_go2stop1[]
                        r_go2stop1_to_stop2stop1[]
                        r_stop2stop1_to_go1stop2[]
                        r_go1stop2_to_stop1stop2[]
                   endpar
default init s0:
    function stopLight($l in LightUnit) = true
    function goLight($l in LightUnit) = false
    function phase = STOP1STOP2
```

types and the R/G pulses as shown in Listing 6. Second, the `definitions` section (see Listing 7) now includes the refined version of submachines `switchTo...`i, the new submachine `Emit` and the `PULSES` program. The phase transition rules (from $Stop1Stop2$ to $Go2Stop1$, from $Go2Stop1$ to $Stop2Stop1$, from $Stop2Stop1$ to $Go1Stop2$, and from $Go1Stop2$ again to $Stop1Stop2$) remain as they are. Finally, the main rule is the sequential execution[7] of the two agents' programs, and the initial state (see Listing 7) now contains the setting to false of the R/G pulses and the agent initialization.

[7] See discussion on the necessity of sequentializing the agents'programs in Sect. 5.4.3.

Listing 6: `oneWayTrafLight` in AsmetaL (The Signature)

```
asm oneWayTrafLight
import StandardLibrary
signature:
 //old signature elements
 ...
 //new elements:
  //the two agent type
  domain PULSESAgent subsetof Agent
  domain ONEWAYTRAFLIGHTCTLTAgent subsetof Agent
  //R/G pulses, one per light unit
  dynamic shared rPulse: LightUnit −> Boolean
  dynamic shared gPulse: LightUnit −> Boolean
  //the two agents
  static pulses: PULSESAgent
  static onewaytraflightctl: ONEWAYTRAFLIGHTCTLTAgent
```

4.3 The clustering algorithm

In order to asses the actual power and level of abstraction of AsmetaL, we have specified a clustering algorithm as described in terms of ASMs by Jensen et alt. [Jensen et al., 2007]. They presented and compared several specifications of a classical clustering algorithm written in AsmL, CoreASM and also Java and reported fragments of them in their paper.

The clustering specification in AsmetaL is very similar (at least for the published parts) to the CoreASM model, since AsmetaL shares with CoreASM several keywords. The length (as number of lines) of the AsmetaL and CoreASM specifications is about the same (around 90 lines), even if the AsmetaL specification includes type declarations.

With respect to the CoreASM specification reported in [Jensen et al., 2007], we preferred to use static functions instead of turbo rules (which are not supported by our simulator yet) to compute the distance between two points and the `sum` of points in a cluster, which is a set of points. For example, the `sum` function is defined as static recursive function as follows.

```
// returns the sum of the points in a Cluster
function sumPoints($c in Cluster) =
  if isEmpty($c) then (0.0,0.0)
  else let ($one = first(asSequence($c))) in
        add($one, sumPoints(excluding($c,$one)))
       endlet
  endif
```

The complete AsmetaL specification can be found at the ASMETA web site [AsmM, 2006].

Listing 7: `oneWayTrafLight` (Definitions and initial state)

```
definitions :
  //old definitions
  ...
  //new definitions
  //submachine Emit
  macro rule r_emit($pulse in Boolean) = $pulse := true
  //submachines switchTo... i for i = 1,2
    rule r_switchToGo($i in LightUnit) = par
                                           r_emit [ rPulse ($i )]
                                           r_emit [ gPulse($i )]
                                         endpar
    rule r_switchToStop($i in LightUnit) = r_switchToGo[$i]
    //program PULSES
    rule r_pulses = forall $l in LightUnit do par
                //upon Event(RPulse(i))
                if (rPulse ($l )) then par
                                    r_switch [ stopLight ($l )]
                                    rPulse ($l ) := false
                                 endpar
                endif
                //upon Event(GPulse(i))
                if (gPulse($l )) then par
                                    r_switch [ goLight($l )]
                                    gPulse($l ) := false
                               endpar
                endif
      endpar
main rule r_Main = seq
                     program(onewaytraflightctl )
                     program(pulses)
                   endseq
default init s0:
   //old
   function stopLight ($l in LightUnit) = true
   function goLight($l in LightUnit) = false
   function phase = STOP1STOP2
   //new
   function rPulse ($l in LightUnit) = false
   function gPulse($L in LightUnit) = false
   agent ONEWAYTRAFLIGHTCTLTAgent: par
                                     r_stop1stop2_to_go2stop1 []
                                     r_go2stop1_to_stop2stop1 []
                                     r_stop2stop1_to_go1stop2 []
                                     r_go1stop2_to_stop1stop2 []
                                   endpar
   agent PULSESAgent: r_pulses[]
```

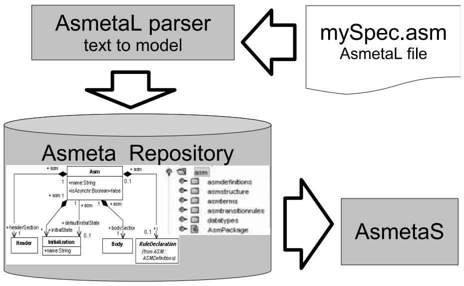

Figure 6: AsmetaS and the ASMETA Repository

5 The AsmetaS Simulator

In this section, we present the design of the ASMETA Simulator (or AsmetaS), its use, its main features and the validation activities we are carrying on.

AsmetaS is integrated in the ASMETA tool set and it operates directly on elements of ASM models in the ASMETA repository. Hence, the simulator reads ASM specifications in terms of JMI objects representing the specification the user wants to simulate. Thanks to this direct integration with the ASMETA repository, this simulation tool does not need to implement a parser, a type checker, and an internal representation of the model to simulate. The specification in the repository can be loaded from textual AsmetaL files by using the AsmetaL parser in our tool set (as shown in Fig. 6), but AsmetaS works regardless the way models are loaded in the repository.

Starting from the ASM model representation in terms of Java objects, at every step the simulator builds the update set according to the theoretical definitions given in [Börger and Stärk, 2003] to construct the *run* of the model under simulation. In the following, we explain the architecture we have designed to perform this task.

5.1 Basic Classes

At every execution step, the simulator must compute the value for every term and expression it evaluates in order to build the update set. We have introduced a class `Value` and its hierarchy (see Fig. 7) to represent all possible values of ASM locations. For every AsmM domain D, we have defined a D`Value` subclass which represents in Java the values of D. For simple domains the translation to Java is straightforward: we have used the corresponding Java types (e.g. values of the ASMETA Integer domain are mapped into the Java integers). Other structured domains required the use of other Java classes (like collections). Note that the encoding in Java of ASM values is approximate: for example, integers used by

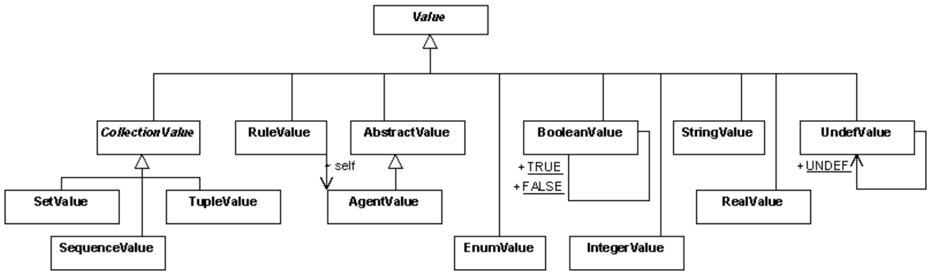

Figure 7: Value hierarchy

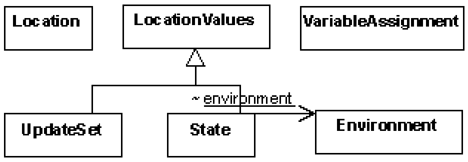

Figure 8: Basic classes

the simulator have a defined range (defined by the Java language), while the ASM integers are the mathematical integers.

Then we have introduced the class `Location` (see Fig. 8) to represent an ASM *location* and the abstract class `LocationValues` which maps locations to their values, i.e. `LocationValues` is a set of pairs (*location*,*val*). The class `LocationValues` has two subclasses: `State` which represents the state of an ASM, and `UpdateSet` which represents an update set. `VariableAssignment` maps logical or location variables (not nullary functions) to their values and it is used to evaluate a let rule, a let term or a macro call rule with parameters. The `Environment` class represents the stream from which to get the values of monitored functions. In the interactive mode (see Sect. 5.3), it will be instantiated by an interactive environment which asks to the user for values of monitored quantities. The state must keep a reference to the environment in use, since the value of monitored functions are provided by the environment.

5.2 AsmetaS Kernel

The simulator keeps the current state (an instance of the `State` class) of the ASM it is simulating and, on request, evaluates the values of terms and computes (and

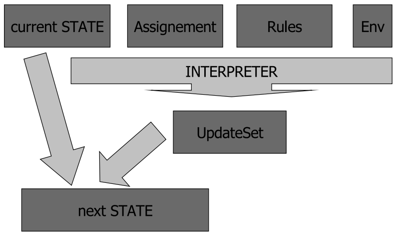

Figure 9: Evaluation process

applies) the update set (an instance of the `UpdateSet` class) to obtain the next state.

Regarding the evaluation of expressions, several solutions are possible: our main goal is to avoid the modification of the metamodel and to make the evaluation process modular and easy to modify and extend. Adding a method *value* in every subclass of the class `Term` in the AsmM would require the modification of the metamodel; this solution is difficult to maintain and expand, since it spreads the evaluation code in all the classes. The classical solution is to introduce one class representing the evaluation process, called `Visitor`, and to use a double dispatching pattern called *visitor* pattern. The visitor pattern would still require the addition of a single method *accept* in every `Term` subclass. The accept method invokes the visit method of the visitor it accepts.

To completely avoid any modification of the metamodel, we have defined a reflective visitor pattern instead of using the classical visitor pattern: the visitor class still defines a method *visit* for every `Term` subclass, but it also inherits a $visit(Object)$ from a `ReflectiveVisitor` which dispatches to the matching method by using the reflection mechanism and not by the methods *accept*. In this way the addition of a subclass in the hierarchy of the class `Term` would require only the addition of a method in the visitor class, while the introduction of another visiting operation would require the introduction of a new extension of the reflective visitor. The reflective visitor pattern proved to be very effective, and we have applied it also to perform other operations (rule evaluation, term and rule substitution, free variables finding, and user interface). For instance, the class `RuleEvaluator` which performs the very crucial task of computing the update set, extends the `ReflectiveVisitor`. It defines a method $visit(RuleType\ R)$, for every *RuleType* subclass of the `Rule` class of the AsmM. Given a rule R for which the simulator must compute the update set, the `RuleEvaluator` calls the matching visit method accordingly to the type of R to obtain the update set of R.

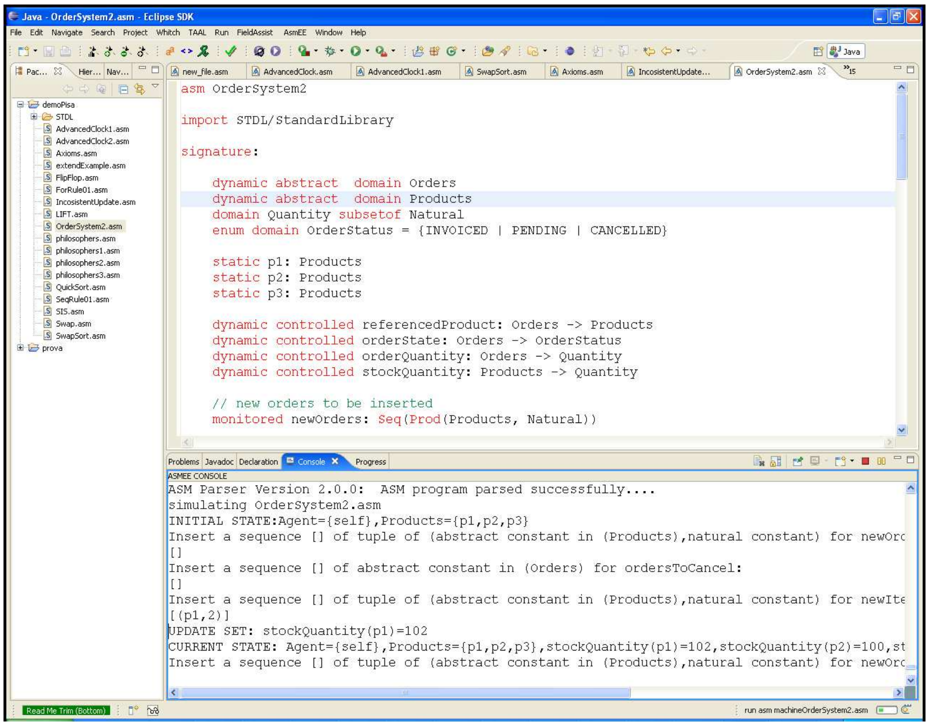

Figure 10: A screenshot of the ASMEE IDE.

5.3 How to use AsmetaS

AsmetaS can be used in a command line mode by passing as arguments the name of the specification file and some optional termination conditions for the run (it is possible to execute a fixed number of steps or to execute till empty-updates). We have developed also a graphical interface based on Eclipse, called ASMEE (ASMETA Eclipse Environment). ASMEE can be used as a graphical front end for the AsmetaL parser to edit ASM specifications (with syntax highlighting support and other editing features), and to export the XMI format of ASM specifications. ASMEE is also a graphical front end of AsmetaS and it allows the user to control the simulation and inspect its results (e.g., by performing single steps forward, observing the functions updates, etc.). The ASMEE can be seen as an IDE of ASM specifications. A screenshot of the ASMEE IDE is shown in Fig. 10.

Depending on the mechanism adopted to fetch values of monitored functions, the simulator can operates in two modes: *interactive mode* and *batch mode*. In the interactive mode, the simulator explicitly asks for values from the standard input device; in case of input errors, it alarms the user by printing an appropriate message on the standard output device inviting the user to address and remove the error. In batch mode, the simulator reads the functions values from a specific file with extension `.env`, containing all the values of monitored functions and in case of errors it terminates throwing an exception.

5.4 Key Features

In the following, we describe the key features currently supported by AsmetaS, with a particular emphasis on those helpful for model validation.

5.4.1 Supported Constructs

AsmetaS currently supports the simulation of Basic ASMs (single-agent) and synchronous multi-agent ASMs. Turbo ASMs rule constructors are not yet supported, except the sequence-rule. We plan to add the support for this kind of rules in the future. Regarding the non deterministic choice, AsmetaS supports the choose-rule with a real pseudo non determinism (i.e. there exist more than one evaluation of the same choose rule, starting from the same state, and producing different update sets). However, choose and forall constructs over infinite domains are unsupported, e.g. a "forall $x in Integer" term or rule is rejected.

No specific time model has been designed in AsmetaS, but it can provided in a conventional manner at specification level.

5.4.2 Recursive Functions

AsmetaS supports the interpretation of recursive static functions. Static functions should be used instead of value returning rules, which are not supported yet. For example, the following function `qsort` returns the ordered version of the sequence of integers taken as argument.

```
function qsort($s in Seq(Integer)) = if length($s) = 0n then [] else
  let ($pivot = first($s)) in union(
     union( qsort [$x | $x in $s with $x < $pivot]),[$y | $y in $s with $y = $pivot]),
         qsort([ $z | $z in $s with $z > $pivot]))
```

5.4.3 Axiom checking

AsmetaS implements an axiom checker, which (optionally) checks in every state reached during the computation if the axioms (if any) declared in the specification are satisfied or not. If an axiom is not satisfied, AsmetaS throws an `InvalidAxiomException`, which keeps track of the violated axiom and of the update set which has caused such violation.

The axiom checker is particularly useful during the first phase of the model development to validate the specification. The designer adds some invariants of the model as axioms, activates the axiom checker, and runs the model with some critical inputs. For example, to the refined version of the traffic light control specification presented in Section 4.2 we added the following axiom to check that the R and G pulses are not both true at the same time for the same light unit:

```
axiom inv_Pulses over gPulse, rPulse:
 forall $i in LightUnit with not (gPulse($i) and rPulse($i))
```

In the first version of the refined model, the software control machine and the pulse-triggered machine running in parallel (by using a **par** in the rule **r_Main** of Listing 7). This was proved to be a wrong refinement by finding an axiom violation during simulation. We realized that the pulse-triggered machine, which reacts to the pulses, must run only after the control machine has emitted the pulses and changed the phase. The problem was solved by defining a simple sequence scheduling between the two agents, as correctly reported in the specification in Listing 7.

The axiom checker was helpful to validate the clustering specification too. The clustering algorithm keeps track of the cluster centers by a controlled function `center` and when it moves a point from a cluster to another, updates the centers of the modified clusters in a quick way by considering only the old centers and the moved point (and not all the other points in the clusters). To check the correctness of such algorithm we have added the following axiom stating that the center of the each cluster (as mathematically defined) is equal to the `center` as updated by the algorithm.

```
axiom inv_center over center: forall $c in clusters with
  center($c) = (x(sumPoints($c)) / size($c), y(sumPoints($c)) / size($c))
```

This axiom helped us to discover faults in early versions of the specification. We had also to slightly modify the axiom above to consider rounding errors.

Although the axiom checker is useful for model validation, it significantly increases the execution time. This is evident for the clustering algorithm by comparing columns e and g of the table on the left in Fig. 11.

5.4.4 Consistent Updates checking

AsmetaS is able to reveal inconsistent updates throwing an `UpdateClashException`. The `UpdateClashException` records the location which are being inconsistently updated and the two different values which are assigned to that location. The user, analyzing this error, can detect the fault in the specification.

As the axiom checker, this feature is useful for model validation. For example, upon running a dining philosophers specification allowing synchronous parallel executions of all agents, inconsistent updates arise due to the tentative of agents to access shared resources at the same time. Several suitable scheduling policies among the agents avoiding any inconsistent update can be found in the specifications available at [AsmM, 2006].

5.4.5 Two extension mechanisms

AsmetaS can be extended in a *heavyweight* or *lightweight* way. Heavyweight extensions, to include, for example, new kinds of terms and rules, require the extension of the metamodel and, therefore, of the AsmetaS code. Lightweight extensions, like those two presented below to customize the interpretation of undefined static functions and the evaluation of monitored functions, do not require, instead, any change to the metamodel and the simulator.

5.4.5.1 Static function interpretation in Java

The first lightweight extension is the introduction of new static undefined functions whose interpretation is given in terms of Java code. In this case, the developer must (i) include the declaration of functions in an AsmetaL module (like `MyLib.asm`), (ii) write a class (like `MyLib.java`) with a static method having name and arguments equals to the new static functions and return value of type Value and (iii) associate the module and the class by calling a method *register* of the `StaticFunctionEvaluator`. We have adopted this simple mechanism for the functions declared in the StandardLibrary. When the `StaticFunctionEvaluator` finds a function f which has been declared in a module M but it has not been defined in M (like all functions of the StandardLibrary), it searches the class C registered with M, and invokes the Java method f of C.

This extension mechanism allows the direct definition of static functions in Java and it generally results in a faster execution. This feature was exploited to develop a version of the clustering specification where several static functions are defined in the Java class `ClusterFunctions` registered with the specification. The static function sum, for instance, becomes:

```
static public TupleValue sumPoints(SetValue sv){
   double dx = 0.0, dy = 0.0;
   for(Value v: sv.getValue()){
      dx += ((RealValue)((TupleValue)v).getValue()[0]).getValue();
      dy += ((RealValue)((TupleValue)v).getValue()[1]).getValue();}
  return new TupleValue({new RealValue(dx),new RealValue(dy)});
```

This clustering specification was much faster as reported in the table in Fig. 11, columns e and f.

5.4.5.2 Monitored function evaluation

The second lightweight extension mechanism allows the designer to extend the way the AsmetaS evaluates monitored functions (by default either from the console by asking to the user or from an environment file). This extension mechanism

a	b	c	d	e	f	g
2	6	10	1.6	0.2	2.4	0.45
4	12	10	2.2	0.94	3.0	1.3
25	75	100	29.1	24.9	159	26.2
33	99	120	43.8	40.0	SO	SO

a: number of points per clusters
b: total number of points
c: number of steps
d: average time of running the ASM model
e: average time of running the ASM model with JAVA interpretation of static functions
f: average time of running the ASM model with the axiom checker
g: average time of running the ASM model with the axiom checker and the JAVA interpretation of static functions
SO: stack overflow

Figure 11: Execution times of the clustering algorithm in AsmetaL

needs the definition of a new class implementing the `MonFuncReader` abstract class and passing an instance of the new class to the AsmetaS when starting the simulation. In this way, one may define a graphical environment which asks to the user the values of monitored variables by means of graphical dialogs or define an ad-hoc environment which reads the monitored quantities from an external device. Interactive and batch simulation (see Section 5.3) are performed by two basic `MonFuncReader` subclasses.

5.4.6 Random simulation

By using the second extension mechanism, we have introduced a random environment which produces random values for monitored functions. The random environment can be used by the developer to validate the specification against, for example, its axioms. We have performed a first validation of the AsmetaS code by using a random simulation (to see, for example, if all language constructs are supported, if `NullPointer` exceptions occur, and so on).

5.4.7 Logging

AsmetaS (and ASMEE) produces a minimal output to show the current state and the update set. Normally, the output is sent to the standard output (and to an XML file called `log.xml` in the working directory). However, the user can increase the output to inspect how the simulator performs particular tasks (including the evaluation of terms, the building of update set for rules, and the substitution of variables) by providing a log4j [LOG4j, 2008] configuration file in which he/she activates and sets the level of the logging facilities of AsmetaS classes.

The log messages are sent to the logger and formatted in XML. We have adopted the XML since the log output can be easily processed in this way by other tools to further analyze the runs produced by AsmetaS.

asmPath	nTimes	state()
interpreter/ArithmeticExpr01.asm	1	Agent={self},f=2 *expected* Agent={self},f=1 *actual*
interpreter/ArithmeticExpr02.asm	1	Agent={self},f=40
interpreter/ArithmeticExpr03.asm	1	Agent={self},f=27
interpreter/test_ge.asm	1	Agent={self},geval=true
interpreter/test_lt.asm	1	Agent={self},f=true
interpreter/nat_int_numbers.asm	1	Agent={self},f2=2,f1=0,f3=0 *expected* Agent={self},f1=0,f2=2,f3=0 *actual*
interpreter/ntoi.asm	1	Agent={self},counter2=2,counter3=1,counter=
interpreter/BooleanExpr01.asm	1	Agent={self},f=false
interpreter/RelationalExpr01.asm	1	Agent={self},f=true

Figure 12: Fit Table of test cases

5.5 Validation of AsmetaS

By executing most of the available AsmetaL specifications, we have performed a first validation of AsmetaS. However assessing the correctness of AsmetaS with respect to the ASM semantics by user simulation is error prone, since it is the user who judges the conformance of the actual outputs with respect to the expected ones, although the axiom and the inconsistent update checkers can help him/her to discover faults in the simulator too.

To provide a more effective and reliable way to assess AsmetaS correctness, we have implemented a wide range of JUnit test cases and Fit tables [FIT, 2007]. A Fit table is a simple table written for example in HTML which specifies some test cases (one for each row) by defining the expected outputs on given inputs. Our Fit tables define the expected final states of simple AsmetaL programs containing each only few types of terms or rules. Then, the tester runs the fit framework and the results are given again in a table which reports, besides the expected outputs, also the actual outputs. For example, the Fit table in Fig. 12 shows the result obtained running a set of test cases, each simulating a specific AsmetaL model (first column with header `asmPath`), for a given number of steps (second column `nTimes`), and for which we specify the expected final state (third column `state()`). Discrepancies of the expected final state and the actual final state are marked in red. Fit tables are available at the ASMETA web site [AsmM, 2006].

6 Related work

A number of ASM tools have been developed for model simulation.

The Abstract State Machine Language (AsmL) [ASML, 2001] 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” [Gurevich et al., 2004]. Adopting a terminology currently used, AsmL is a platform-specific modelling language for the .NET type system. A similar consideration can be made also for the AsmGofer language [Schmid, 2001]. 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 [Castillo, 2001], which adopts a functional style being developed in ML and which has inspired us in the language of terms, and XASM [Anlauff, 2000] which is integrated in Montages, an environment generally used for defining semantics and grammar of programming languages.

Recently, other simulation environments for ASMs have been developed, including the CoreASM [CoreAsm, 2008], an extensible execution engine developed in Java, TASM (Timed ASMs) [Ouimet and Lundqvist, 2007], an encoding of Timed Automata in ASMs, and a simulator-model checker for reactive real-time ASMs [Vasilyev, 2007] able to specify and verify First Order Timed Logic (FOTL) properties on ASM models. Among these, the CoreASM engine is the more comparable to our.

Like our simulator, CoreASM is a general-purpose ASM simulator, is written in Java, and its textual syntax for writing ASM specifications and our AsmetaL notation are very similar (at least in defining the rule schemes). While we statically enforce type correctness, since we perform the type checking prior execution during the evaluation of the OCL constraints defined over the AsmM metamodel for the functions' domains, the CoreASM supports dynamic type checking.

Although dynamic type checking gives more freedom and flexibility to the modeler, this flexibility is at the cost that type checking errors occur unpredictably at run-time and that type errors are detected only if executed. The advantages of static checking are that potential errors can be identified earlier, the specification is better documented, more care is needed in the design, and implementations can take advantage of the additional information to produce more efficient programs with less runtime checking code. Moreover, a CoreASM specification is structurally made of a *header block*, where various definitions take place, and a *rule declaration block* for the rules definitions (including the *init rule* that creates the initial state); however, the CoreASM Kernel does not define anything for the header section. What goes into the header section depends on the *plugins* that are used. Most of the functionalities of the CoreASM engine are implemented, in fact, through plug-ins to the basic kernel. The architecture supports three classes of plug-ins: backgrounds (provide all that is needed to define and work with new backgrounds), rules (to implement specific rule forms) and policies (to implement specific scheduling policies for multi-agent ASMs). Although the plug-in mechanism makes the CoreASM architecture *extensible*, few standard plugins come with the engine and the development of new ones is not so easy as it requires, especially for background plug-ins, an extension of the parser defining the concrete syntax (operators, literals, static functions, etc.) needed for working with elements of the background, an extension to the abstract storage providing encoding and decoding functions for representing elements of the background for storage purposes, and an extension to the interpreter providing the semantics for all the operations defined in the background. Clearly, all these extensions require a certain effort, expertise in Java programming, and knowledge of the CoreAsm extension mechanisms, which do not follow extension standards like those proposed by OSGi [OSGi, 2008] or by Eclipse. Our ASMETA framework does not support the extension via plugins; it can be extended in a more classical heavyweight way only by adding new classes to the metamodel for new concepts, for example for new kinds of terms or rules. These new classes should be sub-classes of existing ones. Then the new Java APIs, and the grammar rules would be (semi)automatically generated for such new constructs, while the extension of the simulator would require the definition of new *visit* methods to the evaluators. We believe that the effort required by extending ASMETA should be comparable with that required by CoreAsm, which is built for extensibility, since ASMETA extensions would be based on known standards (like e.g. EMF, Java inheritance, and the use of reflective visitors). As we tried to make AsmM, and therefore AsmetaL, as complete (i.e. representing all the ASM concepts) as possible, heavyweight extensions are usually not needed. However, AsmetaS offers two lightweight extension mechanisms to customize both the interpretation of undefined static functions and the evaluation of monitored functions, as already explained in Sect. 5.4.5.

7 Conclusions and future directions

We have presented the ASMETA tool set for Abstract State Machines, and in particular the ASMETA language and the simulation engine for writing and executing ASM models, respectively.

The ASMETA framework has been developed exploiting the MDE metamodelling approach. It is based on an abstract specification language, namely the

AsmM metamodel, which represents a set of mathematical concepts used for the definition of ASMs, and acts as an interlingua among tools. The concrete notation, AsmetaL, and its parser have been constructed in a generative manner from the ASMETA framework to effectively write ASM models. An alternative visual notation is also being defined to this purpose, but limited to control-state ASMs. The developed simulation engine makes formal models executable and assists, therefore, the modeler in identifying omission and logical errors. A graphical front-end called ASMEE (ASM Eclipse Environment) has been implemented as an Eclipse plug-in to allow editing and manipulation of AsmM models within an integrated development environment.

Although the ASMETA framework is targeted to the ASMs, our approach can be applied to any other formal method to develop a tool set around it.

Future work will include the integration of more existing tools and the development of new ones in the ASMETA tool set. We believe the development of code engineering tools (including code generation, reverse engineering, and synchronized round-trip engineering) supporting specific compilation techniques is an easy task to accomplish by implementing appropriate *walkers* capable of navigating throughout the AsmM abstract storage.

Moreover, we are evaluating other metamodelling frameworks to better support *model transformations* such as the ATL language [AMMA, 2005], the Xactium XMF Mosaic [XMF Mosaic, 2007], to name a few, and *model evolution activities* [Mens et al., 2005] such as model refinement, model refactoring, model inconsistency management, etc. Today, only limited support is available in model-based 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, grammarware, aspect-oriented software development, etc.

References

[AMMA, 2005] AMMA (2005). The AMMA Platform. `http://www.sciences.univ-nantes.fr/lina/atl/`.

[Anlauff, 2000] Anlauff, M. (2000). XASM - An Extensible, Component-Based ASM Language. In *Abstract State Machines*, pages 69–90.

[ASML, 2001] ASML (2001). The ASML language. `research.microsoft.com/foundations/AsmL/`.

[AsmM, 2006] AsmM (2006). The Abstract State Machine Metamodel website. `http://asmeta.sf.net/`.

[ATGT, 2008] ATGT (2008). ATGT: ASM Tests Generation Tool. `http://cs.unibg.it/gargantini/projects/atgt/`.

[Bézivin, 2004] Bézivin, J. (2004). In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2):21–24.

[Börger, 2007] Börger, E. (2007). The Abstract State Machines Method for High-Level System Design and Analysis. Technical report, BCS Facs Seminar Series Book.

[Börger and Stärk, 2003] Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag.

[Cabot and Teniente, 2006] Cabot, J. and Teniente, E. (2006). Constraint support in MDA tools: A survey. In *ECMDA-FA, Proceedings*, volume 4066 of *LNCS*. Springer.

[Castillo, 2001] Castillo, G. D. (2001). 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.

[CoreAsm, 2008] CoreAsm (2008). The CoreASM Project. `http://www.coreasm.org/`.

[Efftinge, 2006] Efftinge, S. (2006). oAW xText - A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*.

[EMF, 2008] EMF (2008). Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.

[FIT, 2007] FIT (2007). Fit: Framework for integrated test. `http://fit.c2.com/`.

[Gargantini and Riccobene, 2000] Gargantini, A. and Riccobene, E. (2000). Encoding Abstract State Machines in PVS. In et al., Y. G., editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag.

[Gargantini et al., 2003] Gargantini, A., Riccobene, E., and Rinzivillo, S. (2003). Using spin to generate tests from ASM specifications. In *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer.

[Gargantini et al., 2006a] Gargantini, A., Riccobene, E., and Scandurra, P. (2006a). Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware. In *3M4MDA'06 workshop at the European Conference on MDA*.

[Gargantini et al., 2006b] Gargantini, A., Riccobene, E., and Scandurra, P. (2006b). Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan.

[Gargantini et al., 2007a] Gargantini, A., Riccobene, E., and Scandurra, P. (2007a). A metamodel-based simulator for ASMs. In Prinz, A., editor, *Proceedings of the 14th International ASM Workshop*.

[Gargantini et al., 2007b] Gargantini, A., Riccobene, E., and Scandurra, P. (2007b). Ten reasons to metamodel ASMs. In *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, LNCS Festschrift*. Springer.

[GME, 2006] GME (2006). The Generic Modeling Environment (GME). `http://www.isis.vanderbilt.edu/Projects/gme`.

[GMF, 2008] GMF (2008). The Eclipse Graphical Modeling Framework. `http://www.eclipse.org/gmf/`.

[Gurevich et al., 2004] Gurevich, Y., Rossman, B., and Schulte, W. (2004). Semantic essence of AsmL. Technical Report MSR-TR-2004-27, Microsoft Research.

[Hearnden et al., 2002] Hearnden, D., Raymond, K., and Steel, J. (2002). Anti-Yacc: MOF-to-text. In *Proc. of EDOC*, pages 200–211.

[HUTN, 2004] HUTN (2004). OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. `http://www.uml.org/`.

[JavaCC, 2008] JavaCC (2008). Java Compiler Compiler. `https://javacc.dev.java.net/`.

[Jensen et al., 2007] Jensen, O., Koteng, R., Monge, K., and Prinz, A. (2007). Abstraction using ASM tools. In *The 14th International ASM Workshop*.

[JMI, 2002] JMI (2002). Java Metadata Interface Specification, Version 1.0. `http://java.sun.com/products/jmi/`.

[Jouault et al., 2006] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). ATL: a QVT-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM.

[Jouault and Bézivin, 2006] Jouault, F. and Bézivin, J. (Bologna, Italy, 2006). KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*.

[Kent, 2002] Kent, S. (2002). Model Driven Engineering. In *IFM '02: Proc. of the Third International Conference on Integrated Formal Methods*, pages 286–298. Springer-Verlag.

[Kurtev et al., 2002] Kurtev, I., Bézivin, J., and Aksit, M. (Irvine, USA, 2002). Technical Spaces: An Initial Appraisal. In *CoopIS, DOA'2002, Federated Conferences, Industrial track*.

[LOG4j, 2008] LOG4j (2008). Log4J. `http://logging.apache.org/log4j`.

[MDR, 2003] MDR (2003). The Model Driven Repository for NetBeans. `http://mdr.netbeans.org/`.

[Mens et al., 2005] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE'05)*.

[Microsoft DSL Tools, 2005] Microsoft DSL Tools (2005). Microsoft DSL Tools. `http://msdn.microsoft.com/vstudio/DSLTools/`.

[MOF, 2002] MOF (2002). OMG. The Meta Object Facility (MOF) v1.4, formal/2002-04-03.

[Nytun et al., 2006] Nytun, J. P., Prinz, A., and Tveit, M. S. (2006). Automatic generation of modelling tools. In *Proc. of ECMDA-FA*, pages 268–283.

[OCLE, 2005] OCLE (2005). OCL Environment (OCLE). `http://lci.cs.ubbcluj.ro/ocle`.

[OMG, 2008] OMG (2008). The Object Managment Group (OMG). `http://www.omg.org`.

[OSGi, 2008] OSGi (2008). OSGi Alliance. `http://www.osgi.org/`.

[Ouimet and Lundqvist, 2007] Ouimet, M. and Lundqvist, K. (2007). The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. In *Proceedings of the 14th International Workshop on Abstract State Machines (ASM '07)*.

[Riccobene and Scandurra, 2004] Riccobene, E. and Scandurra, P. (2004). Towards an Interchange Language for ASMs. In Zimmermann, W. and Thalheim, B., editors, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer.

[SAL, 2008] SAL (2008). The Symbolic Analysis Laboratory. `http://sal.csl.sri.com/`.

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

[Schmid, 2001] Schmid, J. (2001). AsmGofer. `http://www.tydo.de/Doktorarbeit/AsmGofer`.

[Sztipanovits and Karsai, 1997] Sztipanovits, J. and Karsai, G. (1997). Model-integrated computing. *IEEE Computer*, 30(4):110–111.

[Vasilyev, 2007] Vasilyev, P. (2007). Simulator-model checker for reactive real-time abstract state machines. In *Proceedings of the 14th International ASM Workshop (ASM'07)*. http://rotor.di.unipi.it/AsmCenter/.

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

[XMF Mosaic, 2007] XMF Mosaic (2007). The Xactium XMF Mosaic. `www.modelbased.net/www.xactium.com/`.