Ten Reasons to Metamodel ASMs

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

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

Abstract Model-Driven Engineering (or MDE) is an emerging approach for system development which refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. MDE puts emphasis on bridges between different working contexts and on the integration of bodies of knowledge differently developed. We discuss the mutual advantages that the integration of MDE and Abstract State Machines (ASMs) would provide: MDE can gain rigour and preciseness, while ASMs get a standard abstract notation and a general framework for a wide tool interoperability.

Introduction

Model-driven Engineering (MDE) [12] is an emerging approach for software development and analysis where models play the fundamental role of first-class artifacts. *Metamodelling* is a key concept of the MDE paradigm and it 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. Furthermore, metamodelling allows to settle a "global framework" to enable otherwise dissimilar languages (of possibly different domains, the so called *Domain-specific languages*) 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 to establish precise *bridges* (or *projections*) among the metamodels of these different domain-specific languages to automatically execute model transformations.

To achieve the goal of a global interoperability and merging of bodies of knowledge with rigor and preciseness, an integration of the MDE paradigm with formal methods is necessary [23].

This is a position paper mainly aimed at explaining the feasibility and the advantages of the proposed integration in the context of the Abstract State Machines (or ASMs).

We strongly believe that applying the MDE approach to ASMs is worthwhile at least for the following ten reasons, later discussed: (R1) to have a standard abstract notation as unified representation of ASM concepts independent of any particular implementation platform; (R2) to have a graphical representation of

ASMs also useful for teaching purposes; (R3) to have an interchange format among ASM tools; (R4) to have standard libraries and APIs to use in new or existing tools and programs; (R5) to automatically derive a family of languages, visual/textual notations and their parsers; (R6) to allow tool interoperability; (R7) to help the integration of existing tools; (R8) to help the development of new tools; (R9) to export ASMs to other environments and permit the integration with other specialized external notations and tools (for instance for property verification and testing); and last but not least, (R10) to complement the MDE with a formal approach.

The overall goal of our project is to develop a unified abstract notation for ASM and a general framework for a wide interoperability and integration of tools around ASMs. We started by defining the *AsmM*, a metamodel for ASMs, in [28,10]. We have, therefore, developed the ASMETA framework [10] as an instantiation of the metamodelling framework for ASMs, 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.). ASMETA 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.

Each reason stated above may not suffice to justify the effort of developing a complex metamodel as AsmM, but we hope that all together will convince even the most skeptical reader that the application of the MDE approach to ASMs is worthwhile. Note that not only ASMs would benefit from this approach: we expect that new synergies arise and that the cooperative interaction among ASMs and MDE creates an enhanced combined effect - as outlined in R10.

R.1 To have a standard abstract notation

The success of the ASM as a system engineering method able to guide the development of hardware and software systems, from requirements capture to their implementation, is nowadays widely acknowledged [14]. The increasing application of the ASMs for academic and industrial projects has caused a rapid development of tools for ASM model analysis, namely simulation, property verification, and test generation. Among these tools we can cite AsmGofer [43], AsmL [8], Xasm [50], TASM [45], ASM workbench [15,49], CoreASM [18], ATGT [11] and other tools based on model checkers and theorem provers [25,49,26,42,20].

To encode ASM models, each tool uses a different syntax strictly depending on the implementation environment (C for XASM, Gofer for AsmGofer, .NET for AsmL, etc.), adopts a different internal representation of ASM models, and provides proprietary constructs which extend the basic mathematical concepts of ASMs. There is no agreement around a common standard ASM language. The result is that a practitioner willing to use an ASM tool needs to know its own syntax and that 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 not always straightforward and natural, and one needs to map mathematical concepts, like ASM states (namely universes and functions defined on them), 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 tackling the problem of ASM tool interoperability and integration, we exploited the *metamodelling* approach suggested by the MDE.

According to the MDE terminology, a metamodel defines the *abstract syntax* of a language, i.e. the structure of the language, separated from its *concrete notation*. 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 representation* of a formal notation, establishing a common terminology to discriminate pertinent elements to be discussed, and therefore, helps to communicate understandings, especially if – as in the case of ASMs – the formal method is still evolving and the community is too much heterogeneous to easily come to an agreement on an unique textual notation. Note that the goal of achieving a standrad and lean syntax for ASM specifications is shared with the CoreASM project [18].

In [28,10], a complete metamodel for ASMs is presented. As MDE framework, we adopted the OMG's metamodelling platform. The *AsmM* (Abstract State Machines Metamodel) results into class diagrams developed using the MOF (the OMG's metalanguage to define metamodels) modelling constructs (classes, packages, associations). We developed the metamodel 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 is organized in one package called `ASMETA`, which is further divided into four packages as shown in Fig. 1. Each package covers different aspects of ASMs. The dashed gray ovals in Fig. 1 denote the packages representing the notions of *State* and *Transition System*, respectively.

The `Structure` package defines the 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 transition rules derived from

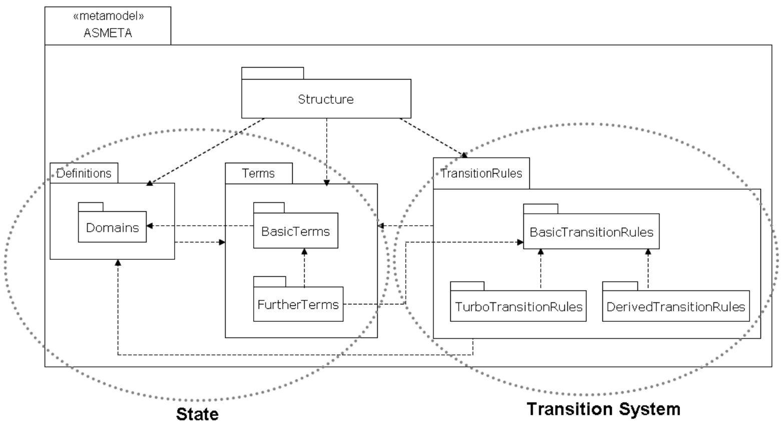

Figure 1. Package structure of the ASM Metamodel

basic and turbo ones (e.g. the case-rule and the while-rule) are contained in the `DerivedTransitionRules` package.

Each class of the metamodel is equipped with a set of relevant *constraints*, OCL (version 2.0 [34]) invariants written to fix how to meaningfully connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams. All OCL constraints have been syntactically checked by using the OCL checker OCLE [36].

In order to make AsmM able to support the languages of existing ASM tools, we have enriched the metamodel with particular forms of domains, special terms and derived rule schemes taken from these languages (see [28] for details). We have borrowed some extended terms including conditional terms and comprehension terms from ASM-SL, maps, sets and sequences from AsmL. Named rules with parameters (`RuleDeclaration`) appear in ASM-SL, while the concepts of submachine computation, iteration, and recursion, modelled in the AsmM respectively by the classes `SeqRule` and `IterateRule`, can be found in XASM as well as in AsmL (with an Object Oriented style, though). The agent representation in the AsmM is similar to the agents of AsmGofer, although Agents is an abstract domain in our metamodel, while Agent is a integer domain in AsmGofer. In Sect. R.9, we clarify how the metamodel is able to capture all all forseeable features of a possible ASM language; therefore, AsmM can be used as standad reference syntax.

R.2 To have a graphical abstract notation

People often claim that formal methods are too difficult to put in practice due to their mathematical-based foundation. In this direction an abstract and visual representation[1], like the one provided by a MOF-compliant metamodel, delivers a more readable view of the modelling primitives offered by a formal method, especially for people, like students, who do not deal well with mathematics but are familiar with the standard MOF/UML. Therefore, the AsmM can be considered a complementary approach to [14] for the presentation of ASMs.

We here give evidence of how the metamodel can be useful to introduce ASMs. We present only a very small fragment of the AsmM whose complete description can be found in [28,10].

Fig. 2 shows the backbone of a *basic ASM*. An instance of the root class `Asm` represents an entire ASM specification. According to the working definition given in [14], 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). Executing a basic ASM means executing its `main rule` starting from one specified *initial state*, i.e. the one denoted by the association end `defaultInitialState`.

The composite relationships between the class `Asm` (the *whole*) and its component classes (the *parts*) assures that each part is included in at most one `Asm` instance.

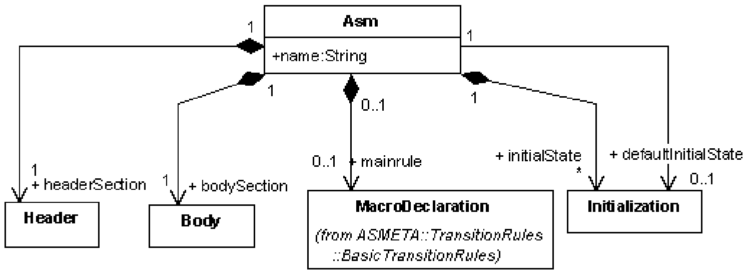

Figure 2. Backbone

An ASM without a main rule and without a set of initial states (see the multiplicity of association ends `mainRule` and `initialState`) is called *module*[2] which is useful to syntactically structure large specifications.

The `Header` (see Fig. 3) consists of some `import clause`*s* and an optional `export clause` to specify the names which are imported from or exported to other ASMs (or ASM modules), and of its `signature` containing the *declarations* of the ASM domains and functions. Every ASM is allowed to use only

[1] It should be noted that the visual representation for the (abstract) syntax of the language has not to be confused with a possible graphical notation for ASM specifications, which we refer to in Sect. R.5.

[2] This definition of module differs slightly from the module concept outlined in Chap. 2 of [14]; but, it has been accorded with the authors.

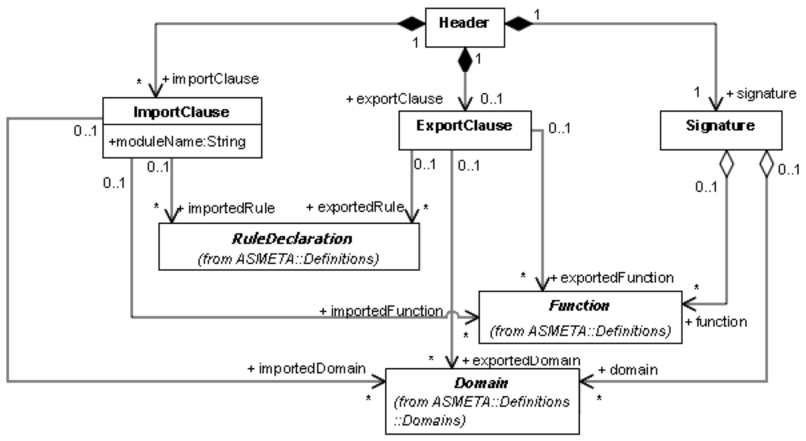

Figure 3. Header

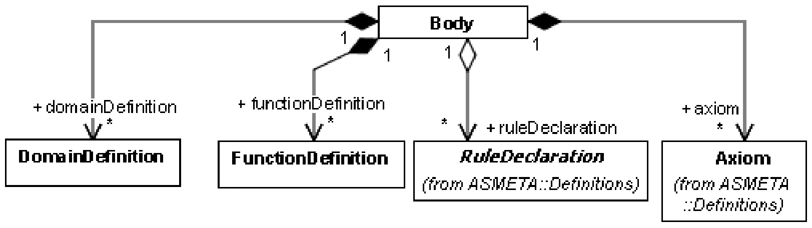

Figure 4. Body

identifiers (for domains, functions and transition rules) which are defined within its signature or imported from other ASMs or ASM modules.

The initialization of an ASM consists of a set of *initial states*. The class `Initialization` (not described here, see [28,10] for details) models the notion of an initial state. All possible initial states are linked (see Fig. 2) to an ASM by the association end `initialState` and one initial state is elected as *default* (see the association end `defaultInitialState`).

The `Body` (see Fig. 4) of an ASM consists of (static) domain and (static/derived) function `definitions` according to domain and function declarations in the signature of the ASM. It also contains `declarations of transition rules` and definitions of `axioms` for constraints one wants to assume for some domains, functions, and transition rules of the ASM.

R.3 To have an interchange format

The interoperability among ASM tools can be (at least partially) achieved by a common interchange format. The work in [7] represents the first and the only attempt in this respect; it was based on the use of a pure XML format and unfortunately it has never been completed.

Whenever a language or formalism is specified in terms of a MOF-compliant metamodel, the MOF enables a standard way to generate an XMI (XML Metadata Interchange) [34] interchange format for models in that language. The main purpose of XMI is to provide an easy interchange of data and metadata between modelling tools and metadata repositories in distributed heterogeneous environments. The XMI format is not for human consumption and it is not to be confused with the "concrete syntax" used by modelers to write their models. It has to be intended, instead, as an effective hard code to be automatically generated for interchanging purposes only.

To tackle the problem of ASM tool interoperability, we exploit the mechanism of deriving a specific XMI format from the metamodel. The ASM-XMI format, given as XML document type definition file (commonly named DTD), has been generated automatically from the AsmM in the MDR framework. First, we have drawn the AsmM with the Poseidon UML tool (v. 4.2) and saved it in the UML-XMI format. Then, we have converted it to the MOF 1.4 XMI by means of the UML2MOF transformation tool provided by the Netbeans MDR project. Finally, we have loaded the MOF model in the MDR framework of Netbeans and according to the rules specified by the MOF 1.4 to XMI 1.2 mapping specification [51], the DTD for AsmM models was generated.

In Section R.6 we discuss the role of the ASM-XMI format for interchanging ASM models among tools.

R.4 To have standard libraries

Applications and tools endowed with MOF-compliant metamodels, can have their *Java Metadata Interface* (JMI) [32] automatically generated. The JMI standard is based on the MOF 1.4 specification and defines a Java application program interface (API) for the creation, storage, access and manipulation of metadata in a MOF-based instance repository.

From the AsmM in the MDR framework we also automatically generate a JMI library for AsmM models (see [28] for more details). JMI can be used in client programs written in Java which want to manipulate ASM models (to read model structure, to modify parts of the specification and create new elements), as well as by tool developers to speed up the creation of new tools supporting ASMs. In Sections R.6, R.7, R.8 we show how the JMI can be useful for tool interoperability, integration of existing tools, and development of new tools.

Besides the XMI and JMI libraries already discussed, other libraries can be developed from MOF-compliant metamodels to provide additional facilities. Among them, we mention CMI (CORBA Metadata Interface) [17] for bridging with the middleware CORBA space.

R.5 To derive concrete notations and their parsers

A MOF-compliant metamodel allows to derive different alternative concrete notations, textual or graphical. Initially, we investigated the use of tools like HUTN (Human Usable Textual Notation) [31] or Anti- Yacc [19] 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. There are better MOF-to-grammar tools now, like xText [22] of OpenArchitectureWare or TCS of AMMA [3], which we may consider to adopt in the future.

In [27] 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 define an EBNF grammar from the AsmM for an ASM textual notation. The resulting language, called AsmetaL[3], is completely independent from any specific platform and allows a natural and straightforward encoding of ASM models. We design AsmM without any specific implementation platform in mind. The language derived from it does not contain any platform-dependent concept. Instead, the language of CoreASM explicitly contains directives for importing plug-ins written in Java, and the AsmL permits the use of the Microsoft .NET library.

In [27], we also provide guidance on how to assemble a JavaCC file given in input to the JavaCC parser generator [2] to automatically produce a parser for the EBNF grammar of the AsmetaL. This parser is more than a grammar checker: it is able to process ASM models written in AsmetaL, to check for their consistency w.r.t. the OCL constraints of the metamodel, and to create instances of the AsmM in a MDR MOF repository through the use of the AsmM-JMIs.

The complete AsmetaL grammar is reported in [28] and is also available in [10] together with the AsmetaL parser.

We have validated the metamodel and the AsmetaL notation to asses their usability and capability to encode ASM models. To this purpose, we have asked to a non ASM expert to port some specifications from [14] and other ASM case studies to AsmetaL. The task was completed within three man-months.

Up to now we have about 140 ASM specifications encoded in AsmetaL and available in [10]. We are strongly confident that AsmetaL satisfies all the desired requirements of expressivity, abstractness and easiness of use.

Note that concrete notations derived from metamodels can be also graphical. For instance, the Eclipse Graphical Modeling Framework (GMF) [4] provides a generative component and runtime infrastructure for developing graphical editors based on Eclipse Modelling Framework (EMF) [1] and the eclipse Graphical Editing Framework (GEF). The GMF follows a novel approach which suggests to derive modelling tools, like graphical model editors, from metamodels [35].

[3] A preliminary version of the AsmetaL language can be found in [41], under the name of AsmM-CS (AsmM Concrete Syntax).

R.6 To allow tool interoperability

The existing ASM tools for model validation and verification, have been developed by encoding an ASM formal model into the language of the implementation environment and exploiting the computation engine and validation/verification algorithms and techniques of the implementation system to compute ASM runs and prove properties. 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, ASM tools are loosely coupled and have syntaxes strictly depending on the implementation environment. This makes the integration of tools hard to accomplish and prevents ASMs from being used in an efficient and tool supported manner during the software development life-cycle. Therefore, a way to support tools interoperabilty is of great interest for the ASM community and can be achieved by the combination of standards like MOF, XMI (R.3), and JMIs (R.4).

Basically, all ASM artifacts/tools can be classified in: *generated*, *based*, and *integrated*. Generated artifacts/tools are derivatives obtained (semi-)automatically by applying to the AsmM metamodel appropriate projections from MOF to the technical spaces Javaware, XMLware, and grammarware. Based artifacts/tools are those developed exploiting the AsmM metamodel and related derivatives. Finally, integrated artifacts/tools are external and existing tools that are connected to the ASM metamodelling environment.

Fig. 5 shows a scenario of interoperability among ASM tools as suggested by our approach. Generated/based tools (like Tool A in the figure) can access ASM models through the APIs (like the AsmM JMIs) in a MOF repository (like the SUN MDR [5]) where ASM models reside. They can also exchange ASM models in the XML/XMI standard format: a XMI reader and writer provided by MDR can be used to load/save an ASM model from/into a XML file.

Integrated tools can interoperate in different ways. Some tools (like Tool B in the figure) can exchange ASM models in the XML/XMI standard format and verify their validity with respect to the given AsmM XMI DTD. Tool providers only need supply their tools with appropriate plug-ins capable of importing and/or exporting the XMI format for the AsmM (by using XMI readers/writers). Other tools (like Tool C in the figure) may keep their input data formats: in this case walkers must be developed to translate ASM models from the repository to the tool proprietary formats. Mixed approaches are also possible, as the one adopted in modifying the ATGT tool, as explained in Section R.7.

A modeler can also start writing her/his ASM specification in AsmetaL and then, through the connection to the repository provided by the parser, transform it, for example, into the XMI interchange format.

R.7 To help the integration of existing tools

We here discuss how we modified the ATGT tool [11] in order to make it AsmM-compliant. ATGT takes an ASM specification (written using the AsmGofer syn-

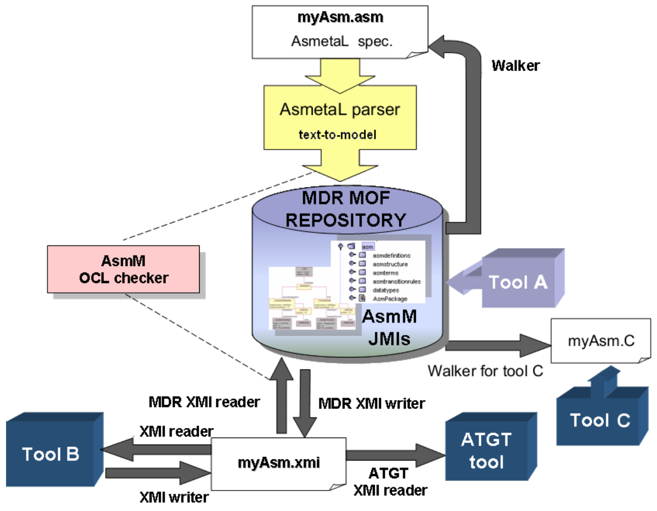

Figure 5. ASM model interchange through XMI and APIs

tax) and produces a set of test predicates, translates the original ASM specification to Promela (the language of the model checker SPIN used to generate tests), and generates a set of test sequences by exploiting the counter example generation of the model checker.

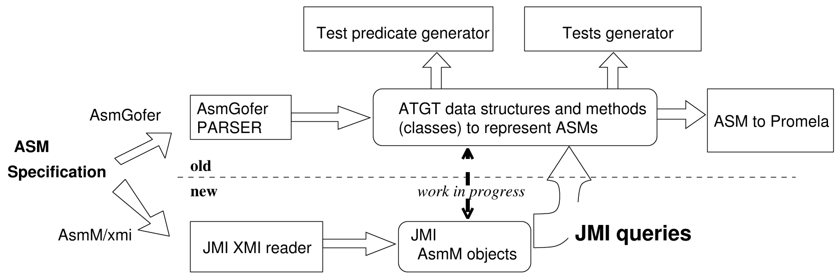

Figure 6. Adapting ATGT to the AsmM

ATGT is written in Java. It already (see Fig. 6) has its own parser for AsmGofer files, which reads a specification and builds an internal representation of the model in terms of Java objects. The tool functionalities are delegated to three components (Test predicate generator, Tests generator, ASM to Promela) which read the data of the loaded ASM specification and perform their tasks.

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

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

Although we did not exploit the power of the metamodel inside ATGT and we simply made ATGT AsmM-compliant, the result is worthwhile and the effort is limited: adding this new feature to ATGT required about two man-months. If we started today from scratch to develop ATGT, we would use directly JMI to represent ASM models, since JMI offers a stable and clean interface that is derived from the metamodel (see Sect. R.4). The use of JMI would avoid the burden of writing internal libraries for representing ASM models. For this reason, we have started working on making the internal representation of ASM models that ATGT adopts equivalent to the JMI, in order to eventually integrate JMI directly in ATGT (work in progress in Fig. 6).

Further advances in the MDE direction [13] would be replacing the ASM to Promela and the AsmGofer parser components by model transformations from the AsmM (as *pivot metamodel*, see Sect. R.9) to Promela metamodel and from Gofer metamodel to the AsmM, provided that such metamodels for Promela and Gofer (linked to their concrete syntax) exist.

R.8 To help the development of new tools

MDE helps developers to build new tools by providing an interchange format (R.3), standard libraries (R.4) and several possible maps to concrete syntaxes and parsers (R.5). By exploiting these technologies, a developer who is interested in developing a new tool for ASMs, does not need to write a parser, an internal representation of ASMs and an interchange format (if he/she wishes to export, import files from other tools). In particular, the development of a grammar can be very time consuming and error prone - specially if one wants to be able to read complete ASM specifications. Internal representations of ASMs are normally bound to the parser which is being defined, and a small change in the parser may require an update of the internal libraries and refactoring of the code. All these tasks can require a good deal of time and effort, although they are not relevant for the particular technique or algorithm being developed. In the MDE approach, the developer needs to understand the metamodel (for example by reading its graphical representation - R.2) and then focus on the functionalities he/she intend to support with the new tool.

For instance, we have developed a general-purpose ASM *simulation engine* [29,10], called AsmetaS, to make AsmM models executable. This tool is an example of Tool A (see Fig. 5) since essentially it is an interpreter which navigates through the MOF repository where ASM models are instantiated (as instances of the AsmM metamodel) to make its computations. We do not have to deal with basic functionalities such as parsing, abstract syntax trees, type checking, etc., since they are already provided by the MOF-environment. We have focused the development on those classes necessary to simulate an ASM, and the construction of the update set has required only the definition of the class UpdateSet representing an update set and the class UpdateSetBuilder building an update set. UpdateSetBuilder introduces a method UpdateSet m(R r) which, for every class R representing a rule, builds the update set for the rule r of class R.

The architecture of this interpreter is very simple and consists in only 20 classes. It also allows a modular and incremental development. A first prototype (available at [10]) has been developed in only three man months and is able to interpret basic, turbo without submachine calls, and synchronous multiagent ASMs.

R.9 To integrate ASMs with other notations/tools

In the MDE direction, AsmM can be seen as the *pivot metamodel* toward a systematic integration among ASM tools and between ASMs and external tools.

According to the view presented in [13], a pivot metamodel of a given formalism or language L is intended as a platform-independent modelling language which abstracts a certain number of general concepts about L. The integration among tools supporting L can be achieved by providing, for the notation L' (a dialect of L) of each tool T_L, a *metamodel* – seen as a platform-specific modelling language – and *model transformations* to the pivot and from the pivot to the L'-metamodel. Hence, the metamodel of the notation L^i of a tool T_L^i can be linked to the metamodel of the notation L^j of another tool T_L^j by the composition of the two transformations from L^i-metamodel to the pivot and from the pivot to the L^j-metamodel. In this way, the interoperability between tools T_L^i and T_L^j is achieved by translating PSM (Platform-specific Model) models written in L^i to L^j and vice versa.

In the ASM context, AsmM can be adopted as pivot metamodel and would allow the integration among ASM tools at the level of metamodels. For example, if we take AsmGofer as tool T_L^i and AsmL as tool T_L^j and we had defined the corresponding metamodels together with precise *transformation bridges* from/to AsmM, we may map an AsmGofer-PSM into an AsmL-PSM.

Similarly, one can integrate the language L or one of its tools T_L with a tool using a notation M by providing a bridge between the pivot metamodel of L to the metamodel of M. In the ASM context, the AsmM may allow the integration between ASMs and tools like the model checkers Spin or SMV, provided that the metamodels for their notations exist.

For most notations M, however, the metamodel does not exists, and M is simply pure text. In this case the bridge must be built between the metamodel of L and a textual notation, and this can be done by using MOF-to-grammar tools, like xText [22] of OpenArchitectureWare or TCS of AMMA [3]. In the ASM context, we may "compile" ASM models into a programming language, like Java, by applying a AsmM-to-Java transformation to the input ASM model.

R.10 To complement the MDE with a formal approach

In the previous sections, we have discussed some advantages that ASMs can gain from MDE. We believe that the MDE paradigm can also gain rigor and preciseness from the integration with ASMs as formal method. The semantics specification of domain-specific modeling languages (defined in terms of a metamodel), for example, is an open problem in the MDE approach. The OMG metamodelling framework provides, by means of *metamodels* and *UML profiles* (UML metamodel extensions for a particular application domain), standard techniques to define the abstract syntax and static semantics (the OCL constraints) of a Domain-specific language. However, it lacks of any standard and rigorous support to provide the dynamic (operational) semantics, which is usually given in natural language. This lack has several negative consequences, as confirmed by existing work in literature which aims at formalizing the UML semantics.

Techniques and approaches to the precise and pragmatic definition of behavioral semantics for domain-specific languages are still under development. One promising method, called *semantic anchoring* relies on the use of well-defined *semantic units* of simple, well-understood constructs (like a finite state machine) and on the use of model transformations that map higher level modeling constructs into configured semantic units. This approach has been followed, for example, by the authors in [16,46], where AsmL is used as a common semantic framework to define the semantic domain of Domain-specific languages.

We believe that any formalism proposed as semantic framework must address the following important characteristics: (i) it should be formal and powerful enough to rigorously define the operational semantics of complex real languages, (ii) it should be executable in order to validate the metamodels' semantics, (iii) it should be endowed with a metamodel-based definition conforming to the metamodelling framework in order to allow the applicability of model transformation tools, and (iv) it should be able to work at high levels of abstraction. According to these requirements, the ASM formalism seems to be a good candidate.

Similarly to the approach in [16], we propose ASMs as semantic framework to define the (operational) semantics of metamodel-based languages. The key idea is a smooth integration of the AsmM metamodel with the OMG framework in order to provide a means to rigorously define the operational semantics of metamodel-based languages and, in particular, of UML extensions (profiles), in a way which permits us to uniformly link abstract syntax, expressed in the MOF metalanguage, and detailed semantics, expressed in ASMs (here promoted as metalanguage, too) of languages.

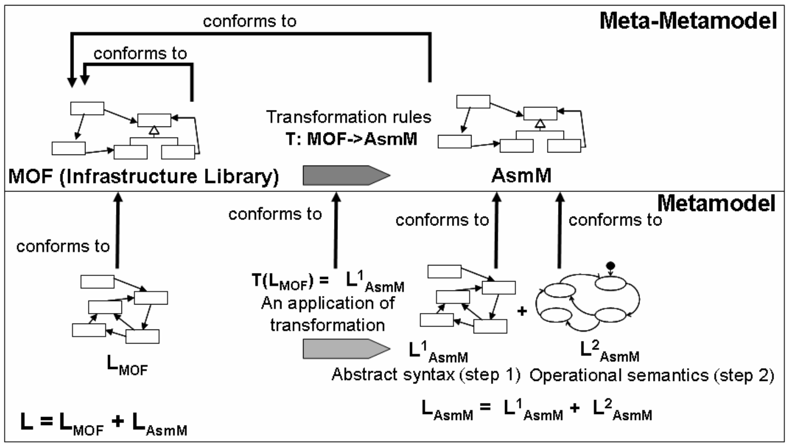

Figure 7. An integrated framework for metamodel-based language specification

In practice, this integration may be done as shown in Fig. 7. At the meta-metamodel level, the MOF core constructs i.e., the Infrastructure Library, have to be mapped into ASM concepts. This may be done by defining a set of transformation rules, T_{MOFToASM}, from the Infrastructure Library metamodel to the AsmM metamodel.

At the metamodel level, a metamodel or a UML profile L_{MOF} of a given language L is translated by T_{MOFToASM} to a ground ASM-compliant metamodel L^1_{AsmM} of L made of multi-sorted first-order structures, i.e. sets with relations, functions and constraints, representing classes and associations of the source L_{MOF} metamodel. L^1_{AsmM} needs to be complemented with the semantic aspects of the language L. The *computational model* which reflects the operational semantics of L, say L^2_{AsmM}, is defined through ASM transition rules. The static structures of the ASM signature L^1_{AsmM} is further refined and enriched with dynamic aspects, e.g., designating some specific entities to be ASM agents, and introducing new functions, which, however, may be present in the original metamodel expressed in terms of OCL query/operations. We say $L_{\text{AsmM}} = L^1_{\text{AsmM}} + L^2_{\text{AsmM}}$ the final result of this modelling activity.

Note that, the process of applying the T_{MOFToASM} can be fully automatized by means of a *transformation engine* like the ATL in the AMMA platform [3], Xactium XMF Mosaic [6], etc. However, a certain human effort is still required to capture in terms of ASM transition rules the behavioural aspects of the given language.

We have applied the proposed methodology to a UML profile for the SystemC language [39] - as part of the definition of a model-based SoC (System-on-chip) design flow for embedded systems [21,40] - to define the operational semantics of the SystemC Process State Machines, an extension of the UML statecharts used to model the reactive behaviour of the SystemC processes.

Although the proposed approach has been first identified and tested for the OMG's framework, it could be easily extended and applied to other metamodelling frameworks.

Related work

Concerning the definition of a concrete language for ASMs, other previous proposals exist. 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 ASMs, 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" [52]. 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 [43]. 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 [15], which adopts a functional style being developed in ML and which has inspired us in the language of terms, and XASM [50] 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 [18], an extensible execution engine developed in Java, and TASM (Timed ASMs) [45], an encoding of Timed Automata in ASMs.

Concerning the metamodeling technique for language engineering, we can mention the official metamodels supported by the OMG [37] for MOF itself, for UML [47], for OCL, etc. Academic communities like the Graph Transformation community [30,44,48] and the Petri Net community [38], have also started to settle their tools on general metamodels and XML-based formats. A metamodel for the ITU language SDL-2000 has been also developed [24]. Recently, a metamodel for the AsmL language is available in the XMI format at [9] as part of a zoo of metamodels defined by using the KM3 meta-language [33]. However, this metamodel is not appropriately documented or described elsewhere, so this prevent us to evaluate it for our purposes.

References

1. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.
2. Java Compiler Compiler. `https://javacc.dev.java.net/`.
3. The AMMA Platform. `http://www.sciences.univ-nantes.fr/lina/atl/`.
4. The Eclipse Graphical Modeling Framework. `http://www.eclipse.org/gmf/`.
5. The MDR (Model Driven Repository) for NetBeans. `http://mdr.netbeans.org/`.

6. The Xactium XMF Mosaic. `http://www.modelbased.net/www.xactium.com/`.
7. M. Anlauff, G. Del Castillo, J. Huggins, J. Janneck, J. Schmid, and W. Schulte. The ASM-Interchange Format XML Document Type Definition (ASM-DTD). `http://www.first.gmd.de/~ma/asmdtd.html`.
8. The ASML Language. `http://research.microsoft.com/foundations/AsmL/`.
9. The AsmL metamodel in the Atlantic Zoo. `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/#AsmL`, 2006.
10. The Abstract State Machine Metamodel and its tool set. `http://asmeta.sf.net/`.
11. ATGT: ASM tests generation tool. `http://cs.unibg.it/gargantini/project/atgt/`.
12. J. Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
13. J. Bézivin, H. Bruneliére, F. J. Jouault, and I. Kurtev. Model Engineering Support for Tool Interoperability. In *The 4th Workshop in Software Model Engineering (WiSME'05)*, Montego Bay, Jamaica, 2005.
14. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.
15. G. D. Castillo. The ASM Workbench - A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 578–581. Springer, 2001.
16. K. Chen, J. Sztipanovits, S. Abdelwahed, and E. K. Jackson. Semantic anchoring with model transformations. In *ECMDA-FA*, pages 115–129, 2005.
17. OMG, CORBA. `http://www.corba.org/`.
18. The CoreASM Project. `http://www.coreasm.org/`.
19. D. Hearnden and K. Raymond and J. Steel. Anti-Yacc: MOF-to-text. In *Proc. of EDOC*, pages 200–211, 2002.
20. A. Dold. A Formal Representation of Abstract State Machines Using PVS. Verifix Technical Report Ulm/6.2, Universitat Ulm, July 1998.
21. E. Riccobene and P. Scandurra and A. Rosti and S. Bocchio. A SoC Design Methodology Based on a UML 2.0 Profile for SystemC. In *Proc. of Design Automation and Test in Europe (DATE'05)*. IEEE, 2005.
22. S. Efftinge. oAW xText - A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.
23. T. M. et al. Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005.
24. J. Fischer, M. Piefel, and M. Scheidgen. A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In *Fourth SDL And MSC Workshop (SAM'04)*, pages 208–223, 2004.
25. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. G. et al., editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
26. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to Generate Tests from ASM Specifications. In E. Böger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer, 2003.
27. 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.
28. A. Gargantini, E. Riccobene, and P. Scandurra. Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Technical Report 97, DTI Dept., University of Milan, 2006.

29. A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Simulator for ASMs. In *14th International ASM Workshop*, Grimstad, Norway, June 7 – 9, 2007.
30. R. Holt, A. Schürr, S. E. Sim, and A. Winter. Graph eXchange Language. `http://www.gupro.de/GXL/index.html`.
31. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. `http://www.uml.org/`.
32. Java Metadata Interface Specification, Version 1.0. `http://java.sun.com/products/jmi/`, 2002.
33. F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 2006.
34. OMG. The Model Driven Architecture (MDA). `http://www.omg.org/mda/`.
35. J. P. Nytun, A. Prinz, and M. S. Tveit. Automatic generation of modelling tools. In *Proc. of ECMDA-FA*, pages 268–283, 2006.
36. OCL Environment (OCLE). `http://lci.cs.ubbcluj.ro/ocle`.
37. The Object Managment Group (OMG). `http://www.omg.org`.
38. Petri Net Markup Laguage (PNML). `http://www.informatik.hu-berlin.de/top/pnml`.
39. E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A UML 2.0 profile for SystemC: toward high-level SoC design. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 138–141. ACM, 2005.
40. E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *Proc. of the 43rd annual Conference on Design Automation (DAC'06)*, pages 915–918, New York, NY, USA, 2006. ACM.
41. 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.
42. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.
43. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.
44. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *J. Padberg (Ed.), UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS*, 2001.
45. The Timed Abstract State Machine (TASM) Language and Toolset. `http://esl.mit.edu/html/tasm.html`.
46. R. Thibodeaux. The Specification of Architectural Languages with Abstract State Machines. In *14th International ASM Workshop*, Grimstad, Norway, June 7 – 9, 2007.
47. OMG. The Unified Modeling Language (UML). `http://www.uml.org`.
48. D. Varró, G. Varró, and A. Pataricza. Towards an XMI–based model interchange format for graph transformation systems. Technical report, Budapest University of Technology and Economics, Dept. of Measurement and Information Systems, September 2000.
49. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science (J.UCS)*, 3(5):689–701, 1997.
50. XASM: The Open Source ASM Language. `http://www.xasm.org`.
51. OMG, XMI Specification, v1.2. `http://www.omg.org/cgi-bin/doc?formal/2002-01-01`.
52. Y. Gurevich and B. Rossman and W. Schulte. Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .