Integrating Formal Methods with Model-driven Engineering

Angelo Gargantini
Università di Bergamo, Italy
angelo.gargantini@unibg.it

Elvinia Riccobene
Università degli Studi di Milano, Italy
elvinia.riccobene@dti.unimi.it

Patrizia Scandurra
Università di Bergamo, Italy
patrizia.scandurra@unibg.it

Abstract

In this paper, we present our position and experience on integrating formal methods *with the* Model-driven Engineering *(MDE) approach to software development. Both these two approaches have advantages and disadvantages, and we here show how the advantages of one can be exploited to cover or weaken the disadvantages of the other. We also propose an* in-the-loop *integration which allows the development of a general framework for software engineering where rigorousness and preciseness of formal methods are combined with flexibility and automation of the MDE. We discuss the feasibility of unifying these two separate worlds, referring to our experience on integrating the Abstract State Machine formal method with the Eclipse Modeling Framework supporting MDE facilities.*

1. Introduction

It is nowadays widely acknowledged that the use of *Formal Methods* (FMs), based on rigorous mathematical foundations, is essential for system development, especially for high-integrity systems where safety or security are important. On the other hand, the *Model-driven Engineering* (MDE) [1], [2] is emerging as a new paradigm in software engineering, which bases system development on (meta-)modeling and model transformations, and provides methods to build bridges between similar or different technical spaces and domains Both these two approaches have advantages and disadvantages.

In this paper, we discuss how these two approaches can be combined showing how the advantages of one can be exploited to cover or weaken the disadvantages of the other. We refer to our experience in integrating the Abstract State Machine (ASM) formal method [3] with the EMF (Eclipse Modeling Framework) (as framework for MDE). The effort of this work has been up to now twofold worth: ASMs used to provide semantics to languages defined in the MDE context [4], and MDE used in building and integrating tools around ASMs [5], [6]. Here, we also propose an *in-the-loop* integration that allows the development of a general framework for software engineering where rigorousness and preciseness of FMs (in our case, ASMs) are combined with flexibility and automation of the MDE.

The remainder of this paper is organized as follows. Sect. 2 presents our integration envisioning by suggesting how MDE methodologies and technologies can be combined with FMs. Sect. 3 provides basic concepts concerning ASMs. Sections 4 and 5 show a concrete scenario of in-the-loop integration between the ASM formal method and the EMF framework. Sect. 6 sketches some related work. Finally, our conclusion and future directions are provided in Sect. 7.

2. Integration Envisioning

Fig. 1 briefly summaries advantages and disadvantages of the MDE and FMs.

Advantages of FMs. The use of formal methods in system engineering is becoming essential, especially during the early phases of the development process. Indeed, an abstract model of the system can be used to understand if the system under development satisfies the given requirements (by simulation and model-based testing), and guarantees certain properties by formal analysis (validation & verification).

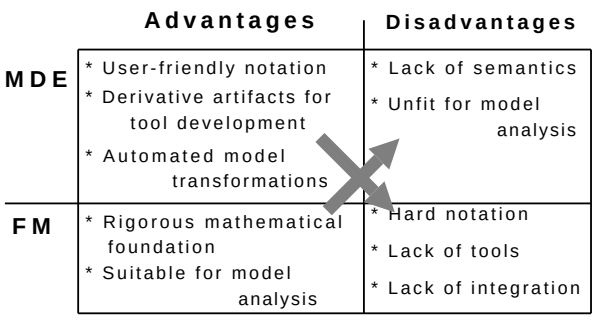

Figure 1. Formal methods and MDE

Disadvantages of FMs. While there are several cases proving the applicability of formal methods in industrial applications and showing very good results, many practitioners are, however, still reluctant to adopt formal methods. Besides the well-known lack of training, this skepticism is mainly due to: the complex notations that formal techniques use rather than other lightweight and more intuitive graphical notations, like the Unified Modeling Language (UML); the lack of easy-to-use tools supporting a developer during the life cycle activities of the system development, possibly in a seamless manner; and the lack of integration among formal methods themselves and their associated tools.

Advantages of MDE. MDE technologies with a greater focus on architecture and automation yield higher levels of abstraction in system development by promoting models as first-class artifacts to maintain, analyze, simulate, and eventually reduce into code or transformed into other models. Meta-modeling is a key concept of the MDE paradigm and it is intended as a way to endow a language or a formalism with an abstract notation, so separating the abstract syntax and semantics of the language from its different concrete notations. Metamodel-based modeling languages are increasingly being defined and adopted for specific domains of interest addressing the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [2].

Disadvantages of MDE. Although the definition of a language abstract syntax by a metamodel is well mastered and supported by many meta-modeling environments (EMF/Ecore, GME/MetaGME, AMMA/KM3, XMF-Mosaic/Xcore, etc.), the semantics definition of this class of languages is an open and crucial issue. Currently, meta-modeling environments are able to cope well with most syntactic and transformation definition issues, but they lack of any standard and rigorous support to provide the (possibly executable) semantics of metamodels, which is usually given in natural language. This implies that most currently adopted metamodel-based languages are not yet suitable for effective model analysis due to their lack of a strong semantics necessary for a formal model analysis assisted by tools.

The lack of user-friendly notations, of integration of techniques, and of their tool inter-operability, is still a significant challenge for formal methods that can be achieved by exploiting the metamodeling approach suggested by the MDE. Sect. 2.1 briefly introduces a process for language engineering which starts by defining an abstract notation for a FM in terms of a metamodel, and then to build a general framework for tools development and integration around the FM.

On the other hand, the problem of providing a way to express the semantics of metamodel-based languages and to perform model validation and verification can be solved by the use of FMs. Sect. 2.2 presents an approach to endow language metamodels with precise (and possibly executable) semantics, and to associate formal models, suitable for model analysis, to language terminal models by automatic model mapping.

2.1. MDE for FMs

Applying the MDE development principles to a formal method should have the following overall goal: (a) to provide an intuitive modeling notation having rigorous syntax and semantics, possibly supporting a graphical view of the model; (b) to allow modeling techniques which facilitate the use of FMs in many stages of the development process, 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) to support an open and flexible architecture to make easier the development of new tools and the integration with other existing tools.

In practice, this activity consists mainly of

- designing the formal language by metamodeling (i.e. building a metamodel of the formal notation),
- defining language concrete syntaxes, i.e. metamodel derivatives (also called *language artifacts*), to handle (i.e. create, storage, control, exchange, access, manipulate) language models, and
- developing processing tools by exploiting the chosen metamodeling framework and the language artifacts able to process and analyze such models.

In principle, the choice of a specific meta-modeling framework should not prevent the use of models in other different meta-modeling spaces, since model transformations among meta-modeling framework should be theoretically supported by the environments. However, although in theory one could switch framework later, a commitment with a precise metamodeling framework is better done at the very early stage of the development process, mainly for practical reasons. The chosen MDE framework should support easy (e.g. graphical) editing of (meta) models, model to model transformations, and text to model and model to texts mappings to assist the developing a concrete notations in textual form. It should also possibly provide a mapping to a programming language (i.e. API artifacts) to allow the integration in programs and software applications.

2.2. FMs for MDE

Applying a formal method to a language L defined in a meta-modeling framework should have the following overall goal: (a) allow the definition of the behaviors (semantics) of models conforming to L and (b) provide several techniques and methods for the formal analysis of such models (e.g. validation, property proving, model checking, etc,).

A metamodel-based language L has a well-defined semantics if a semantic domain S is identified and a semantic mapping $M_S : A \rightarrow S$ is provided [7] between the L's abstract syntax A (i.e. the metamodel of L) and S to give meaning to syntactic concepts of L in terms of the semantic domain elements.

The semantic domain S and the mapping M_S can be described in varying degrees of formality, from natural language to rigorous mathematics. It is very important that both S and M_S are defined in a precise, clear, and readable way. The semantic domain *S* is usually defined in some formal, mathematical framework (transition systems, pomsets, traces, the set of natural numbers with its underlying properties, are examples of semantic domains). The semantic mapping M_S is not so often given in a formal and precise way, possibly leaving some doubts about the semantics of L. Thus, a precise and formal approach to define it is desirable.

Sometimes, in order to give the semantics of a language L, another helper language L', whose semantics is clearly defined and well established, is introduced. Therefore, M'_S and S' should be already well-defined for L'. L' can be exploited to define the semantics of L by (1) taking S' as semantic domain for L too, i.e. $S = S'$, (2) by introducing a *building function* $M : A \rightarrow A'$, being A' the abstract syntax of L', which associates an element of A' to every construct of A, and (3) by defining the semantic mapping $M_S : A \rightarrow S$ as $M_S = M'_S \circ M$.
In this way, the semantics of L is given by translating terminal models m of A to models of L'.

The M function *hooks* the semantics of *A* to the S' semantic domain of the language L'. The complexity of this approach depends on the complexity of building the function M.

To be a good candidate, a language L' should (i) be abstract and formal to rigorously define model behavior at different levels of abstraction, but without formal overkill; (ii) be able to capture heterogenius models of computation (MoC) in order to smoothly integrate different behavioral models; (iii) be executable to support model validation; (iv) be endowed with a model refinement mechanism leading to correct-by-construction system artifacts; and (v) be supported by a set of tools for model simulation, testing, and verification.

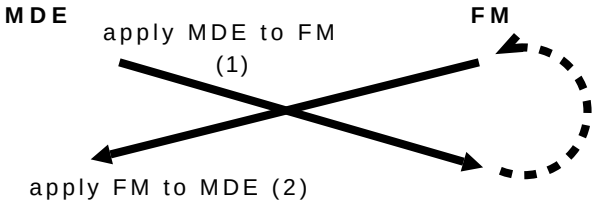

Figure 2. In the loop integration of FM and MDE

2.3. In-the-loop integration

Although the two activities of applying the MDE to a FM and apply a FM to the MDE can be considered unrelated and could be performed in parallel even by using two different notations for the MDE and FMs, the best results can be obtained by a tight integration between the MDE and a FM in a *in-the-loop* integration approach. In this approach the MDE technology and the FM notation are unique in both of the above activities and the application of the MDE to the FM is carried on before the application of the FM to the MDE. Thanks to the first activity, the FM will be endowed with a metamodel and possibly a set of tools (e.g. a grammar, artifacts, etc.) which can be used in the second activity to automatize (meta-)model transformations and apply suitable tools for formal analysis (i.e. validation and verification) of models. Indeed, although for applying FM to the MDE it is in principle not required that the FM is provided with a metamodel (see Sect. 2.2), a formal notation endowed with a representation of its concepts in terms of a metamodel would allow to exploit MDE transformation languages (as ATL) to define the building function M and to automatize the application of M as model transformation by means of a transformation engine. Therefore, having a metamodel is a further constraint for an helper language L', and it justifies why the second activity must precede the first one.

Sect. 4 and 5 present our instantiation of the *in-the-loop* integration with the EMF (Eclipse Modeling Framework) as MDE technologies supporting framework and the ASMs (Abstract State Machines) as formal method. This choice is justified by the following motivations:

- EMF is based on an open-source Eclipse framework and unifies the three most important technologies, i.e. Java, XML, and UML, currently used for software development.

- ASMs own all the characteristics of preciseness, abstraction, refinements, executability, metamodel-based definition that we identified as the desirable properties a FM should have in order to be a good candidate for integration.

3. Abstract State Machines

Abstract State Machines are an extension of Finite State Machines, where unstructured "internal" control states are replaced by states comprising arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates defined on them. The *transition relation* is specified by "rules" describing the modification of the functions from one state to the next. Basically, a transition rule has the form of *guarded update* "**if** *Cond* **then** *Updates*", where *Updates* are a set of function updates of the form $f(t_1, \ldots, t_n) := t$ and are simultaneously executed[1] when *Cond* is true.

The notion of ASMs moves from a definition which formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, and in a structured and recursive way, *Structured or Turbo ASMs*, to a generalization where multiple agents interact *Multi-agent ASMs*. Appropriate rule constructors also allow non-determinism (existential quantification) and unrestricted synchronous parallelism (universal quantification).

A complete mathematical definition of the ASM method can be found in [3], together with a presentation of the great variety of its application in different fields like: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, verification of compilation schema and compiler back-ends, etc.

4. EMF for ASMs

We started by defining a metamodel [8], [9], the Abstract State Machine Metamodel (AsmM), as abstract syntax description of a language for ASMs. From the AsmM, by exploiting the MDE approach and its facilities (derivative artifacts, APIs, transformation libraries, etc.), we obtained in a generative manner (i.e. semi-automatically) several artifacts (an interchange format, APIs, etc..) for the creation, storage, interchange, access and manipulation of ASM models [6]. The AsmM and the combination of these language artifacts lead to an instantiation of the EMF metamodeling framework for the ASM application domain, the ASM mETA-modeling framework (ASMETA) that provides a global infrastructure for the interoperability of ASM tools (new and existing ones) [9], [5].

1. f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule to a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update the function f to t on parameters $t_1, \ldots, t_n$. This produces another state S_{i+1} which differs from S_i in the new interpretation of f.

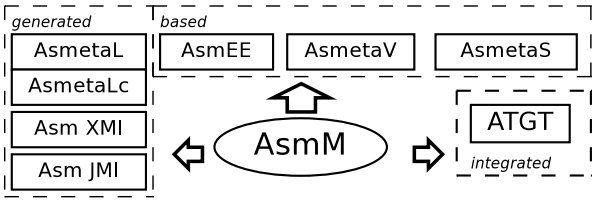

Figure 3. The ASMETA tool set

The ASMETA tool set (see Fig. 3) includes (among other things) a textual concrete syntax, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints; a simulator, *AsmetaS*, to execute ASM models; the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; the *ATGT* tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end called *ASMEE* (ASM Eclipse Environment) which acts as IDE and it is 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 Ecore 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 AsmetaS). Integrated artifacts/tools are external and existing tools that are connected to the ASMETA environment.

5. ASMs for EMF

We here describe how the ASM formal method can be exploited as helper language to define a formal *semantic framework* to provide languages with their (possible *executable*) semantics natively with their metamodels.

Recall, from Sect. 2.2, that the problem of giving the semantics of a metamodel-based language L is reduced to define the function $M : A \rightarrow A'$, being A and

A' the language and the helper language abstract syntaxes, respectively. Let us assume the ASMs as helper language satisfying the requirements, given in Sect. 2.2, of having a mathematical well-founded semantics and a metamodel-based representation. The semantic domain S_{AsmM} is the first-order logic extended with the logic for function updates and for transition rule constructors defined in [3] and the *semantic mapping* $M_S : AsmM \rightarrow S_{AsmM}$ to relate syntactic concepts to those of the semantic domain is given in [6].

Exploiting the ASMs, the semantics of a metamodel-based language is expressed in terms of ASM transition rules by providing the building function $M : A \longrightarrow AsmM$. As already mentioned above, the definition of the function M may be accomplished by different techniques (see [4]), which differ in the way a terminal model is mapped into an ASM. As example of such techniques, the *semantic hooking* technique is presented below. A concrete example is also provided by applying the described technique to a possible metamodel for the simple Petri net formalism. The results of this activity are executable semantic models for Petri nets which can be make available in a model repository either in textual form using AsmetaL or also in abstract form as instance model of the AsmM metamodel.

Even the reader could have expected the case study of a domain specific language (DSL) metamodel, we preferred to report here an example of metamodel whose semantics is well-know, as, indeed, that of the Petri Nets. In our opinion, this example facilities the reader understanding of our approach since (s)he can concentrate on acknowledging the ASM capabilities of being a very abstract language able to express the semantics, rather then concentrating in understanding the semantics of a specific, possibly unknown DSL whose semantics would have had to be presented in natural language.

Semantic hooking. The semantic hooking endows a language metamodel *A* with a semantics by means of a unique ASM for any model conforming to A. By using this technique, designers *hook* to the language metamodel *A* an abstract state machine Γ_A, which is an instance of ***AsmM*** and contains all data structures modeling elements of A with their relationships, and all transition rules representing behavioural aspects of the language. Γ_A does not contain the initialization of functions and domains, which will depend on the particular instance of *A*. The function which adds the initialization part is called ι. Formally, the building function M is given by $M(m) = \iota_A(\Gamma_A, m)$, for all m conforming to A.

Γ_A: $AsmM$, is an abstract state machine which

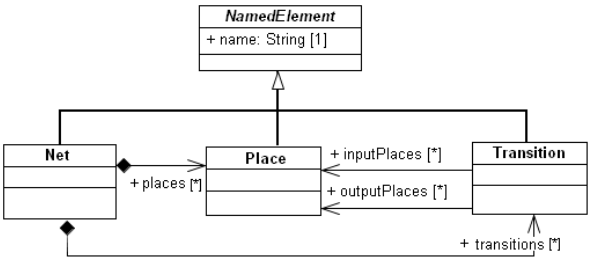

Figure 4. A metamodel for basic Petri nets

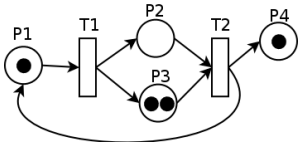

Figure 5. A basic Petri net with its initial marking

contains only declarations of functions and domains (the signature) and the behavioral semantics of L in terms of ASM transition rules.

ι_A: $AsmM \times A \longrightarrow AsmM$, properly initializes the machine. ι_A is defined on an ASM a and a terminal model m instance of A; it navigates m and sets the initial values for the functions and the initial elements in the domains declared in the signature of a. The ι_A function is applied to Γ_A and to the terminal model m for which it yields the final ASM.

5.1. Basic Petri Nets semantics

Fig. 4 shows the metamodel for the basic Petri net formalism. It describes the static structure of a net consisting of places and transitions (the two classes `Place` and `Transition`), and of directed arcs (represented in terms of associations between the classes `Place` and `Transition`) from a place to a transition, or from a transition to a place. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. Places may contain (see the attribute `tokens` of the `Place` class) any non-negative number of tokens, i.e. infinite capacity. Moreover, arcs are assumed to have a unary weight. Fig. 5 shows (using a graphical concrete syntax) an example of Petri net (with its initial marking) that can be intended as instance (a terminal model) of the Petri net metamodel in Fig 4.

According to the semantic hooking approach, first we have to specify an ASM Γ_{PT} (i.e. a model conforming to the AsmM metamodel) containing only dec-

larations of functions and domains (the signature) and the behavioral semantics of the Petri net metamodel in terms of ASM transition rules. Listing 1 reports a possible Γ_{PT} in AsmetaL notation. It introduces abstract domains for the nets themselves, transitions, and places. The static function *isEnabled* is a predicate denoting whether a transition is enabled or not. The behavior of a generic Petri net is provided by two rules: *r_fire*, which express the semantics of token updates upon firing of transitions, and *r_PetriNetReact*, which formalizes the firing of a non-deterministic subset of all enabled transitions. The main rule executes all nets in the *Net* set.

Listing 1. Γ_{PT}

```
asm PT_hooking
signature:
 abstract domain Net
 abstract domain Place
 abstract domain Transition

 //Functions on Net
   controlled places: Net −> Powerset(Place)
   controlled transitions: Net −> Powerset(Transition)

 //Functions on Place
   controlled tokens : Place −> Integer

 //Functions on Transition
   controlled inputPlaces: Transition −> Powerset(Places)
   controlled outputPlaces: Transition −> Powerset(Places)
   static isEnabled : Transition −> Boolean

definitions:
  function isEnabled ($t in Transition) =
            (forall $p in inputPlaces($t) with tokens($p)>0)

  rule r_fire($t in Transition) =
    seq
      forall $i in inputPlaces($t) do tokens($i) := tokens($i)−1
      forall $o in outputPlaces($t) do tokens($o) := tokens($o)+1
    endseq

  rule r_PetriNetReact($n in Net) =
   choose $transSet in Powerset(Transitions($n))
         with (forall $t in $transSet with isEnabled($t)) do
    iterate let ($t = chooseOne($transSet)) in par
        remove($t,$transSet)
        if isEnabled($t) then r_fire[$t] endif
      endpar endlet

//Run all Petri nets
    main rule r_Main = forall $n in Net do r_PetriNetReact[$n]
```

One has also to define a function ι_{PT} which adds to Γ_{PT} the initialization necessary to make the ASM model executable. Any model transformation tool can be used to automatize the ι_{PT} mapping by retrieving data from a terminal model m and creating the corresponding ASM initial state in the target ASM model. We adopted the ATL model transformation engine to implement such a mapping. Essentially, for each class instance of the terminal model, a static 0-ary function is created in the signature of the ASM model Γ_{PT} in order to initialize the domain corresponding to the underlying class. Moreover, class instances with their properties values and links are inspected to initialize the ASM functions declared in the ASM signature. For example, for the Petri net shown in Fig. 5, the ι_{PT} mapping would automatically add to the original Γ_{PT} the initial state (and therefore the initial marking) shown in Listing 2. The initialization of the abstract domains `Net`, `Transition`, and `Place`, and of all functions defined over these domains, are added to the original Γ_{PT}.

Listing 2. ι_{PT}

```
asm PT_hooking
signature:
   ....
   static myNet: Net
   static P1,P2,P3,P4:Place
   static t1,t2:Transition
   ....
default init s0:
 //Functions on Net
 function places($n in Net) = at({myNet −> {p1,p2,p3,p4}},$n)
 function transitions($n in Net) = at({myNet −> {t1,t2}},$n)

 //Functions on Place (the "initial marking")
 function tokens($p in Places) =
                  at({p1−>1,p2−>0,p3−>2,p4−>1},$p)

 //Functions on Transition
 function inputPlaces($t in Transition) =
                  at({t1−>p1,t2−>{p2,p3}},$t)
 function outputPlaces($t in Transition) =
                  at({t1−>{p2,p3},t2−>{p4,p1}},$t)
```

6. Related work

Concerning the metamodeling technique for language engineering, we can mention the official metamodels supported by the OMG for MOF itself, for UML, for OCL, etc. A recent result [10] shows how to apply metamodel-based technologies for the creation of a language description for Sudoku. This is on the same line of our approach of exploiting MDE technologies to develop a tool-set around ASms.

Formal methods communities like the Graph Transformation community [11], [12] and the Petri Net community [13], 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 [14]. Recently, a metamodel for the AsmL language is available as part of a zoo of metamodels defined by using the KM3 meta-language [15]. However, this metamodel is not appropriately documented or described elsewhere, so this prevented us to evaluate it for our purposes.

On the problem of integrating graphical notations and formal methods, [16] shows how the process algebra CSP and the specification language Object-Z, can be integrated into an object-oriented software engineering process employing the UML as a modeling and Java as an implementation language. In [17], the author presents an approach to formal methods technology exploitation which introduces formal notations into critical systems development processes. Both approaches are based on translating graphical models to formal specifications, and are similar to our approach on moving from terminal models of a metamodel-based language to an ASM specification.

An MDE-based approach for integrating different formal methods was recently proposed in [18]. Firstly, the heterogeneous formal models are introduced into MDE as domain specific languages by metamodeling. Then, transformation rules are built for semantics mapping. At last, model-text syntax rules are developed, so as to map models to programs. As case study, the approach was applied for bridging MARTE to LOTOS.

On the application of ASMs for specifying the execution semantics of metamodel-based languages in a MDE style, we can mention the translational approach described in [19], [20]. They propose a *semantic anchoring* to well-established formal models of computation (such as FSMs, data flow, and discrete event systems) built upon AsmL [21] (an ASM dialect), by using the transformation language GME/GReAT (Graph Rewriting And Transformation language) [22]. Still concerning the translational category, two other experiments have to be mentioned: the dynamic semantics of the AMMA/ATL transformation language [23] and SPL, a DSL for telephony services, [24] have been specified in the XASM [25], an open source ASM dialect. A direct mapping from the AMMA meta-language KM3 to an XASM metamodel is used to represent metamodels in terms of ASM universes and functions, and this ASM model is taken as basis for the dynamic semantics specification of the ATL metamodel. However, this mapping is neither formally defined nor the ATL transformation code which implements it have been made available in the ATL transformations Zoo or as ATL use case [26]; only the Atlantic XASM Zoo [27], a mirror of the Atlantic Zoo metamodels expressed in XASM (as a collection of universes and functions), has been made available. By exploiting the semantic framework in [4], we defined in [28] the semantics of the AVALLA language of the AsmetaV validator, a domain-specific modeling language for scenario-based validation of ASM models.

7. Conclusion and future directions

On the basis of our experience in developing the ASMETA toolset, we believe a formal method can gain benefits from the use of MDE automation means either for itself and toward the integration of different formal techniques and their tool inter-operability. Indeed, the metamodel-based approach has the advantage of being suitable to derive from the same metamodel several artifacts (concrete syntaxes, interchange formats, APIs, etc.) which are useful to create, manage and interchange models in a model-driven development context, settling, therefore, a flexible infrastructure for tools development and inter-operability. Moreover, metamodeling allows to establish a "global framework" to enable otherwise dissimilar languages (of possibly different domains) to be used in an interoperable manner by defining precise *bridges* (or *projections*) among different domain-specific languages to automatically execute model transformations. That is in sympathy with the *SRI Evidential Tool Bus idea* [29], and can contribute positively to solve inter-operability issues among formal methods, their notations, and their tools.

On the other hand, the definition of a means for specifying rigorously the semantics of metamodels is a necessary step in order to develop formal analysis techniques and tools in the model-driven context. Along this research line, for example, we are tackling the problem of formally analyzing visual models developed with the UML Profile for SystemC [30]. In conclusion, we believe MDE principles and technologies combined with formal methods elevate the current level of automation in system development and provide the widely demanded formal analysis support, especially exploiting the in-the-loop approach.

Future work will include the definition of the executable semantics of the AsmM metamodel itself by using the ASM-based semantic framework outlined in Sect. 2.2. In this way, we would make a further step in the direction of a tighter integration between ASM and EMF by *closing the loop* (see Fig. 2), i.e. we would use the Asm formal method itself in order to give its semantics by defining suitable semantic mappings in the Asmeta/EMF framework.

References

[1] J. Bézivin, "On the Unification Power of Models," *Software and System Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[2] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

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

[4] A. Gargantini, E. Riccobene, and P. Scandurra, "A semantic framework for metamodel-based languages," *Journal of Automated Software Engineering*, vol. Online First, 2009.

[5] ——, "Model-driven language engineering: The ASMETA case study," in *ICSEA*. IEEE Computer Society, 2008, pp. 373–378.

[6] ——, "A metamodel-based language and a simulation engine for abstract state machines," *J. UCS*, vol. 14, no. 12, pp. 1949–1983, 2008.

[7] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?" *IEEE Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[8] A. Gargantini, E. Riccobene, and P. Scandurra, "Metamodelling a Formal Method: Applying MDE to Abstract State Machines," DTI Dept., University of Milan, Tech. Rep. 97, 2006.

[9] "The Abstract State Machine Metamodel website," `http://asmeta.sf.net/`, 2006.

[10] T. Gjøsæter, I. F. Isfeldt, and A. Prinz, "Sudoku - a language description case study," in *SLE*, ser. LNCS, D. Gasevic, R. Lämmel, and E. V. Wyk, Eds., vol. 5452. Springer, 2008, pp. 305–321.

[11] R. Holt, A. Schürr, S. E. Sim, and A. Winter, "Graph eXchange Language," `http://www.gupro.de/GXL/index.html`.

[12] 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.

[13] "Petri Net Markup Laguage (PNML)," `http://www.informatik.hu-berlin.de/top/pnml`.

[14] 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)*, 2004, pp. 208–223.

[15] F. Jouault and J. Bézivin, "KM3: a DSL for Metamodel Specification," in *Proc. of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 2006.

[16] M. Mueller, E.-R. Olderog, H. Rasch, and H. Wehrheim, "Integrating a formal method into a software engineering process with uml and java," *Form. Asp. Comput.*, vol. 20, no. 2, pp. 161–204, 2008.

[17] J. Armstrong, "Industrial integration of graphical and formal specifications," *Journal of Systems and Software*, vol. 40, no. 3, pp. 211 – 225, 1998.

[18] T. Zhang, F. Jouault, J. Bézivin, and J. Zhao, "A MDE Based Approach for Bridging Formal Models," in *Theoretical Aspects of Software Engineering, 2008. TASE '08. 2nd IFIP/IEEE International Symposium on*. IEEE Computer Society, 2008, pp. 113–116.

[19] K. Chen, J. Sztipanovits, and S. Neema, "Toward a semantic anchoring infrastructure for domain-specific modeling languages," in *EMSOFT*, 2005, pp. 35–43.

[20] ——, "Compositional specification of behavioral semantics," in *DATE*, 2007, pp. 906–911.

[21] "The ASML language website," `http://research.microsoft.com/foundations/AsmL/`, 2001.

[22] D. Balasubramanian, A. Narayanan, C. VanBuskirk, and G. Karsai, "The graph rewriting and transformation language: Great," in *International Workshop on Graph Based Tools (GraBaTs)*, 2006.

[23] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio, "Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs," LINA, Tech. Rep. 06.02, 2006.

[24] ——, "A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development," LINA, Tech. Rep. 06.03, 2006.

[25] M. Anlauff, "XASM - An Extensible, Component-Based ASM Language," in *Proc. of Abstract State Machines*, 2000, pp. 69–90.

[26] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "Atl: a qvt-like transformation language," in *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 719–720.

[27] "The Atlantic XASM Zoo," `http://www.eclipse.org/gmt/am3/zoos/atlanticXASMZoo/`, 2006.

[28] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra, "Exploiting the ASM method for Validation & Verification of Embedded Systems," in *Proc. of ABZ'08, LNCS 5238*. Springer-Verlag, 2008, pp. 71–84.

[29] J. M. Rushby, "Harnessing disruptive innovation in formal verification," in *SEFM*, 2006, pp. 21–30.

[30] A. Gargantini, E. Riccobene, and P. Scandurra, "A Model-driven Validation & Verification Environment for Embedded Systems," in *Proc. of the IEEE third Symposium on Industrial Embedded Systems (SIES'08)*. IEEE, 2008.