A Metamodel-based Simulator for ASMs

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

[1] Dip. di Ing. Informatica e Metodi Matematici, Università di Bergamo, Italy
angelo.gargantini@unibg.it
[2] Dip. di Tecnologie dell'Informazione, Università di Milano, Italy
{riccobene,scandurra}@dti.unimi.it

Abstract In this paper we present a general-purpose simulation engine for ASM specifications. It has been developed as part of the ASMETA (ASMs metamodelling) toolset which is a set of tools for ASMs based on the metamodelling framework of the Model-Driven Engineering. We briefly present the ASMETA framework, how it has been developed, the concrete textual notation or language (AsmetaL) it adopts for effectively writing ASM specifications and the Asmeta simulator (called AsmetaS). We explain the architecture of the simulator, its kernel engine, how it works within the ASMETA tool set, and how it takes advantages from the metamodelling approach. We discuss the features currently supported by the simulator and how it has been validated.

1 Introduction

The Abstract State Machines (ASMs) [12] are nowadays acknowledged as a formal method successfully employed as systems 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 [31,18,35], and execution engines for simulation and testing purposes [32,8,36,14,34,15,19].

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. 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 mathematical concepts of the ASMs. Therefore, ASM tools are loosely coupled and their integration 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. The result is that a practitioner willing to use ASM tools needs to know all the different syntaxes and that most ASM researcher papers, presentations and examples, still use their own ASM notation, normally not defined

by a grammar but in terms of mathematical concepts. 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 ASM, 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) [11,23,26].

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, and allowing to settle a flexible object-oriented infrastructure for tools development and interoperability. A metamodel-based abstract syntax definition has the great 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. Therefore, a metamodel could serve as *standard interlingua* on which tool development should be based.

Furthermore, metamodelling allows to settle a "global framework" to enable otherwise dissimilar languages (of possibly different domains) to be used in an interoperable manner in different *technical spaces*, namely working contexts with a set of associated concepts, knowledge, tools, skills, and possibilities. Indeed, it allows establishing precise *bridges* (or *projections*) among the metamodels of these different domain-specific languages to automatically execute model transformations.

Exploiting the advantages offered by the metamodelling approach, we have developed an ASM metamodelling (ASMETA) framework which provides a global infrastructure for interoperability of ASM application tools (new and existing ones) including ASM model editors, ASM model repositories, ASM model validators, ASM model verifiers, ASM simulators, ASM-to-Any code generators, etc. The ASMETA framework helps developers to build new tools by providing an XMI-based interchange format, standard JMI APIs, and several MOF-related facilities which supply standard projections toward other technical spaces. A developer who is interested in developing a new tool for the ASMs can completely base the tool development on the ASMETA framework and exploit all technologies provided by ASMETA in terms of specification language, abstract storage (i.e. the MOF-based model repository), APIs, interchange format, etc.

In this paper we present a case study of a tool based on the ASMETA framework: a *simulator* (called AsmetaS), written in Java, to make the ASM models executable. We present the architecture of the AsmetaS, its execution engine, how it works within the ASMETA tool set, and how it takes advantages from the metamodelling approach. We also discuss the features supported by the simulator, how it can be extended, and how it has been validated.

The paper is organized as follows. Sect. 2 presents the overall development process of the ASMETA framework and the ASMETA tool set. Sect. 3 presents a language we have defined to write ASMETA models in a textual notation. The simulator is presented in Sect. 4. Related and future work are given in sections 5 and 6, respectively.

2 The ASMETA framework

ASMETA is an *instantiation* of the OMG metamodelling framework for the ASMs, and it has been developed to create and handle ASM models exploiting the advantages offered by the metamodelling approach and its related facilities (in terms of derivatives, libraries, APIs, etc.).

In the following, we first explain the process, based on the metamodelling approach, of developing a tool set around the ASM formal method, and then we present the ASMETA toolset.

2.1 The ASMETA development process

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

1. Choice of a metamodelling framework and supporting technologies;
2. Development of the metamodel;
3. Metamodel derivatives development as additional facilities to handle models:
   (a) model serialization (like the OMG's XMI standard, an XML-based format, OMG's CORBA Metadata Interfaces, etc.);
   (b) APIs to access and manipulate models in a model repository (like JMIs, CORBA IDLs, etc.);
   (c) one or more concrete syntaxes (textual, graphical, or mixed) with their parsers for type-checking and model storing;
4. Development of tools based on the metamodelling framework and SW artifacts to integrate existing tools with the metamodelling framework;
5. Validation of the metamodel and its derivatives.

The process may turn out to be iterative, to come back to previous steps and make corrections.

To implement the ASMETA as MDE framework, we chose the OMG metamodelling platform even though many other implementations of the MDE principles exist, like the AMMA metamodelling platform [5], the Xactium XMF Mosaic [7] initiative, the Model-integrated Computing (MIC) [25] and its tool-suite, the Software Factories and their Microsoft DSL Tools [4]. 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 [6] as model repository, the Poseidon UML tool (v. 4.2) as metamodel editor, the XMI 1.2 format and JMIs as generated by the MOF MDR Netbeans framework, and the OCL support provided by the OCLE tool [27]. Note that

current UML tools present several limitations regarding the support of OCL constraints [13].

Steps 2. and 3. lead to the *instantiation* of the OMG metamodelling framework, that we call ASMETA. We started by defining the *AsmM*, a metamodel for ASMs [21,9], and then we derived from the metamodel some additional facilities to handle models: an XMI (XML Metadata Interchange) [28] interchange format for ASM models; JMI (Java Metadata Interfaces) APIs for the creation, storage, access and manipulation of ASM models in a MOF-based instance repository; a concrete textual notation, called AsmetaL (ASMETA Language), and its parser to effectively edit ASM models conforming to the AsmM metamodel. Details on the development of the metamodel and its derivatives can be found in [21,9]. How to build SW artifacts to integrate external and existing tools without forcing them to waive their own notation and internal data representation, is reported in [21].

The validation process applies 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. This was our approach. We have 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 [12] and other ASM case studies to AsmetaL. The task was completed within three man-months, by encoding almost all the examples provided in [12]. Up to now we have several ASM specifications encoded in AsmetaL and available in [9].

The validation of the metamodel derivatives consists 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 [21] details can be found on how the ASM tools interoperability is achieved by the ASMETA.

2.2 The ASMETA tool set

Fig. 1 shows our plan of the ASMETA tool set which: (a) provides an intuitive modeling notation having rigorous syntax and semantics, and provides a graphical view of the model; (b) allows modeling techniques which facilitate the use of the ASMs in many stages of the development process and which integrates dynamic (operational) and static (declarative) descriptions, and 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 (c) supports an open and flexible architecture to make easier the development of new tools and integration with other existing tools.

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

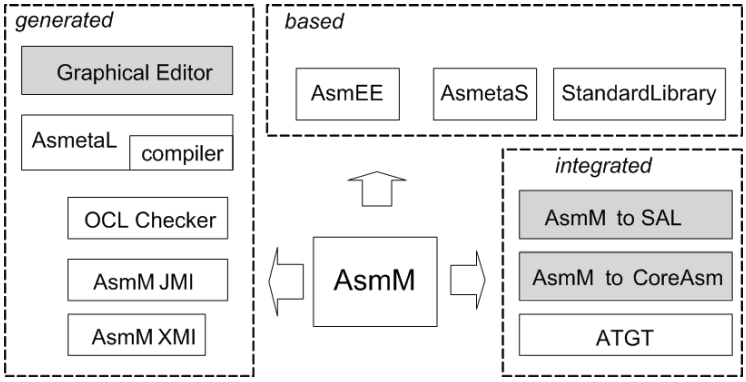

Figure 1. The ASMETA tool set

– The **AsmM metamodel**, based on MOF 1.4, (the *abstract syntax*), edited with the Poseidon tool v.2.6 according to a UML profile for MOF 1.4. It represents in an abstract way concepts and constructs of the ASM formalism as described in [12]. It 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 (Sync/Async) ASMs, so reflecting the natural classification of abstract state machines. The complete metamodel contains 115 classes, 114 associations, and 150 OCL class invariants, approximatively.

– The **AsmM OCL checker**, based on the OCLE [27] tool and 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 runtime) in terms of Java objects.

– 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, for the interchange of ASM models among tools by XML serialization. The AsmM-XMI format and the AsmM-JMIs have been generated automatically from the AsmM metamodel by using the MOF MDR (Model Driven Repository) for NetBeans [6].

– 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** to parse the 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 which implement a set of canonical operations on domains.

– A **graphical notation**, generated from the AsmM (the abstract syntax) as an alternative *concrete syntax* to be used by modelers to effectively write ASM models in a graphical form.

– 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 **ASM Tests Generation Tool** (ATGT) [10], an existing tool for test case generation from models, which has been made AsmM-compliant.

– The **AsmM-to-CoreAsm** and **AsmM-to-SAL** components which export ASMETA models to the CoreASM [15] simulation tool and to the SAL verifier [29].

– 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. 4). 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 [9], under GNU General Public License (GPL).

3 AsmetaL programs

The ASMETA Language (or AsmetaL) is a metamodel-based language, in the sense that it has been defined exploiting the MOF-to-text approach suggested by the MDE and applicable to our ASM metamodel to derive a concrete syntax compliant to the metamodel.

Initially, we investigated the use of tools like HUTN (Human Usable Textual Notation) [22] or Anti- Yacc [16] 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 strongly reflecting the object-oriented nature of the MOF meta-language, while ASM is not an object-oriented formalism (even though it can model OO concepts). There are better MOF-to-grammar tools

now, like xText [17] of OpenArchitectureWare or TCS of AMMA [5], which we may consider to adopt in the future.

In [20] we define 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).

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

Since this paper focuses on the simulator, we do not present here details of AsmetaL. Its complete EBNF grammar can be found in [21,9] and a detailed user guide is available at [9]. Note that a previous version of the language, called AsmM concrete syntax, was presented in [30].

In [20], we also provide guidance on how to automatically assemble a script file and give it in input to the JavaCC parser generator [2] 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 use of the AsmM-JMIs. All OCL constraints have been syntactically checked and implemented by the OCL checker OCLE. 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 metamodel.

ASM specifications encoded in AsmetaL (and suitable to be parsed by the simulator) reflect the *working definition* of an ASM model as given in [12] and are structured into four sections: a *header*, a *body*, a *main rule* and an *initialization*. Fig. 2 shows a template of AsmetaL programs[1]. The name of the ASM is specified before the header section together with the optional keyword `isAsyncr` to specify if the ASM is an asynchronous multi-agent or not. For single-agent ASMs, `isAsyncr` has no meaning. Since we consider an ASM *module* as an ASM

[1] We adopt the following conventions: keywords appear in **bold** face; a pair of square braces [] (not in bold face) indicates that the enclosed expression is optional; 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_".

without the main rule and without a set of initial states, a module is specified like an ASM but replacing the keyword `asm` by the keyword `module`.

The header section consists of some *import clauses* and one *export clause* which describe the ASM interface for the communication with other ASMs or ASM modules. The *signature* contains the *declarations* of domains and functions 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. If the ASM has no main rule, as default, the ASM is started executing in parallel the agent programs given by the agent initialization clauses in a specified initial state.

The initialization section consists of a set of *initial states*, one of which is elected as *default*. Fig. 2 shows the schema of an initial state. An initial state defines an initial value for every dynamic function and every *concrete-domain* already declared in the signature of the ASM[2]. The initial state associates each *agent* domain (as subset of the predefined *Agent* domain) with its *program* (a named transition rule).

Fig. 3 shows the specification written in AsmetaL of a Flip-Flop device. The model, originally presented in [12, page 47] and reported below, 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:

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

4 AsmetaS

In this section we present the basic design of the ASMETA Simulator, its use, its main features and the validation activities we are carrying on. AsmetaS is integrated in the ASMETA tool-set framework and it operates directly on instances

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

AsmM Element	Concrete Syntax
ASM	[**asyncr**] (**asm**|**module**) name
Header	[**import** m_1[(**id**$_{11}$,...,**id**$_{1h1}$)] ... **import** m_k[(**id**$_{k1}$,...,**id**$_{khk}$)]] [**export** **id**$_1$,...,**id**$_s$] or [**export** *] **signature:** [dom_decl$_1$ ... dom_decl$_n$] [fun_decl$_1$ ... fun_decl$_m$]
- **id**$_{i1}$,...,**id**$_{ihi}$ are names for domains, functions and rules imported from module m_i (if omitted all the content of the export clause of m_i is imported) - **id**$_1$,...,**id**$_s$ are names for domains, functions and rules exported from the ASM (export * is used to export all functions and rules) - dom_decl$_i$ and fun_decl$_i$ are declarations of domains and functions	
Body	**definitions :** [**domain** D_1= Dterm$_1$ ... **domain** D_n= Dterm$_n$] [**function** f_1[(p_{11} **in** D_{11},...,p_1**in** D_{1k1})]= Fterm$_1$ ... **function** f_f[(p_{f1} **in** D_{f1},...,p_f**in** D_{fkf})]= Fterm$_f$] [rule_decl$_1$ ... rule_decl$_e$] [axiom_decl$_1$ ... axiom_decl$_v$]
-Dterm$_i$ and Fterm$_i$ are terms specifying the definitions of the domains D_i and functions f_i, - p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the functions f_i, -rule_decl$_i$ and axiom_decl$_i$ are declarations of rules and axioms	
Main Rule	[**main** rule_decl]
Initial State	[**default**] **init** sn: [**domain** D_1= Dterm$_1$ ... **domain** D_n= Dterm$_n$] [**function** f_1[(p_{11} **in** D_{11},...,p_{1t1} **in** D_{1t1})]= Fterm$_1$ ... **function** f_m[(p_{m1} **in** D_{m1},...,p_{mtm} **in** D_{mtm})]= Fterm$_m$] [**agent** A_1: r_1 ... **agent** A_u: r_u]
— sn is the name of the initial state, - Dterm$_i$ and Fterm$_i$ are terms specifying the initial value of the domains D_i and functions f_i, - p_{ij} are variables ranging in the domain D_{ij} and specifying the formal parameters of the functions f_i, - A_i and r_i are the agents domains and their associated *programs*	

Figure 2. Template of AsmetaL programs

```
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 3. Flip-Flop Specification in AsmetaL

of ASM models in the ASMETA repository which is a metadata repository based on the Netbeans [6] libraries. Hence, the simulator reads ASM specifications in terms of instances of JMI objects representing the specification the user wants to simulate; therefore, it 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. 4), 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 [12] to construct the *run* of the model under simulation. In the following, we explain the architecture we have designed to perform this task.

4.1 Basic Classes

At every step of the execution, 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. 5) to represent all the possible values of ASM locations. For every AsmM domain D, we have defined a

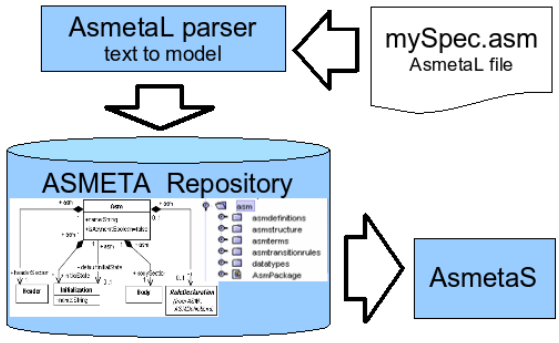

Figure 4. AsmetaS and the ASMETA Repository

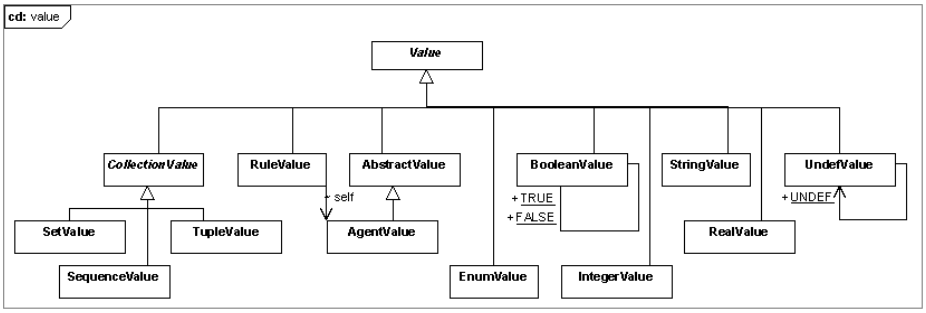

Figure 5. Value hierarchy

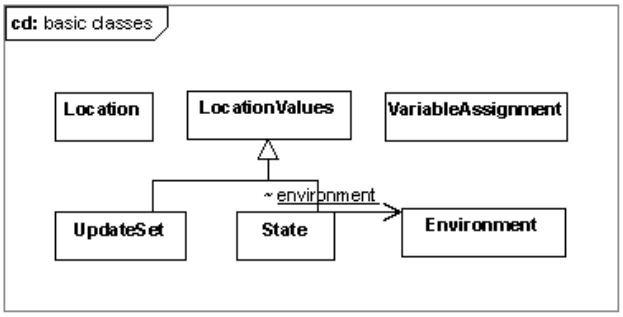

Figure 6. Basic classes

D`Value` subclass which represents in Java the values of *D*. For simple domains the translation to Java is straightforward: we have used the correspondent Java types (e.g. values of the ASMETA Integer domain are represented by 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 the integers used by 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. 6) 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 where to get the values of monitored functions. In the interactive mode (see Sect. 4.3), it will be instantiated by an interactive environment which asks to the user for the 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.

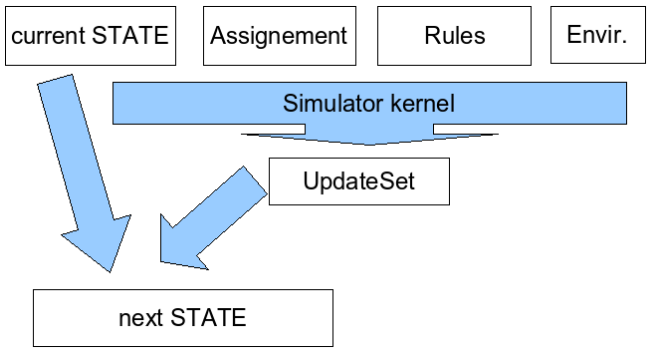

Figure 7. Evaluation process

4.2 AsmetaS Kernel

The simulator keeps the current state (an instance of `State` class) of the ASM it is simulating and on request evaluates the values of terms and computes (and applies) the update set (an instance of `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, and Moreover, 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 accept methods. 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). The complete hierarchy of the visitor pattern used is shown in Fig. 8 [3].

To compute the update set, a `RuleEvaluator` which extends the `ReflectiveVisitor` is introduced. 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.

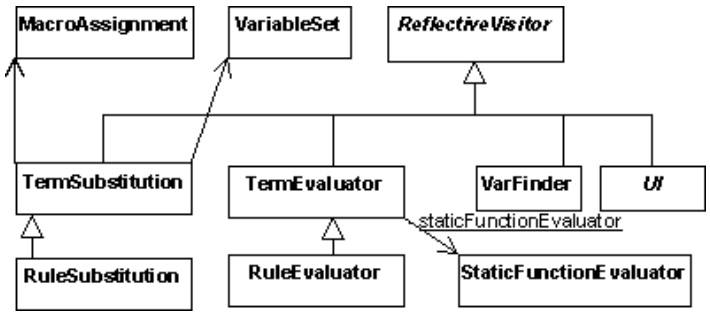

Figure 8. Reflective Visitor Classes

The same reflective visitor pattern is applied also for variable substitution in terms and rules (in case of macro call rules with parameters). If a $R(x)$ is

[3] Note that the content of Fig. 5, 6, and 8 could be back annotated at metamodel level, by defining a separate, but related, metamodel which would represent these extra information for the evaluation of terms.

called with $x = t$, the `RuleSubstitution` visits R and returns a new rule with x substituted by t.

4.3 How to use AsmetaS

AsmetaS can be used in a command line mode. In this mode it is invoked from a shell by passing it as arguments the name of the specification file and some optional termination conditions for the run (option `-n 10` for a fixed number of steps, or option `-n?` 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. 9.

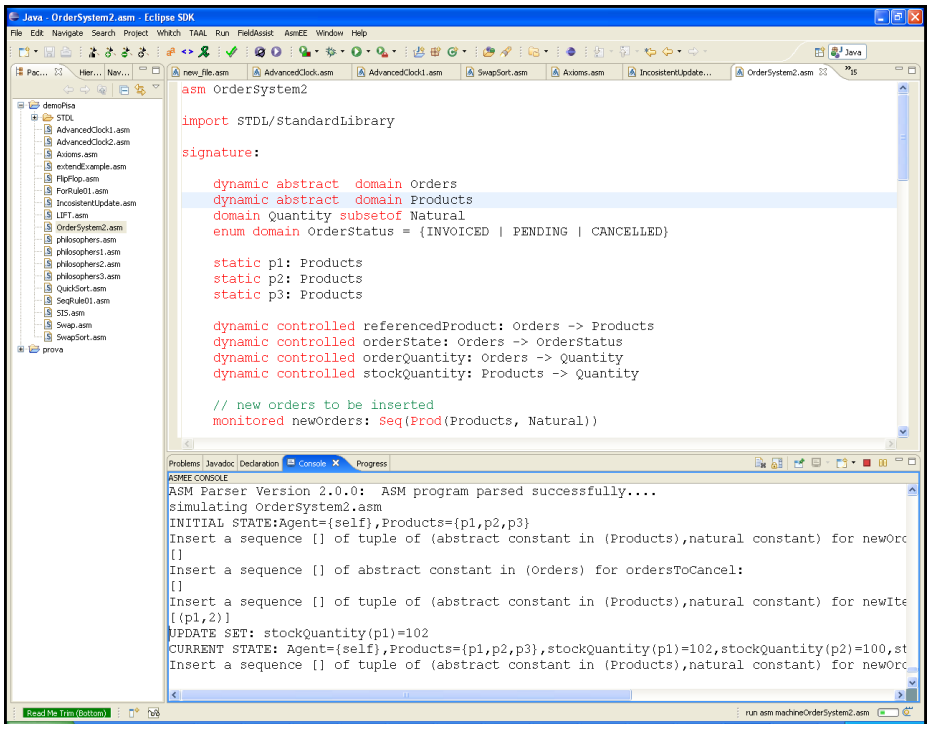

Figure 9. A screenshot of the ASMEE IDE.

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.

4.4 Key Features

A list of the key features currently supported by AsmetaS follows.

Supported Constructs AsmetaS currently supports all the terms in the meta-model, all the rules except the TurboRules and their derived, but it does support the SeqRule. We plan to add the support for this kind of rules in the future. Regarding the non deterministic choice, it supports the ChooseRule with a real pseudo non determinism (i.e. there exist two evaluations of the same choose rule, starting from the same state, and producing two 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.

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]))
```

Axiom checker AsmetaS implements an axiom checker, which (optionally) at the end of each transition execution checks if the axioms (if any) expressed over the currently executed ASM specification are satisfied or not. If an axiom is not satisfied, AsmetaS throws an `InvalidAxiomException`, which keeps track of the violated axiom.

Consistent Updates checking The simulator also includes a checker for revealing inconsistent updates; in case of inconsistent updates an `UpdateClashException` is thrown to alarm the user. 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.

Two extension mechanisms AsmetaS can be extended in two ways without modification of its code to customize both the interpretation of undefined static functions and the evaluation of monitored functions. Any extension of the metamodel, for example to include new kinds of terms and rules, would require the modification of the AsmetaS code.

The first 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) put the declaration of the 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 (like plus, ...). 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.

The second 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 needs the definition of a new class implementing the `Environment` 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.

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 if all the constructs are supported, if `NullPointer` exceptions occur and so on).

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 [3] 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.

4.5 Validation

We have validated the AsmetaS code by defining a wide range of JUnit test cases and Fit tables [1]. A Fit table is a simple table written for example in HTML

which specifies some test cases by defining the expected outputs by executing some operations 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. 10 shows the result obtained running a set of test cases (one for each row) while 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()`). Discrepancy of the expected final state and the actual final state are marked in red. Fit tables are available at the ASMETA web site [9].

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 10. Fit Table

5 Related work

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

The Abstract State Machine Language (AsmL) [8] 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" [37].

Adopting a terminology currently used, AsmL is a platform-specific modeling language for the .NET type system. A similar consideration can be made also for the AsmGofer language [32]. 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 [14], which adopts a functional style being developed in ML and which has inspired us in the language of terms, and XASM [36] 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 [15], an extensible execution engine developed in Java, TASM (Timed ASMs) [34], an encoding of Timed Automata in ASMs, and a simulator-model checker for reactive real-time ASMs [33] able to specify and verify First Order Timed Logic (FOTL) properties on ASM models. Among these, the CoreASM engine is the more comparable to ours.

Like our simulator, CoreASM is a general-purpose ASM simulator, is written in Java, and its textual syntax for writing ASM specs 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 runtime 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 multiagent 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

extension points require a certain effort and expertise in Java programming. Our ASMETA framework does not support the extension via plugins; it can be extended in a more classical way only by adding new classes to the metamodel for new concepts, for example for new kinds of terms or rules. The AsmetaS offers two extension mechanisms to customize both the interpretation of undefined static functions and the evaluation of monitored functions, as already explained in Sect. 4.4.

6 Conclusions and future directions

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

The ASMETA metamodelling framework has been developed based on the MDE's metamodelling approach. It is based on a core 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. A concrete textual notation, AsmetaL, and a 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. The developed simulation engine makes ASM models executable and assists, therefore, the modeler in identifying omission and logical errors. The model analyzer is still under development and it will be used to prove whether certain desired properties of the system are true. The integration with the ATGT tester is already available and it can be used to generate a complete test set for the implementation. Finally, a graphical front-end called ASMEE (ASM Eclipse Environment) has been implemented as an Eclipse plug-in to allow editing and manipulation of ASM models within an integrated development environment.

Although ASMETA targets the ASMs, our approach can be applied to any 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 ASMEE IDE. We believe that 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 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 language [5], the Xactium XMF Mosaic [7], to name a few, and *model evolution activities* [24] 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

1. Fit: Framework for integrated test. http://fit.c2.com/.
2. Java Compiler Compiler. https://javacc.dev.java.net/.
3. Log4j. http://logging.apache.org/log4j.
4. Microsoft DSL Tools in Visual Studio 2005. http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/.
5. The AMMA Platform. http://www.sciences.univ-nantes.fr/lina/atl/.
6. The Model Driven Repository for NetBeans. http://mdr.netbeans.org/.
7. The Xactium XMF Mosaic. www.modelbased.net/www.xactium.com/.
8. The ASML Language. research.microsoft.com/foundations/AsmL/.
9. The Abstract State Machine Metamodel website. http://asmeta.sf.net/.
10. ATGT: ASM tests generation tool. http://cs.unibg.it/gargantini/projects/atgt/.
11. Jean Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
12. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
13. Jordi Cabot and Ernest Teniente. Constraint support in MDA tools: A survey. In *ECMDA-FA, Proceedings*, volume 4066 of *LNCS*. Springer, 2006.
14. G. Del 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.
15. The CoreASM Project. http://www.coreasm.org/.
16. D. Hearnden and K. Raymond and J. Steel. Anti-Yacc: MOF-to-text. In *Proc. of EDOC*, pages 200–211, 2002.
17. Sven Efftinge. oAW xText - A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.
18. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich et al., editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
19. A. Gargantini, E. Riccobene, and S. Rinzivillo. 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, 2003.
20. A. Gargantini, E. Riccobene, and P. Scandurra. Deriving a textual notation from a metamodel: an experience on bridging Modelware and Grammarware. In *3M4MDA'06 workshop at the European Conference on MDA*, 2006.
21. Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan, 2006.
22. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. http://www.uml.org/.
23. Stuart Kent. Model driven engineering. In *IFM '02: Proc. of the Third International Conference on Integrated Formal Methods*, pages 286–298. Springer-Verlag, 2002.
24. Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005.
25. Model Integrated Computing (MIC). http://www.isis.vanderbilt.edu/Research/mic.html.

26. Jan Pettersen Nytun, Andreas Prinz, and Merete Skjelten Tveit. Automatic generation of modelling tools. In *Proc. of ECMDA-FA*, pages 268–283, 2006.
27. OCL Environment (OCLE). `http://lci.cs.ubbcluj.ro/ocle`.
28. The Object Managment Group (OMG). `http://www.omg.org`.
29. The Symbolic Analysis Laboratory. `http://sal.csl.sri.com/`.
30. 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.
31. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *J. of Universal Computer Science*, 3(4):377–413, 1997.
32. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.
33. Anatol Slissenko and Pavel Vasilyev. Simulator-model checker for reactive real-time abstract state machines. http://rotor.di.unipi.it/AsmCenter/.
34. Timed Abstract State Machine. `http://esl.mit.edu/html/tasm.html`.
35. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 3(5):689–701, 1997.
36. XASM: The Open Source ASM Language. `http://www.xasm.org`.
37. Y. Gurevich and B. Rossman and W. Schulte. Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .