

# Combining Formal Methods and MDE Techniques for Model-driven System Design and Analysis

Angelo Gargantini\*, Elvinia Riccobene,<sup>†</sup> and Patrizia Scandurra\*

\*Dipartimento di Ingegneria dell'Informazione e Metodi Matematici (DIIMM)

Università di Bergamo, Viale Marconi, 5 - 24044 Dalmine (BG), Italy

Email: {angelo.gargantini, patrizia.scandurra}@unibg.it

<sup>†</sup>Dipartimento di Tecnologie dell'Informazione (DTI)

Università degli Studi di Milano, via Bramante 65 - 26013 Crema (CR), Italy

E-mail: elvinia.riccobene@dti.unimi.it

**Abstract**—The use of *formal methods*, based on rigorous mathematical foundations, is essential for system specification and proof, especially for safety critical systems. On the other hand, *Model-driven Engineering* (MDE) is emerging as new approach to software development based on the systematic use of models as primary artifacts throughout the engineering life-cycle by combining domain-specific modeling languages (DSMLs) with model transformers, analyzers, and generators. This paper presents our position and experience on combining flexibility and automation of the MDE approach with rigorosity and preciseness of formal methods to achieve significant boosts in both productivity and quality in model-driven design and analysis of software and systems. An *in-the-loop integration* is proposed where, on one hand, MDE principles are used to engineer a language and a tool-set around a formal method for its practical adoption in systems development life cycle, and, on the other hand, the same formal method is used in the same MDE context to endow modeling languages with a precise and (possibly) executable semantics and to perform formal analysis of systems models written in those languages. A concrete scenario of in-the-loop integration is presented in terms of the Abstract State Machine formal method and the Eclipse Modeling Framework. This integration allows system design using the Eclipse Modeling Framework and formal system analysis by Abstract State Machines in a seamless and systematic way, as shown by a concrete case study.

**Keywords**—Formal methods; Model Driven Engineering; Abstract State Machines; model semantics; model execution and analysis

## I. INTRODUCTION

Using *Formal Methods* (FMs), which have rigorous mathematical foundations, for system development is nowadays extremely important, especially for high-integrity systems where safety or security need to be formally proved. On the other hand, the *Model-driven Engineering* (MDE) [2], [3] 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 approaches have advantages and disadvantages that we here shortly summarize (see Fig. 1).

This paper is the extended version of the conference paper [1].

This work is supported in part by the Italian Government under the project PRIN 2007 D-ASAP: *Architetture Software Adattabili e Affidabili per Sistemi Pervasivi* (2007XKEHFA).

**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).

**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 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) [4]; the lack of easy-to-use tools supporting a developer during the life cycle activities of 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 transform into code or 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

	Advantages	Disadvantages
MDE	<ul style="list-style-type: none"> <li>* User-friendly notation</li> <li>* Derivative artifacts for tool development</li> <li>* Automated model transformations</li> </ul>	<ul style="list-style-type: none"> <li>* Lack of semantics</li> <li>* Unfit for model analysis</li> </ul>
FM	<ul style="list-style-type: none"> <li>* Rigorous mathematical foundation</li> <li>* Suitable for model analysis</li> </ul>	<ul style="list-style-type: none"> <li>* Hard notation</li> <li>* Lack of tools</li> <li>* Lack of integration</li> </ul>

Fig. 1: Formal methods and MDE

and semantics of the language from its different concrete notations. Although the foundation constituents of the MDE are still evolving, some MDE principles are implemented in meta-modeling/programming frameworks like the OMG MDA (Model Driven Architecture) [5], Model-integrated Computing (MIC) [6], Software Factories and Microsoft Domain-Specific Languages (DSLs) tools (as part of the Visual Studio SDK) [7], Eclipse/EMF [8], etc. 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 [3].

**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 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 (such as the UML) 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.

In [1], we described 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. In this paper, we extend and deepen this combination view with the final goal of developing a model-driven approach for designing systems according to the MDE principles, and analyzing models by exploiting formal techniques.

Section II provides some related work concerning connections between formal methods and MDE.

Section III describes an overall process, based on the MDE approach, for engineering a language and a tool-set for a formal method. This allows to overcome the lack of user-friendly notations, of integration of techniques, and of their tool inter-operability. This deficiency still poses a significant challenge for formal methods.

Section IV presents an approach to endow language metamodels with precise executable semantics, and we discuss techniques for formal analysis that can be used once formal models are associated to language terminal models by, possibly, automatic model mapping. This addresses the problem of expressing semantics of metamodel-based languages and performing model validation and formal verification.

In order to combine in a tight way rigorousness and preciseness of FMs with flexibility and automation of the MDE, in Section V an *in-the-loop* integration is proposed, where the same MDE technology and FM techniques are involved in both the two activities: MDE for FMs and FMs for MDE.

Section VI provides basic concepts concerning the Abstract State Machine formal method which is later used to implement the *in-the-loop* approach.

Sections VII and VIII show a concrete scenario of *in-the-*

loop integration between the ASM formal method and the EMF framework. On one side, we report our experience in exploiting MDE methodology to engineer a language and a tool-set for the ASMs in order to support their practical use in systems development life cycle. On the other side, we show how ASMs can be used to provide semantics to languages defined in the MDE context and how to perform formal analysis of models developed by MDE technology.

A complete case study is presented in Section IX which shows how MDE-based technologies are used to define a metamodel-based language for the Tic-Tac-Toe, and the ASM-based semantic framework is used to define an executable semantics of the language and to support semantics validation and formal verification of models.

Section X shows how to get a tighter integration between ASM and EMF by *closing the loop*, i.e. by using the ASM formal method itself to define the semantics of the ASMs in the EMF framework.

Finally, our conclusion and future directions are provided in Section XI.

## II. RELATED WORK

Software languages play a cornerstone role in system development. Language engineering processes have been considered in many contexts of software engineering [9]. Concerning the metamodeling technique of MDE for (software) language engineering, many proposals have been presented, which pay attention to the fact that language descriptions take different form in different technical spaces (e.g. metamodels, schemas, grammars, and ontologies) and typically multiple languages (from different technical spaces) need to be used together and integrated in most software development scenarios. A process to engineer languages address several aspects of a language: structure, constraints, textual and graphical representation, parser/compiler, transformational and executional behavior. Research usually faced only one of these aspects, therefore, a comparison with related work can be often done considering single aspects of a language development process.

Formal methods communities have only recently started to settle their tools on metamodels and MDE platforms. A non exhaustive list of such efforts follows. An Event-B metamodel and an EMF-based Framework for Event-B have been recently developed [10] to provide an EMF-based front-end to the Rodin platform, an Eclipse-based IDE for Event-B that provides support for refinement and mathematical proof of Event-B models.

The Maude Formal Tool Environment [11] is an executable rewriting logic language suited for the specification of object-oriented open and distributed systems. It offers tool support for reasoning about Maude specifications and, recently, also an Eclipse plug-in that allows to connect the Maude environment to the KM3 metamodeling framework using ATL (the ATLAS Transformation Language) [12] transformations.

Within the Graph Transformation community, using the concepts of graph transformations and metamodeling, the transformation language GReAT (Graph Rewriting And Transformation language) [13] has been designed to address the

specific needs of the model transformation area of the Model Integrated Computing. It is supported by tools that allow the rapid prototyping and realization of transformation tools.

To the best of our knowledge, the development of the above mentioned languages and tools did not follow a model-driven engineering process like the one described here in Section III.

A metamodel for the ITU language SDL-2000 has been also developed [14]. The authors present also a semi-automatic *reverse engineering* methodology that allows the derivation of a metamodel from a formal syntax definition of an existing language. The SDL metamodel has been derived from the SDL grammar using this methodology. A very similar method to bridge *grammarware* and *modelware* is also proposed by other authors in [15] and in [16]. These approaches are complementary to the development process presented in Sect. III. Our approach has to be considered a *forward engineering* process consisting in deriving a concrete textual notation from an abstract metamodel.

A recent result [17] 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.

Within the ASM community, a number of notations and tools have been developed for the specification and analysis [18]. The Abstract State Machine Language (AsmL) developed by the Foundation Software Engineering group at Microsoft is the greatest effort. AsmL is a rich executable specification language, based on the theory of Abstract State Machines, expression- and object- oriented, and fully integrated into the Microsoft .NET framework. 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" [19]. Adopting a terminology currently used, AsmL is a platform-specific modeling language for the .NET type system. Of the remaining tools for ASMs, let us mention the more popular ones: the CoreASM, an extensible execution engine developed in Java, TASM (Timed ASMs), an encoding of Timed Automata in ASMs, and a simulator-model checker for reactive real-time ASMs [20] able to specify and verify First Order Timed Logic (FOTL) properties on ASM models. Among these, the CoreASM engine is the more comparable to our. Other specific languages for the ASMs, no longer maintained, are ASM-SL, which adopts a functional style being developed in ML and which has inspired us in the language of terms, the AsmGofer language based on the Gofer environment, and XASM which is integrated in Montages, an environment generally used for defining semantics and grammar of programming languages. All the above tools, however, do not rely on MDE principles and techniques, and, except CoreASM that is based on an extensible architecture, none of the others are designed to support model exchange and tool integration. Recently, a metamodel for the AsmL language is available as part of a zoo of metamodels defined by using the KM3 meta-language. However, this metamodel is not appropriately documented or described elsewhere, so this prevented us to evaluate it.

Regarding the derivation of concrete grammars for meta-

models, developing a grammar for the ASMs from the meta-model was challenging and led us to the definition of a bridge between grammars and metamodels as explained in [21]. This part of the process required at least six man month. Although we did not automatize these rules, because no advanced model-to-text tools were available at that time and because we wanted to derive only one grammar for AsmetaL, the rules may be easily reused for other formalisms. Several model-to-text tools exist now: EMFText [22] working for Ecore metamodels, TCS [23] (Textual Concrete Syntax) for metamodels written in KM3, TEF (Textual Editing Framework) for EMF-based metamodels, etc. Vice versa, Xtext [24] allows to derive a language metamodel from the language concrete textual grammar. An overview of textual grammars and metamodel is given in [25]. Other more complex model-to-text tools, capable of generating text grammars from specific MOF based repositories, exist [26], [27]. These tools render the content of a MOF-based repository (known as a MOFlet) in textual form, conforming to some syntactic rules (grammar). However, though automatic, since they are designed to work with any MOF model and generate their target grammar based on predefined patterns thus they do not permit a detailed customization of the generated language.

On the problem of integrating graphical notations and formal methods, [28] 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 [29], the author presents an approach to formal methods technology exploitation which introduces formal notations into critical systems development processes. Furthermore, [30] proposes a metamodel-based transformation technique, which is founded by a set of structural and semantic mappings between UML and B, to assist derivation of formal B specifications from UML diagrams. All these 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. However, they are tailored for the UML, while our approach refer to generic metamodel-based languages, and they perform only one side of the in-the-loop integration.

An MDE-based approach for integrating different formal methods was recently proposed in [31]. As in our approach, formal models are introduced into MDE as domain specific languages by developing their metamodels. Then, transformation rules are defined to obtain notation bridges. At last, model-to-text syntax rules are developed, so to map models into programs. As case study, the approach was applied for bridging MARTE to LOTOS. The main goal of their work is to integrate different formal notations in software development, however they do not provide semantics to them. General challenges of tool integration are discussed in [32], where a software language engineering solution technique is presented that apply MDE principles to address tool interoperability.

Concerning the problem of specifying the semantics of metamodel-based languages, some recent works, such as Kermeta [33], aim at providing executability into current metamodeling frameworks. Another effort toward this same

direction is presented in [34] where the authors describe the M3Actions framework to support operational semantics for EMF models. The Maude formalism is also proposed in [35] as a way for specifying the semantics of visual modeling languages.

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 [36]. They propose a *semantic anchoring* to well-established formal models of computation (such as FSMs, data flow, and discrete event systems) built upon AsmL, by using the transformation language GME/GReAT. The proposed approach offers up predefined and well-defined sets of *semantic units* for future (conventional) anchoring efforts. However, we see two main disadvantages in this approach: first, it requires well understood and safe behavioral language units and it is not clear how to specify the language semantics from scratch when these language units do not yet exist; second, in *heterogeneous systems*, specifying the language semantics as composition of some selected primary semantic units for basic behavioral categories [37] is not always possible, since there may exist complex behaviors which are not easily reducible to a combination of existing ones. Still concerning the translational category, in [38] the dynamic semantics of the AMMA/ATL transformation language was specified in the XASM [39] 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 [12]; only the Atlantic XASM Zoo [40], a mirror of the Atlantic Zoo metamodels expressed in XASM (as a collection of universes and functions), has been made available. A further recent result [41] proposes ASMs, Prolog, and Scheme as description languages in a framework named EProvide 2.0 for prototyping the operational semantics of metamodel-based languages. Their approach is also translational as it is based on three bridges: a physical, a logical, and a pragmatical bridge between grammarware language and modeling framework.

By exploiting our ASM-based semantic framework [42], we also defined the semantics of the AVALLA language [43] of the AsmetaV validator, a domain-specific modeling language for scenario-based validation of ASM models. Moreover, in [44] we adapt one of the techniques in [42], the *meta-hooking*, for UML profiles, and we show its application to the *SystemC Process (SCP) state machines* formalism of the SystemC UML profile [45].

### III. MDE FOR FMS

Applying the MDE development principles to a formal method has the overall goal of engineering a language and a tool-set around the formal method in order to support its practical use in systems development life cycle.

The MDE methodology for engineering software languages is well established in the context of domain-specific languages

[46]. Nevertheless, this model-driven development process can be adapted to formal methods, too.

The first step of this engineering process is the *choice of a metamodeling framework and its supporting technologies*. 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 meta-modeling 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 development of concrete notations in textual form. It should also provide a mapping towards programming languages (i.e. API artifacts) to allow the integration with other software applications.

Once a metamodeling framework has been chosen, the further main steps, that might require iterative processing, of the process are the following.

**Design of a language abstract syntax.** In the MDE context, the *abstract syntax* of a specification language is defined by means of a *metamodel* [47]. It is an object-oriented model of the vocabulary of the language. It represents concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models. Precise guide lines exist (e.g., [46]) to drive this modeling activity that leads to an instantiation of the chosen metamodeling framework for a specific domain of interest. This is a critical process step since the metamodel is the starting point for tool development.

**Development of tools.** Software tools are developed starting from the language metamodel. They can be classified in *generated*, *based*, and *integrated*, depending on the decreasing use of MDE generative technologies for their development. The effort required by the user increases, instead. Software tools automatically derived from the metamodel are considered generated. Based tools are those developed exploiting artifacts (APIs and other concrete syntaxes) and contain a considerable amount of code that has not been generated. Integrated tools are external and existing tools that are connected to the language artifacts: a tool may use just the XMI format, other tools may use the APIs or other derivatives. In the sequel we explain these kinds of tools.

1) *Development of language artifacts.* From the language metamodel, several *language artifacts* are generated for model handling – i.e. model creation, storage, exchange, access, manipulation –, and these artifacts can be reused during the development of other applications. Artifacts are obtained by exploiting standard or proprietary mappings from the metamodeling framework to several technical spaces, as XMLware for model serialization and interchange, and Javaware for model representation in terms of programmable objects (through standard APIs).

2) *Definition and validation of concrete syntax(es).* Language concrete notations (textual, graphical or both) can be introduced for the human use of editing models conforming

to the metamodel. Several tools exist to define (or derive) concrete textual grammars for metamodels. For example, EMFText [22] allows defining text syntax for languages described by an Ecore metamodel and it generates an ANTLR grammar file. TCS [23] (Textual Concrete Syntax) enables the specification of textual concrete syntaxes for Domain-Specific Languages (DSLs) by attaching syntactic information to metamodels written in KM3. A similar approach is followed by the TEF (Textual Editing Framework) [48]. Other tools, like the Xtext by openArchitectureWare [49], following different approaches, may fit in our process as well. Depending on the degree of automation provided by the chosen framework, concrete syntax tools can be classified between generated and based software.

Besides to be defined, concrete grammars must be also validated. To this aim, a pool of models written in the concrete syntax and acting as benchmark has to be selected. During this activity it is important to collect information about the coverage of language constructs (classes, attributes and relations) to check that all them are used by the examples. Writing wrong models and checking that they are not accepted is important as well. Coverage evaluation can be performed by using a code coverage tool and instrumenting the parser accordingly. This validation activity is also useful to provide confidence that the metamodel correctly captures concepts and constructs of the underline formal method.

3) *Development of other tools.* Metamodel, language artifacts, and concrete syntaxes are the foundations over which new tools can be developed and existing ones can be integrated.

#### IV. FMS FOR MDE

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

##### A. Language semantics definition

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 [50] 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.

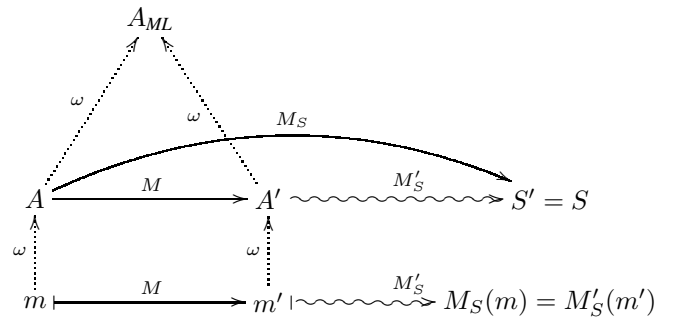


Fig. 2: The building function  $M$

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) 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) defining the semantic mapping  $M_S : A \rightarrow S$  as

$$M_S = M'_S \circ M$$

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  $M$ .

Note that the function  $M$  can be applied to terminal models conforming to  $A$  in order to obtain models conforming to  $A'$ , as shown in Fig. 2. In this way, the semantic mapping  $M_S : A \rightarrow S$  associates a well-formed terminal model  $m$  conforming to  $A$  with its semantic model  $M_S(m)$ , by first translating  $m$  to  $m'$  conforming to  $A'$  by means of the  $M$  function, and then applying the mapping  $M'_S$  which is already well-defined.

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 heterogeneous models of computation (MoC) in order to smoothly integrate different behavioral models; (iii) be endowed with a model refinement mechanism leading to correct-by-construction system artifacts. Furthermore, as MDE specific requirement (iv),  $L'$  should be possibly endowed with a metamodel-based definition in order to automatize the application of building function  $M$  by exploiting MDE techniques of automatic model transformation.

##### B. Formal analysis

Besides the above stated requirements about the expressive power of  $L'$  as notation, it is important that formal analysis of models written in  $L'$  is supported by a set of tools for model execution, as simulation or testing, and for model verification. Indeed, the main goal of applying a formal notation to the semantics of  $L$  is to allow formal analysis of the models written in  $L$ .

As main formal activities that are allowed by applying a formal method to a language  $L$ , we identify at least: *model validation* and *property verification*.

*Validation* is intended as the process of investigating a model (intended as formal specification) with respect to its user perceptions, in order to ensure that the specification really reflects the user needs and statements about the application, and to detect faults in the specification as early as possible with limited effort. Techniques for validation include *scenarios generation*, when the user builds scenarios describing the behavior of a system by looking at the observable interactions between the system and its environment in specific situations; *simulation*, when the user provides certain input and observes if the output is the expected one or not (it is similar to code debugging); *model-based testing*, when the specification is used as oracle to compute test cases for a given critical behavior of the system at the same level of the specification. These abstract test cases cannot be executed at code level since they are at a wrong level of abstraction. Executable test cases must be derived from the abstract ones and executed at code level to guarantee conformance between model and code.

In any case, validation should precede the application of more expensive and accurate methods, like *requirements formal analysis* and *verification of properties*, that should be applied only when a designer has enough confidence that the specification captures all informal requirements. Formal verification has to be intended as the mathematical proof of system properties, which can be performed by hand or by the aid of model checkers (which are usable when the variable ranges are finite) or of theorem provers (which require strong user skills to drive the proof).

Model validation techniques can be also used during the development of the language semantics of  $L$  for *semantic validation*. This activity consists in checking (or proving, if possible) that the building function  $M$  really captures the intended semantics of  $L$ , and it must be performed before any formal analysis of models. Indeed every later formal activity on models written in  $L$  is based on  $M$  and a faulty  $M$  would jeopardize the results obtained.

## V. 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 an *in-the-loop* integration approach. In this approach, the MDE framework and the FM notation are the same in both of the above activities and the application of the MDE to the FM is carried out 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

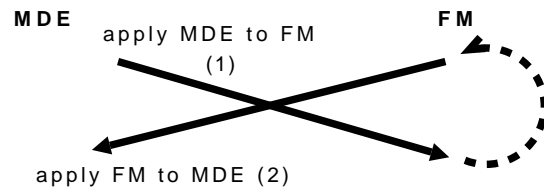


Fig. 3: In the loop integration of FM and MDE

metamodel (see Sect. IV), a formal notation endowed with a representation of its concepts in terms of a metamodel would allow the use of 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. VII and VIII present our instantiation of the *in-the-loop* integration with the EMF (Eclipse Modeling Framework) as MDE 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 well known technologies, i.e. Java, XML, and UML, currently used for software development.
- ASMs own all the characteristics of preciseness, abstraction, refinement, executability, metamodel-based definition that we identified as the desirable properties a FM should have in order to be a good candidate for integration.

In order to make a further step in the direction of a tighter integration between ASM and EMF, Sect. X shows how effectively we can *close the loop* (see Fig. 3) by describing the semantics of ASMs representation in the EMF framework by using the ASM formal method itself.

## VI. ABSTRACT STATE MACHINES

Abstract State Machines are an extension of FSMs [51], where unstructured 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 (boolean functions) defined on them, while 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 Condition then Updates**” where *Updates* is a set of function updates of the form  $f(t_1, \dots, t_n) := t$  that are simultaneously executed when *Condition* is true,  $f$  is an arbitrary  $n$ -ary function, and  $t_1, \dots, t_n, t$  are first-order terms. To fire this rule to a state  $S_i$ ,  $i \geq 0$ , evaluate all terms  $t_1, \dots, t_n, t$  at  $S_i$  and update the function  $f$  to  $t$  on parameters  $t_1, \dots, t_n$ . This produces another state  $S_{i+1}$  which differs from  $S_i$  only in the new interpretation of the function  $f$ . An ASM  $M$  is therefore a finite set of rules for such guarded multiple function updates.

Function are classified as *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions, and *basic* functions which can be

*static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* and *output* (only write) functions.

These is a limited but powerful set of *rule constructors* that allow to express simultaneous parallel actions (*par*), sequential actions (*seq*), iterations (*iterate*, *while*, *rec-while*), and submachine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification *choose*) and unrestricted synchronous parallelism (universal quantification *forall*).

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

The notion of ASMs formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*. Furthermore, it supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synchronous/Asynchronous Multi-agent ASMs*.

Although the ASM method comes with a rigorous mathematical foundation, ASMs provide accurate yet practical industrially viable behavioral semantics for pseudocode on arbitrary data structures. We quote here this *working* definition of an ASM defined as a tuple (*header*, *body*, *main rule*, *initialization*).

The *header* contains the *name* of the ASM and its *signature*, namely all declarations of domains, functions, and predicates. The header may contain also *import* and *export* clauses, i.e., all names for functions and rules that are, respectively, imported from other ASMs, and exported from the current one. We assume that there are no name clashes in these signatures.

The *body* 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 (definitions) of transition rules and definitions of *axioms* for invariants one wants to assume for domains and functions of the ASM.

The (unique) *main rule* is a transition rule and represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body). The main rule is *closed* (i.e. it does not have parameters) and since there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on a variable assignment, but only on the state of the machine.

The *initialization* of an ASM is a characterization of the initial states. An initial state defines an initial value for domains and functions declared in the signature of the ASM. *Executing* an ASM means executing its main rule starting from a specified initial state.

A complete mathematical definition of the ASM method can be found in [52], together with a presentation of the great variety of its successful application in different fields such as: 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

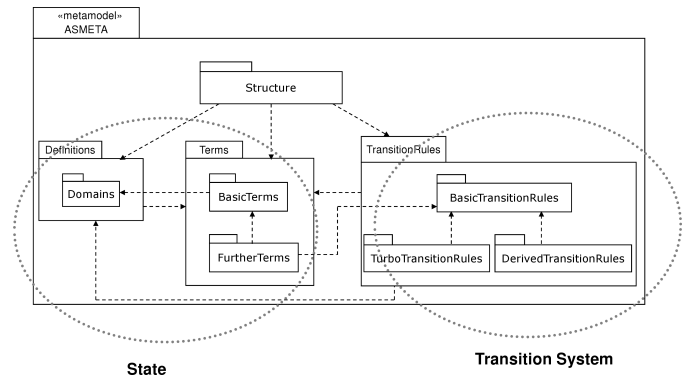


Fig. 4: Package structure of the AsmM metamodel

analysis of protocols, architectural design, language design, verification of compilation schemas and compiler back-ends, etc.

## VII. EMF FOR ASMS

In addition to its mathematical-based foundation, a metamodel-based definition for ASMs has been given [53], [54]. This ASM metamodel allowed us to apply MDE techniques for developing a general framework, called ASMmETA- modeling framework (ASMETA) [55], for a wide inter-operability and integration of new and existing tools around ASMs (ASM model editors, ASM model repositories, ASM model validators, ASM model verifiers, ASM simulators, ASM-to-Any code generators, etc.).

### A. ASM Metamodel

We started by defining a metamodel [55], [56], [53], [54], the *Abstract State Machine Metamodel* (AsmM), as abstract syntax description of a language for ASMs. The aim was that of developing a *unified* abstract notation for the ASMs, independent from any specific implementation syntax and allowing a more direct encoding of the ASM mathematical concepts and constructs.

The complete AsmM metamodel is organized in one package called ASMETA containing 115 classes, 114 associations, and 150 class invariants expressed in the OMG OCL language [57], approximatively. The ASMETA package is further divided into four packages as shown in Fig. 4. Each package covers different aspects of the ASMs. The dashed gray ovals in Fig. 4 denote packages representing the notions of *State* and *Transition System*, respectively. The Structure package defines architectural constructs (modules and machines) required to specify the backbone of an ASM model. The Definitions package contains all basic constructs (functions, domains, constraints, rule declarations, etc..) which characterize algebraic specifications. The Terms package provides all kinds of syntactic expressions which can be evaluated in a state of an ASM. The TransitionRules package contains all possible transition rules schemes of Basic and Turbo ASMs. All *derived* transition rules are contained in the DerivedTransitionRules package. These rules are other ASM transition rule schemes derived from the basic

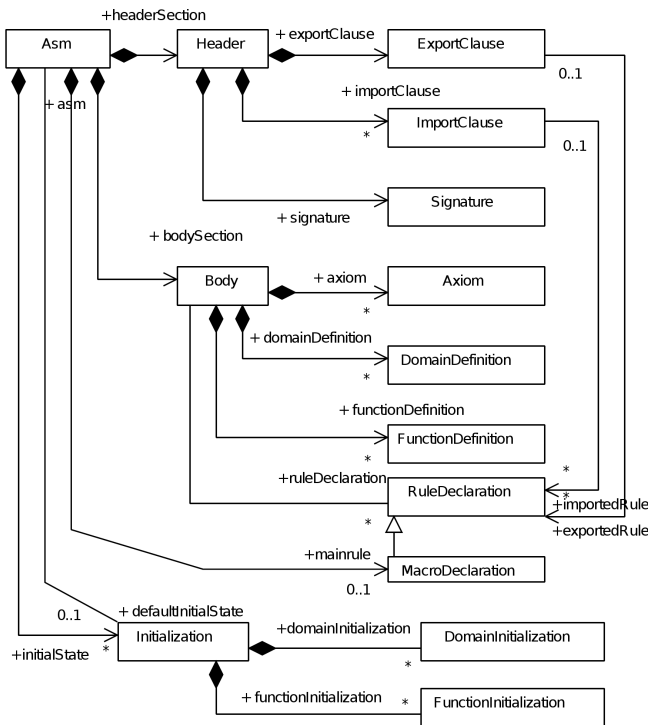


Fig. 5: Backbone

and the turbo ones, respectively. Although they could be easily expressed at model level in terms of other existing rule schemes, they are considered “syntactic sugar” and therefore they have been included in the metamodel. Example of such rules are the case-rule and the (turbo) iterative/recursive while-rule. All relations between packages are of type *uses*.

We present here only a very small fragment of the *AsmM* whose complete description can be found in [53], [55]. Fig. 5 shows the backbone of a *basic ASM*. An instance of the root class *Asm* represents an entire ASM specification. According to the definition given in Sect. VI, a basic ASM has a name and is defined by a *Header* (to establish the signature), a *Body* (to define domains, functions, and rules), a main rule, and a set of initial states (instances of the *Initialization* class). All possible initial states are linked to an ASM by the association end *initialState* and one initial state is elected as *default* (see the association end *defaultInitialState*). ASM rule constructors are represented by subclasses of the class *Rule*, not reported here.

## B. ASMETA tool-set

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 [58]. The *AsmM* and the combination of these language artifacts lead to an instantiation of the EMF metamodeling framework for the ASM application domain, the ASMETA framework that provides a global infrastructure

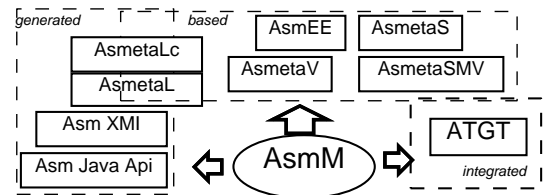


Fig. 6: The ASMETA tool set

for the interoperability of ASM tools (new and existing ones) [59].

The ASMETA tool set (see Fig. 6) 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* constraints expressed in the OCL language; a simulator, *AsmetaS*, to execute ASM models; the *Avalla* language for scenario-based validation of ASM models, with its supporting tool, the *AsmetaV* validator; a model checker *AsmetaSMV* [60] for model verification by NuSMV; 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.

## VIII. 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. We also describe how the ASM tool-set provides a concrete support for model analysis.

### A. Language semantics definition

Recall, from Sect. IV, 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. IV, 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 [52] and the *semantic mapping*  $M_S : AsmM \rightarrow S_{AsmM}$  to relate syntactic concepts to those of the semantic domain is given in [58].



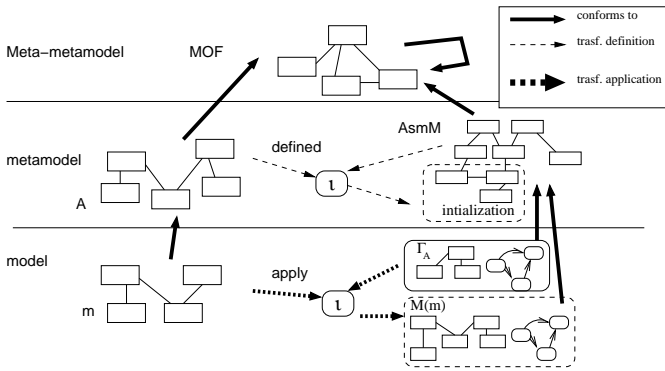


Fig. 7: Semantic hooking

The semantics of a metamodel-based language is expressed in terms of ASM transition rules by providing the building function  $M : A \rightarrow AsmM$ . As already mentioned, the definition of the function  $M$  may be accomplished by different techniques (see [42]), 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. This technique is used in Section IX-B to provide behavioral semantics of the language in our case study.

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  $\Gamma_A$ , which is an instance of  $AsmM$  and contains all data structures modeling elements of  $A$  with their relationships, and all transition rules representing behavioral aspects of the language.  $\Gamma_A$  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  $\iota$ . Formally, the building function  $M$  is given by  $M(m) = \iota_A(\Gamma_A, m)$ , for all  $m$  conforming to  $A$ .

$\Gamma_A : AsmM$ , is an abstract state machine which contains only declarations of functions and domains (the signature) and the behavioral semantics of  $L$  in terms of ASM transition rules.

$\iota_A : AsmM \times A \rightarrow AsmM$ , properly initializes the machine.  $\iota_A$  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  $\iota_A$  function is applied to  $\Gamma_A$  and to the terminal model  $m$  for which it yields the final ASM.

Examples of applying the semantic hooking technique to define the semantics of a metamodel-based language can be found in [42] for a metamodel of Finite State Machines and in [1] for a metamodel of the Petri net formalism. The latter is also reported in Appendix A and can be viewed as an example which facilitates the reader in understanding our approach since the semantics of Petri nets is well-known.

## B. Formal analysis

The ASM-based semantic framework supports formal analysis of ASM models by exploiting the ASMETA tool-set (see Section VII-B for details) for model validation and verification.

1) *Model validation*: Simple model validation can be performed by *simulating* ASM models with the ASM simulator (see Section VII-B) to check a system model with respect to the desired behavior to ensure that the specification really reflects the user needs and statements about the system, and to detect faults in the specification as early as possible with limited effort.

The AsmetaS simulator can be used in a standalone way to provide basic simulation of the overall system behavior. As key features for model validation, AsmetaS supports *axiom checking* to check whether axioms expressed over the currently executed ASM model are satisfied or not, *consistent updates checking* for revealing inconsistent updates, *random simulation* where random values for monitored functions are provided by the environment, *interactive simulation* when required input are provided interactively during simulation, and configurable *logging* facilities to inspect the machine state. Axiom checking and random simulation allow the user to perform a draft system validation with minimal effort, while interactive simulation, although more accurate, requires the user interaction.

The most powerful validation approach is the *scenario-based validation* [61] by the ASM validator (see Section VII-B). The AsmetaV validator is based on the AsmetaS simulator and on the Avalla modeling language. This last provides constructs to express execution scenarios in an algorithmic way as interaction sequences consisting of *actions* committed by the *user actor* to set the environment (i.e. the values of monitored/shared functions), to check the machine state, to ask for the execution of certain transition rules, and to enforce the machine itself to make one step (or a sequence of steps by step until) as reaction of the actor actions.

AsmetaV reads a user scenario written in Avalla, it builds the scenario as instance of the Avalla metamodel by means of a parser, it transforms the scenario and the AsmetaL specification which the scenario refers to, to an executable AsmM model. Then, AsmetaV invokes the AsmetaS interpreter to simulate the scenario. During simulation the user can pause the simulation and watch the current state and value of the update set at every step, through a watching window. During simulation, AsmetaV captures any check violation and if none occurs it finishes with a “PASS” verdict. Besides a “PASS”/“FAIL” verdict, during the scenario running AsmetaV collects in a final report some information about the *coverage* of the original model; this is useful to check which transition rules have been exercised.

2) *Model checking*: The ASMETA tool-set provides support for temporal properties verification of ASM models by means of the model checker AsmetaSMV [60], which takes in input ASM models written in AsmetaL and maps these models into specifications for the model checker NuSMV [62].

AsmetaSMV supports both the declaration of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulas. CTL/LTL properties to verify are declared directly into the ASM model as (special) axioms of the form:

$$\text{axiom over } [ctl \mid ltl] : p$$

where the over section specifies if  $p$  is a CTL or a LTL formula. No knowledge of the NuSMV syntax is required to

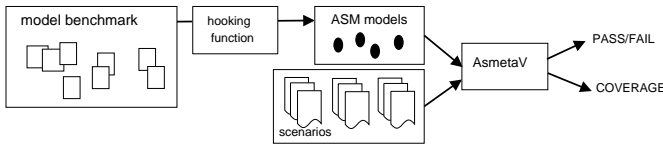


Fig. 8: Semantic validation by AsmetaV

the user in order to use AsmetaSMV.

3) *Language semantics validation*: The ASMETA tool-set and the validation techniques can also be used for *language semantics validation*. Indeed, this activity is performed through the validation of the hooking function  $M$  presented in Section VIII-A by applying it to a collection of meaningful examples. The ASM models obtained from the application of  $M$  to the examples can be validated in different ways providing increasing degrees of confidence in the semantics correctness. *Random simulation* allows checking if errors like inconsistent updates and type errors, occur. *Interactive simulation* can provide evidence that the semantics captures the intended behavior, but it requires the user to provide the correct inputs and to judge the correctness of the observed behavior. The most powerful validation approach is the *scenario-based validation*. As shown in Fig. 8, a suitable set of models are selected as benchmark for language semantic validation; these models are translated into ASM models by the hooking function  $M$ ; moreover, a set of scenarios specifying the expected behavior of the models must be provided by the user and are used for validation. These scenarios can be written from scratch in the Avalla language, or alternatively, if the language  $L$  has already a simulator, these scenarios may be derived from the execution traces generated by such a simulator. The second approach is useful to check the conformance of the semantics implemented by  $L_S$  with respect to the semantics defined by the hooking function  $M$ . The ASM validator provides also useful information about the coverage obtained by the scenarios.

## IX. THE TIC-TAC-TOE EXAMPLE

As a case study, we consider Tic-Tac-Toe as a language, where a Tic-Tac-Toe board is an instance of the language. We use MDE-based technologies to define a metamodel for a description language of the Tic-Tac-Toe game, and the ASM-based semantic framework for the definition of the execution semantics of a board (for playing) including correctness checking by validation and verification.

### A. Tic-Tac-Toe abstract syntax

Fig. 9 shows the metamodel for the Tic-Tac-Toe. It describes the static structure of a board (the Board class) maintaining data seen by users: rows (the Row class) and squares (the Square class). A board has (see references hrows, vrows, and drows): three horizontal rows, three vertical rows, and two diagonal rows. Totally, in a board there are nine squares (see the reference square), three per each row (the squareInRow reference). The SKind enumeration type denotes the kind of symbols a square can contain (cross, nought, empty). The default symbol is empty.

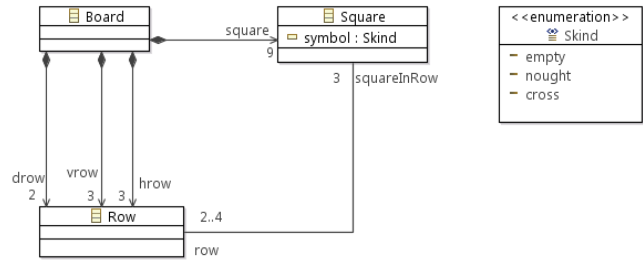


Fig. 9: A metamodel for Tic-Tac-Toe

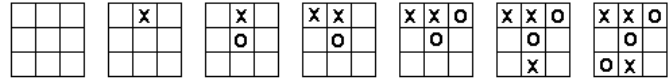


Fig. 10: Examples of Tic-Tac-Toe boards

Each square is contained in one row and one vertical row. Some squares may be contained in more than one row. The square in the center, for example, is contained in the middle vertical row and horizontal row, and in the two diagonal rows. All these structural constraints can be expressed in OCL. For example, the following OCL invariant

```
Context : Board
inv RowColumnCommonSquares :
self.hrow.squareInRow ->
intersection(self.vrow.squareInRow)->size()=1
```

states that an horizontal row and a vertical row can only have exactly one square in common.

Fig. 10 shows (using a graphical concrete syntax) examples of Tic-Tac-Toe boards as instances (terminal models) of the Tic-Tac-Toe metamodel in Fig 9.

### B. Tic-Tac-Toe semantics definition

According to the hooking technique, first we have to specify an ASM  $\Gamma_{Tic-Tac-Toe}$  containing the signature and the behavioral semantics of the Tic-Tac-Toe metamodel in terms of ASM transition rules. Listings 1 (for the signature), 2 and 3 (for the transition rules) report portions of a possible  $\Gamma_{Tic-Tac-Toe}$  in AsmetaL for a computer (symbol O) vs user (symbol X) Tic-Tac-Toe game. The complete ASM model is reported in Appendix B.

The signature (see Listing 1) introduces domains and functions for representing a board such as the enumeration SKind, domains for squares and rows as subsets of the predefined Integer domain, and so on. The signature also provides domain and functions for managing the overall game. Each player takes alternating turns (see the function status) trying to earn three of their symbols in a row horizontally, vertically, or diagonally. The game can end with a player winning (represented by the whoWon function) by getting three of his/her symbol in row (as denoted by the function hasThreeOf) or end in a draw, i.e. no spaces left on the board with none winning (as denoted by the noSquareLeft function). The winner is determined by position of board; no history needs to be recorded (only board position before and after turn). If there is no winner after nine clicks, there is a

Listing 1:  $\Gamma_{Tic-Tac-Toe}$  signature

```

asm Tictactoe
signature:
//For representing a board
enum domain Skind = {CROSS|NOUGHT|EMPTY}
domain Square subsetof Integer
domain Row subsetof Integer
static squaresInRow: Prod(Row,Integer) -> Square
controlled symbol: Square -> Skind

//For managing the game
enum domain Finalres = {PLAYERX|PC|TIE}
enum domain Status = {TURNX|CHECKX|TURNPC|CHECKPC
|GAMEOVER}
monitored playerX:Square // move of X
controlled status: Status
controlled whoWon: Finalres
derived noSquareLeft : Boolean
derived hasThreeOf: Prod(Row,Skind) -> Boolean

//For PC strategies
controlled count: Integer
derived openingPhase: Boolean
controlled lastMoveX: Square
static isCorner: Square -> Boolean
static isEdge: Square -> Boolean
static isCenter: Square -> Boolean
derived hasTwo: Row -> Boolean
static opposite: Square -> Square

```

tie. Note that the square selected by the player X (the user) is represented by a monitored function `moveX`, and therefore is provided at each step as input value to the ASM; the computer move (the square to mark) is instead calculated according to some playing strategies. Further domains and functions are introduced in the signature to implement these PC strategies, as better explained later in the text.

The behavior of the overall game is provided by the main rule `r_Main` (see Listing 2) where at each step a check for a winner or a tie (rule `r_checkForAWinner`) or a move of a player is executed depending on the status of the game. The two rules `r_movePlayerX` and `r_movePC` specify the execution behavior of the two players. The behavior of the user (player X) is straightforward as the square to mark is provided interactively through the monitored function `moveX`. The behavior of the computer depends instead by the chosen strategy as formalized by the invoked `r_tryStrategy` rule.

Listing 3 reports the definition of the `r_tryStrategy` rule and of the invoked macro rules for making a computer play the game. To this goal, we formalize by ASM rules a children's strategy that is divided in two phases: *opening phase* (opening of the game) and *draw phase* (after opening of both players). Note that to build an unbeatable opponent (especially if we want to learn a computer to play it), we need to use a *minimax* approach of Game Theory. We remark that this is out of the scope of this work. So, here we limit to express a children's strategy.

For the opening phase (see the `r_opening_strategy` rule in Listing 3), as first player the computer has three possible positions to mark during the first turn. Superficially, it might seem that there are nine possible positions, corresponding to the nine squares in the board. However, by rotating the

Listing 2:  $\Gamma_{Tic-Tac-Toe}$  transition rules for game management

```

asm Tictactoe
...
rule r_movePC = par
  r_tryStrategy[NOUGHT]
  count := count + 1
  status := CHECKPC
endpar

rule r_movePlayerX = if symbol(playerX)= EMPTY
  then par
    symbol(playerX):= CROSS
    count := count + 1
    lastMoveX := playerX
    status := CHECKX
  endpar
  else status := TURNX
endif

rule r_checkForAWinner($symbol in Skind) =
//GAME OVER WITH A WINNER?
if (exist $r in Row with hasThreeOf($r,$symbol)) then
  par
    status := GAMEOVER
    if $symbol = CROSS then whoWon:= PLAYERX
    else whoWon:= PC
  endif
endpar
//GAME TIE?
else if ( noSquareLeft )
  then par
    status := GAMEOVER
    whoWon := TIE
  endpar
else
  if $symbol = CROSS then status:= TURNPC
  else status:= TURNX
endif endif endif

main rule r_Main =
if status = TURNX then r_movePlayerX[]
else if status = CHECKX then r_checkForAWinner[CROSS]
else if status = TURNPC then r_movePC[]
else if status = CHECKPC then r_checkForAWinner[NOUGHT]
endif endif endif endif

```

board, we will find that in the first turn, every corner mark is strategically equivalent to every other corner mark. The same is true of every edge mark. For strategy purposes, there are therefore only three possible first marks: corner, edge, or center. The computer can win or force a draw from any of these starting marks; however, playing the corner gives the opponent the smallest choice of squares which must be played to avoid losing. In the `r_opening_strategy` rule, the computer chooses therefore a corner (see the rule `r_playACorner`) in case of first player. As second player, the computer must respond to X's opening mark in such a way as to avoid the forced win. The computer (player O) must always respond to a corner opening with a center mark, and to a center opening with a corner mark. An edge opening must be answered either with a center mark, a corner mark next to the X, or an edge mark opposite the X. For simplicity, in this case we play always the center as formalized in the `r_opening_strategy` rule. Any other responses will allow X to force the win. Once the opening is completed, O's task is to follow the below draw strategy in order to force

Listing 3:  $\Gamma_{Tic-Tac-Toe}$  transition rules for the game strategies

```

asm Tictactoe
...
//A very naive player: choose an empty square and mark it.
rule r_naive_strategy ($symbol in Skind)=
  choose $s in Square with symbol($s)=EMPTY
  do symbol($s):= $symbol

rule r_playACorner($symbol in Skind) =
  choose $s in Square with (symbol($s)=EMPTY and isCorner($s))
  do symbol($s):= $symbol

//Opening strategy
rule r_opening_strategy ($symbol in Skind)=
  if (count=0) //first mark
  then r_playACorner[$symbol]
  else //second mark
    if symbol(5) = EMPTY then symbol(5):=$symbol //play the center
    else r_playACorner[$symbol] //we play a corner
  endif
endif

//Mark with $symbol the last empty square within row $r
rule r_markLastEmpty ($r in Row, $symbol in Skind) =
  choose $x in {1,2,3} with symbol(squaresInRow($r,$x))=EMPTY
  do symbol(squaresInRow($r,$x)) := $symbol

//Draw strategy (with no fork creation/block)
rule r_draw_strategy ($symbol in Skind) =
  choose $wr in Row with hasTwo($wr)
  do r_markLastEmpty[$wr,$symbol] //1. Win or 2. Block
  ifnone
    if symbol(5)=EMPTY
    then symbol(5):=$symbol //3. Center
    else if (isCorner(lastMoveX) and symbol(opposite(lastMoveX))=EMPTY)
    then symbol(opposite(lastMoveX)):= $symbol //4. Opposite corner
    else choose $s in Square with (symbol($s)=EMPTY and isCorner($s))
    do symbol($s):= $symbol //5. Empty Corner
    ifnone r_naive_strategy[$symbol] //6. Empty edge
  endif endif

//Computer strategy selection
rule r_tryStrategy ($symbol in Skind) =
  if openingPhase then r_opening_strategy[$symbol]
  else r_draw_strategy[$symbol]
endif

```

the draw, or else to gain a win if X makes a weak play.

For the draw phase (see the *r\_draw\_strategy* rule in Listing 3), the PC try a *draw strategy* with no fork creation or block. Essentially, the computer can play Tic-Tac-Toe if it chooses the move with the highest priority in the following list:

1. Win: you have two in a row, play the third to get three in a row.
2. Block: the opponent has two in a row, play the third to block.
3. Center: Play the center.
4. Opposite Corner: the opponent is in the corner, play the opposite corner.
5. Empty Corner: Play an empty corner.
6. Empty Side: Play an empty edge.

For this example, the function  $\iota_{Tic-Tac-Toe}$  that adds to  $\Gamma_{Tic-Tac-Toe}$  the initialization necessary to make the ASM model executable do not present variability among terminal models (unless one want to start playing from a partially

Listing 4: A winning scenario for player O

```

1  scenario winPC
2  load Tictactoe.asm
3  set playerX := 2;
4  step until status = TURNPC;
5  step until status = TURNX;
6  check symbol(2)=CROSS;
7  check symbol(5)=NOUGHT;
8  set playerX := 1;
9  step until status = TURNPC;
10 step until status = TURNX;
11 check symbol(1)=CROSS;
12 check symbol(3)=NOUGHT;
13 set playerX := 8;
14 step until status = GAMEOVER;
15 check symbol(7)=NOUGHT;
16 check whoWon = PC;

```

full board). In this case,  $\iota_{Tic-Tac-Toe}$  is to be intended as a constant function always producing in the target ASM model the same ASM initial state. One possible, for example, is as follows:

**default init s0:**

```

function symbol($s in Square) = EMPTY
//A polite computer: it allows the user (X) to play first
function status = TURNX
function count = 0

```

#### C. Tic-Tac-Toe semantic validation

The validation of the semantics of the Tic-Tac-Toe case study consists in checking that the mapping function defined in IX-B really captures the intended semantics of the case study language. Among the semantics validation techniques discussed in Section VIII-B, we have used interactive and scenario-based simulation. By interactive simulation, we have used the ASM specification and the AsmetaS simulator to interactively play Tic-Tac-Toe (player vs computer) and check that the ASM model actually captures the desired behavior.

For scenario-based simulation, Listing 4 reports a scenario in Avalla corresponding to the board configurations shown in Fig. 10. In this scenario, the player opens by crossing cell 2 (line 3), the PC responds in the cell 5 (line 7), and the player crosses cell 1. At this point the PC correctly responds by occupying cell 3 (line 12). If the player puts the cross in cell 8 (line 13), the PC takes advantage of that and wins. This scenario shows the smart opening of the PC (as second player) and that the PC is able both to block the player to win and to take advantage of the opportunity to win.

#### D. Tic-Tac-Toe formal verification

Once we were confident that the semantics of the Tic-Tac-Toe as specified really captures the intended behavior, we tried to model and *prove* some formal properties. The first one states that the specification is fair and allows both player to win. To model this fact, we have introduced in the specification the following three temporal properties written in Computational Tree Logic (CTL).

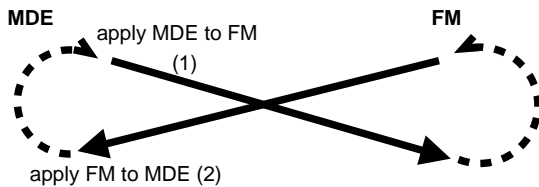


Fig. 11: Closing the in-the-loop integration

```
//the player can win
axiom over CTL: EF(whoWon=PLAYER)
//the computer can win
axiom over CTL: EF(whoWon=PC)
//the match can terminate tie
axiom over CTL: EF(whoWon=TIE)
```

The meaning of  $EF(\phi)$  is given by the E (*exist*) operator which means along at least one path (possibly) and the F operator which means finally: eventually  $\phi$  has to hold (somewhere on the subsequent path). We have automatically proved the three properties via model checking by using the AsmetaSMV component [60].

We wanted also to prove that the match always finishes and we added the following property:

```
axiom over CTL: AF((status = GAMEOVER))
```

It means that on all paths (A) starting from the initial state, *status* will eventually (F) become *GAMEOVER*. This was proved false by the model checker which provided a counter example for it. Analyzing the counter example, we noticed that the player can indefinitely postpone the end of a game by keeping to try to put a cross in an already occupied cell.

## X. CLOSING THE LOOP

This section shows a portion of the definition of the executable semantics of the AsmM metamodel itself by using the ASM-based semantic framework outlined in Sect. IV. We apply the semantic hooking approach on a small portion of the AsmM metamodel concerning the interpretation of the ASM update-rule. In this way, we close the in-the-loop integration between the formal method (ASM) and the MDE framework (EMF), as depicted in Fig. 11.

### A. AsmM semantics

We have to specify, in general, an ASM  $\Gamma_{AsmM}$  (i.e. a model conforming to the AsmM metamodel) containing declarations of functions and domains (the signature) and the behavioral semantics of the AsmM metamodel itself in terms of ASM transition rules.

ASM rule constructors are represented in the AsmM metamodel by subclasses of the class `Rule`. Fig. 12 shows a subset of basic forms of a transition rule under the class hierarchy rooted by the class `BasicRule`: update-rule, conditional-rule, skip, do-in-parallel (block-rule), extend, etc.

Listing 5 reports a fragment  $\Gamma_{AsmM}$  in AsmetaL notation, for the interpretation of an ASM update-rule. It contains domains and function declarations induced from the AsmM metaclasses themselves for static/structural concepts (terms,

rule constructors, etc.). Further domains and functions are introduced to denote run-time concepts like locations, values, updates, etc., according to the theoretical definitions given in [52] to construct the *run* of the ASM model under simulation.

A supporting execution engine has to keep the current state of the ASM model and, on request, evaluates the values of terms and computes (and applies) the update set to obtain the next state. To this purpose, an abstract domain `Value` and its sub-domains are introduced to denote all possible values of ASM terms. The function `eval` computes the value for every term (expression) in the current ASM state. The abstract domain `Location` represents the ASM concept of basic object containers (memory units), named *locations*, abstracting from particular memory addressing and object referencing mechanisms. Functions `sign` and `elements` denote, respectively, the pair of a function name  $f$ , which is fixed by the signature, and an optional argument  $(v_1, \dots, v_n)$ , which is formed by a list of dynamic parameter values  $v_i$  of whatever type, forming a location. Two functions `currentState`, which represents the state of an ASM, and `updateSet`, which represents an update set, are used as tables to denote location-value pairs  $(loc, v)$  (updates) and are the basic units of state change. The `assignment` function maps location variables to their values for variable assignment in a state.

The very crucial task is that of computing at each step the ASM update set. To this purpose, there exist a rule `visit(RuleType R)` for every `RuleType` subclass of the `Rule` class of the `AsmM`. Given a rule `R`, the matching `visit` method is invoked accordingly to the type of `R` to obtain the update set of `R`. As example of such a kind of rule, Listing 5 reports the rule `r_visit` to compute the update set for an update-rule type.

One has also to define a function  $\iota_{PT}$  which adds to  $\Gamma_{AsmM}$  the initialization necessary to make the ASM model executable. Any model transformation tool can be used to automatize the  $\iota_{AsmM}$  mapping by retrieving data from a terminal model  $m$  and creating the corresponding ASM initial state in the target ASM model. A model transformation engine may 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  $\Gamma_{AsmM}$  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.

### B. AsmM semantics validation

We applied the scenario-based approach for the validation of the semantics. We initially collected a set of `AsmetaL` examples representing all ASM constructs. In order to build an extensive set of scenario specifying the expected behavior of the system, instead of writing the scenario by hand, we simulated the original examples with `AsmetaS` (the simulator of `AsmetaL` models, see Sect. VII) itself, parsed the log files produced by `AsmetaS` in order to obtain valid scenario files in the `Avalla` syntax. Then we run the validator with the scenarios and the translation of the input examples by the semantic

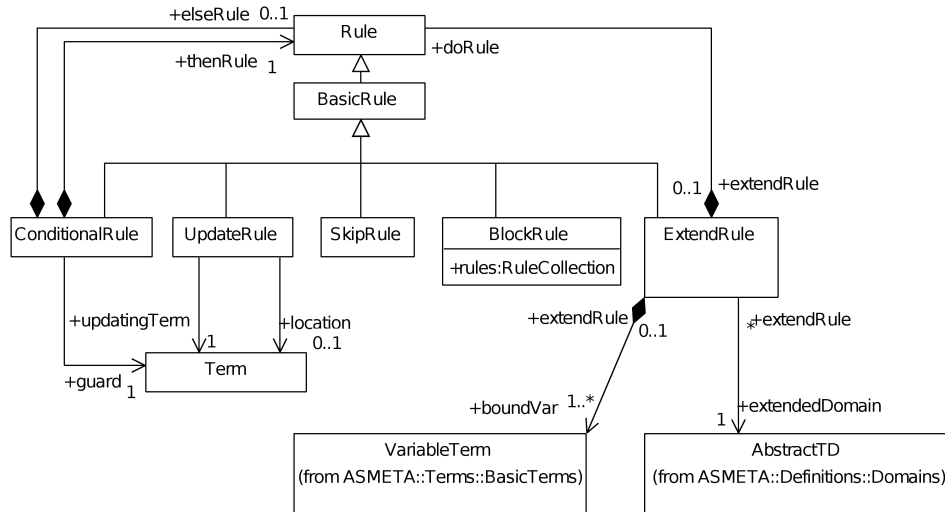


Fig. 12: A fragment of the AsmM metamodel for function terms and update-rules

Listing 5:  $\Gamma_{AsmM}$ 

```

asm AsmM_hooking
signature:
// Signature induced from the AsmM metamodel:
abstract domain Function
abstract domain Term
concrete domain VariableTerm subsetof Term
concrete domain FunctionTerm subsetof Term
concrete domain LocationTerm subsetof FunctionTerm
...
abstract domain Rule
concrete domain UpdateRule subsetof Rule
...
controlled updatingTerm: UpdateRule -> TupleTerm
controlled location: UpdateRule -> Term
...
// Signature for run-time concepts:
abstract domain Value
abstract domain Location
controlled sign: Location -> Function
controlled elements: Location -> Seq(Value)
//Function for the evaluation of ASM terms
static eval: Term -> Value
...
//Functions for the current state of the ASM and memory updates
controlled currentState: Location -> Value
controlled updateSet: Location -> Value
controlled assignment: VariableTerm -> Value
...
definitions:
rule r_visit($r in UpdateRule) =
let ( content = eval(updatingTerm($r)) in
if isLocationTerm(location($r))
then extend Location with $! do
par
sign($!):= funct(location($r))
elements($!):= values(eval(arguments(location($r))))
updateSet($!):= content
endpar
else if isVariableTerm(location($r))
then assignment(location($r)):= content
endif
endif
endlet
...

```

proposed above. In this way we have checked the conformance of AsmetaS with the semantics of the ASM as defined by the hooking function  $M$ .

## XI. 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 interoperability. 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.). They are useful to create, manage and interchange models in a model-driven development context, settling, therefore, a flexible infrastructure for tools development and interoperability. Moreover, metamodeling allows to establish a “global framework” to enable otherwise dissimilar languages (of possibly different domains) to be used in an inter-operable 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* [63], 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 SystemC UML Profile [64]. Formal ASM models obtained from graphical SystemC-UML models can potentially drive practical SoC model analysis like simulation, architecture evaluation and design exploration.

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.

## REFERENCES

- [1] A. Gargantini, E. Riccobene, and P. Scandurra, "Integrating formal methods with model-driven engineering," in *The Fourth International Conference on Software Engineering Advances, ICSEA 2009, 20-25 September 2009, Porto, Portugal*, K. Boness, J. M. Fernandes, J. G. Hall, R. J. Machado, and R. Oberhauser, Eds. IEEE Computer Society, 2009, pp. 86–92.
- [2] J. Bézivin, "On the Unification Power of Models," *Software and System Modeling*, vol. 4, no. 2, pp. 171–188, 2005.
- [3] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
- [4] "OMG. The Unified Modeling Language (UML), v2.1.2," <http://www.uml.org>, 2007.
- [5] "OMG. The Model Driven Architecture (MDA Guide V1.0.1)," <http://www.omg.org/mda/>, 2003.
- [6] S. J. and K. G., "Model-Integrated Computing," *IEEE Computer*, pp. 110–112, 1997.
- [7] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Addison Wesley, 2007.
- [8] "Eclipse Modeling Framework (EMF)," <http://www.eclipse.org/emf/>.
- [9] D. Gasevic, R. Lämmel, and E. V. Wyk, Eds., *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5452. Springer, 2009.
- [10] C. Snook, F. Fritz, and A. Illisaov, "An EMF Framework for Event-B," in *Workshop on Tool Building in Formal Methods - ABZ Conference, 2010*.
- [11] "The Maude System," <http://maude.cs.uiuc.edu/>.
- [12] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, "ATL: a QVT-like transformation language," in *Proc. OOPSLA'06*. ACM, 2006, pp. 719–720.
- [13] A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo, "The design of a language for model transformations," *Software and System Modeling*, vol. 5, no. 3, pp. 261–288, 2006.
- [14] J. Fischer, M. Piefel, and M. Scheidgen, "A Metamodel for SDL-2000 in the Context of Metamodelling ULF," in *Proc. SAM'04*, 2004, pp. 208–223.
- [15] M. Alanen and I. Porres, "A Relation Between Context-Free Grammars and Meta Object Facility Metamodels," *Turku Centre for Computer Science*, Tech. Rep., 2003.
- [16] M. Wimmer and G. Kramler, "Bridging grammarware and modelware," in *Proc. of the 4th Workshop in Software Model Engineering (WiSME'05)*, Montego Bay, Jamaica, 2005.
- [17] T. Gjøsæter, I. F. Isfeldt, and A. Prinz, "Sudoku - a language description case study," in *Proc. SLE'08*, 2008, pp. 305–321.
- [18] "Abstract State Machines tools," <http://www.eecs.umich.edu/gasm/tools.html>.
- [19] Y. Gurevich and B. Rossman and W. Schulte, "Semantic Essence of AsmL," Microsoft Research Technical Report MSR-TR-2004-27, March 2004.
- [20] A. Slissenko and P. Vasilyev, "Simulation of timed abstract state machines with predicate logic model-checking," *J. UCS*, vol. 14, no. 12, pp. 1984–2006, 2008.
- [21] 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.
- [22] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and refinement of textual syntax for models," in *ECMDA-FA*, 2009.
- [23] F. Jouault, J. Bézivin, and I. Kurtev, "TCS: a DSL for the specification of textual concrete syntaxes in model engineering," in *Proceedings of the fifth international conference on Generative programming and Component Engineering (GPCE'06)*, 2006.
- [24] S. Efftinge, "oAW xText - A framework for textual DSLs," in *Workshop on Modeling Symposium at Eclipse Summit*, 2006.
- [25] P.-A. Muller, F. Fondement, F. Fleurey, M. Hassenforder, R. Schneckenburger, S. Gérard, and J.-M. Jézéquel, "Model-driven analysis and synthesis of textual concrete syntax," *Software and System Modeling*, vol. 7, no. 4, pp. 423–441, 2008.
- [26] "OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01," <http://www.uml.org/>.
- [27] D. Hearnden, K. Raymond, and J. Steel, "Anti-Yacc: MOF-to-text," in *Proc. of EDOC*, 2002, pp. 200–211.
- [28] M. Möller, 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.
- [29] J. Armstrong, "Industrial integration of graphical and formal specifications," *J. of Systems and Software*, vol. 40, no. 3, pp. 211–225, 1998.
- [30] A. Idani, J.-L. Boulanger, and L. P. 0002, "A generic process and its tool support towards combining uml and b for safety critical systems," in *Proc. CAINE*, 2007, pp. 185–192.
- [31] T. Zhang, F. Jouault, J. Bézivin, and J. Zhao, "A MDE Based Approach for Bridging Formal Models," in *TASE '08*. IEEE Computer Society, 2008, pp. 113–116.
- [32] Y. Sun, Z. Demirezen, F. Jouault, R. Tairas, and J. Gray, "A model engineering approach to tool interoperability," in *SLE*, 2008, pp. 178–187.
- [33] P.-A. Muller, F. Fleurey, and J.-M. Jezequel, "Weaving Executability into Object-Oriented Meta-Languages," in *Proc. MODELS*, 2005.
- [34] M. Soden and H. Eichler, "Towards a model execution framework for Eclipse," in *Proc. of the 1st Workshop on Behavior Modeling in Model-Driven Architecture*. ACM, 2009.
- [35] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo, "Analyzing rule-based behavioral semantics of visual modeling languages with maude," in *SLE*, ser. Lecture Notes in Computer Science, D. Gasevic, R. Lämmel, and E. V. Wyk, Eds., vol. 5452. Springer, 2008, pp. 54–73.
- [36] K. Chen, J. Sztipanovits, and S. Neema, "Toward a semantic anchoring infrastructure for domain-specific modeling languages," in *EMSOFT*, 2005, pp. 35–43.
- [37] —, "Compositional specification of behavioral semantics," in *DATE*, 2007, pp. 906–911.
- [38] 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.
- [39] M. Anlauff, "XASM - An Extensible, Component-Based ASM Language," in *Proc. of Abstract State Machines*, 2000, pp. 69–90.
- [40] "Atlantic XASM Zoo," <http://www.emn.fr/z-info/atlanmod/index.php/Xasm/>, 2001.
- [41] D. A. Sadilek and G. Wachsmuth, "Using grammarware languages to define operational semantics of modelled languages," in *TOOLS (47)*, 2009, pp. 348–356.
- [42] A. Gargantini, E. Riccobene, and P. Scandurra, "A semantic framework for metamodel-based languages," *Journal of Automated Software Engineering*, vol. 16, no. 3-4, pp. 415–454, 2009.
- [43] 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, 2008, pp. 71–84.
- [44] E. Riccobene and P. Scandurra, "An executable semantics of the SystemC UML profile," in *ABZ 2010*, ser. LNCS, M. F. et al., Ed., vol. 5977, 2010, pp. 75–90.
- [45] E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini, "SystemC/C-based model-driven design for embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 8, no. 4, 2009.
- [46] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Software: Practice and Experience*, vol. 39, no. 15, pp. 1253 – 1292, October 2009.
- [47] J. Bézivin, "In Search of a Basic Principle for Model Driven Engineering," *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, vol. V, no. 2, pp. 21–24, 2004. [Online]. Available: <http://www.upgrade-cepis.org/issues/2004/2/up5-2Bezivin.pdf>
- [48] "Textual Editing Framework." <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef>, 2009.
- [49] "openArchitectureware website," [www.openarchitectureware.org](http://www.openarchitectureware.org), 2009.
- [50] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?" *IEEE Computer*, vol. 37, no. 10, pp. 64–72, 2004.
- [51] E. Börger, "The ASM method for system design and analysis. A tutorial introduction," in *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, Ed., vol. 3717. Springer, 2005, pp. 264–283.
- [52] E. Börger and R. Stärk, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
- [53] 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.
- [54] —, "Ten reasons to metamodel ASMs," in *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis, LNCS Festschrift*. Springer, 2007.

- [55] “The Abstract State Machine Metamodel website,” <http://asmeta.sf.net/>, 2006.
- [56] E. Riccobene and P. Scandurra, “Towards an Interchange Language for ASMs,” in *Abstract State Machines. Advances in Theory and Practice*, ser. LNCS 3052, W. Zimmermann and B. Thalheim, Eds. Springer, 2004, pp. 111 – 126.
- [57] “OMG. Object Constraint Language (OCL), v2.0 formal/2006-05-01,” 2006.
- [58] A. Gargantini, E. Riccobene, and P. Scandurra, “A Metamodel-based Language and a Simulation Engine for Abstract State Machines,” *J. UCS*, vol. 14, no. 12, pp. 1949–1983, 2008.
- [59] —, “Model-driven language engineering: The ASMETA case study,” in *International Conference on Software Engineering Advances, ICSEA*. IARIA: Published by IEEE Computer Society, 2008, pp. 373–378.
- [60] P. Arcaini, A. Gargantini, and E. Riccobene, “AsmetaSMV: A way to link high-level ASM models to low-level NuSMV specifications,” in *ABZ 2010*, ser. LNCS, M. F. et al., Ed., vol. 5977, 2010, pp. 61–74.
- [61] E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, Eds., *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5238. Springer, 2008.
- [62] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking,” in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, July 2002.
- [63] J. M. Rushby, “Harnessing Disruptive Innovation in Formal Verification,” in *Proc. SEFM*, 2006, pp. 21–30.
- [64] A. Gargantini, E. Riccobene, and P. Scandurra, “Model-driven design and ASM-based analysis of embedded systems,” in *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, L. Gomes and J. M. Fernandes, Eds. Norwell, MA, USA: IGI Global, 2009, pp. 24–54.

#### APPENDIX A BASIC PETRI NETS SEMANTICS

A concrete example is here provided by applying the semantic hooking technique to a possible metamodel for the Petri net formalism. The results of this activity are executable semantic models for Petri nets which can be made available in a model repository either in textual form using AsmetaL or also in abstract form as instance model of the AsmM metamodel.

Fig. 13 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. 14 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. 13.

According to the semantic hooking approach, first we have to specify an ASM  $\Gamma_{PT}$  (i.e. a model conforming to the AsmM metamodel) containing only declarations of functions and domains (the signature) and the behavioral semantics of the Petri net metamodel in terms of ASM transition rules. Listing 6 reports a possible  $\Gamma_{PT}$  in AsmetaL notation. It introduces abstract domains for the nets themselves, transitions, and

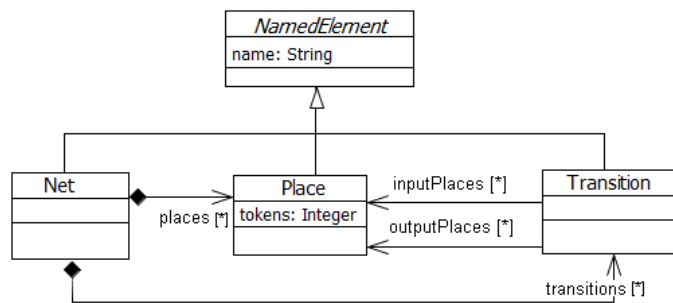


Fig. 13: A metamodel for basic Petri nets

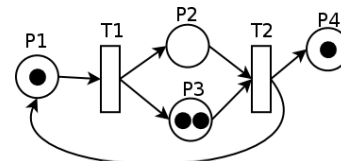


Fig. 14: A basic Petri net with its initial marking

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.

One has also to define a function  $\iota_{PT}$  which adds to  $\Gamma_{PT}$  the initialization necessary to make the ASM model executable. Any model transformation tool can be used to automatize the  $\iota_{PT}$  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  $\Gamma_{PT}$  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  $m_{PT}$  shown in Fig. 14, the  $\iota_{PT}$  mapping would automatically add to the original  $\Gamma_{PT}$  the initial state (and therefore the initial marking) leading to the final ASM model shown in Listing 7. The initialization of the abstract domains *Net*, *Transition*, and *Place*, and of all functions defined over these domains, are added to the original  $\Gamma_{PT}$ .



APPENDIX B  
ASM SPECIFICATION FOR TIC-TAC-TOE

Listing 6:  $\Gamma_{PT}$ 

```
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]
```

Listing 7:  $\iota_{PT}(\Gamma_{PT}, m_{PT})$ 

```
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)
```

Listing 8:  $\Gamma_{Tic-Tac-Toe}$  - the complete signature

```
asm Tictactoe
signature:
//For representing a board
enum domain Skind = {CROSS|NOUGHT|EMPTY}
domain Square subsetof Integer
domain Row subsetof Integer
domain Three subsetof Integer
static squaresInRow: Prod(Row,Three) -> Square
controlled symbol: Square -> Skind
//For managing the game
enum domain Finalres = {PLAYERX|PC|TIE}
enum domain Status = {TURNX|CHECKX|TURNPC|CHECKPC
  |GAMEOVER}

monitored playerX:Square // move of X
controlled status: Status
controlled whoWon: Finalres
derived noSquareLeft : Boolean
derived hasThreeOf: Prod(Row,Skind) -> Boolean
//For PC strategies
domain Count subsetof Integer
controlled count: Count
derived openingPhase: Boolean
controlled lastMoveX: Square
static isCorner: Square -> Boolean
static isEdge: Square -> Boolean
static isCenter: Square -> Boolean
derived hasTwo: Row -> Boolean
static opposite: Square -> Square

definitions:
domain Square = {1..9}
domain Count = {0..9}
domain Row = {1..8}
domain Three = {1..3}

function squaresInRow($r in Row,$x in Three) =
if $r = 1 then if $x = 1 then 1 else if $x = 2 then 2 else 3 endif endif
else if $r = 2 then if $x = 1 then 4 else if $x = 2 then 5 else 6 endif endif
else if $r = 3 then if $x = 1 then 7 else if $x = 2 then 8 else 9 endif endif
else if $r = 4 then if $x = 1 then 1 else if $x = 2 then 4 else 7 endif endif
else if $r = 5 then if $x = 1 then 2 else if $x = 2 then 5 else 8 endif endif
else if $r = 6 then if $x = 1 then 3 else if $x = 2 then 6 else 9 endif endif
else if $r = 7 then if $x = 1 then 1 else if $x = 2 then 5 else 9 endif endif
else if $x = 1 then 3 else if $x = 2 then 5 else 7 endif endif
endif endif endif endif endif endif

function noSquareLeft = not(exist $s in Square with symbol($s)=EMPTY)

function hasThreeOf ($r in Row, $symbol in Skind) =
(symbol(squaresInRow($r,0)) = $symbol) and
(symbol(squaresInRow($r,0)) = symbol(squaresInRow($r,1))) and
(symbol(squaresInRow($r,0)) = symbol(squaresInRow($r,2)))

function openingPhase = count=0 or count=1

function isCenter($s in Square) = $s =5
function isCorner($s in Square) = $s =1 or $s=3 or $s=7 or $s=9
function isEdge($s in Square) = $s =2 or $s =4 or $s=6 or $s=8

//return true iff $r has two equal symbols and the third square is EMPTY
function hasTwo($r in Row) =
(exist $i1 in Three, $i2 in Three, $i3 in Three
with ($i1!=$i2 and $i1!=$i3 and $i2!=$i3 and
(symbol(squaresInRow($r,$i1)) = symbol(squaresInRow($r,$i2))) and
(symbol(squaresInRow($r,$i1)) != EMPTY) and
(symbol(squaresInRow($r,$i3)) = EMPTY)))

function opposite($s in Square) =
if $s=1 then 9 else if $s=3 then 7 else if $s=7 then 3
else if $s=9 then 1 endif endif endif
```

Listing 9:  $\Gamma_{Tic-Tac-Toe}$  transition rules

```

//A very naive player: choose an empty square and mark it.
rule r_naive_strategy ($symbol in Skind)=
  choose $s in Square with symbol($s)=EMPTY
  do symbol($s):= $symbol

rule r_playACorner($symbol in Skind) =
  choose $s in Square with (symbol($s)=EMPTY and isCorner($s))
  do symbol($s):= $symbol

//Opening strategy
rule r_opening_strategy ($symbol in Skind)=
  if (count=0) then r_playACorner[$symbol]
  else if symbol(5) = EMPTY then symbol(5):=$symbol //play the center
  else r_playACorner[$symbol] //we play a corner
  endif endif

//Mark with $symbol the last empty square within row $r
rule r_markLastEmpty ($r in Row, $symbol in Skind) =
  choose $x in {1,2,3} with symbol(squaresInRow($r,$x))=EMPTY
  do symbol(squaresInRow($r,$x)) := $symbol

//Draw strategy (with no fork creation/block)
rule r_draw_strategy ($symbol in Skind) =
  choose $wr in Row with hasTwo($wr)
  do r_markLastEmpty[$wr,$symbol] //1. Win or 2. Block
  ifnone
    if (symbol(5)=EMPTY) then symbol(5):=$symbol //3. Center
    else if (isCorner(lastMoveX) and symbol(opposite(lastMoveX))=EMPTY)
      then symbol(opposite(lastMoveX)):= $symbol //4. Opposite corner
    else choose $s in Square with (symbol($s)=EMPTY and isCorner($s))
      do symbol($s):= $symbol //5. Empty Corner
      ifnone r_naive_strategy[$symbol] //6. Empty edge
    endif endif

//Computer strategy selection
rule r_tryStrategy ($symbol in Skind) =
  if openingPhase then r_opening_strategy[$symbol]
  else r_draw_strategy[$symbol] endif

rule r_movePC = par r_tryStrategy[NOUGHT]
  count := count + 1
  status := CHECKPC
endpar

rule r_movePlayerX = if symbol(playerX)= EMPTY
  then par symbol(playerX):= CROSS
  count := count + 1
  lastMoveX := playerX
  status := CHECKX
  endpar
  else status := TURNX endif

rule r_checkForAWinner($symbol in Skind) =
  //GAME OVER WITH A WINNER?
  if (exist $r in Row with hasThreeOf($r,$symbol)) then
    par status := GAMEOVER
    if $symbol = CROSS then whoWon:= PLAYERX
    else whoWon:= PC endif
  endpar
  else if ( noSquareLeft ) //GAME TIE?
    then par status := GAMEOVER whoWon := TIE endpar
  else if $symbol = CROSS then status:= TURNPC
  else status:= TURNX endif endif endif

main rule r_Main = if status = TURNX then r_movePlayerX[]
  else if status = CHECKX then r_checkForAWinner[CROSS]
  else if status = TURNPC then r_movePC[]
  else if status = CHECKPC then r_checkForAWinner[NOUGHT]
  endif endif endif endif

```