SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

SP&E

A model-driven process for engineering a toolset for a formal method‡

Paolo Arcaini[1], Angelo Gargantini[2], Elvinia Riccobene[1,*] and Patrizia Scandurra[2]

[1] *Università degli Studi di Milano - Italy*
[2] *Università degli Studi di Bergamo - Italy*

SUMMARY

This paper presents a model-driven software process suitable to develop a set of integrated tools around a formal method. This process exploits concepts and technologies of the Model-Driven Engineering (MDE) approach, like metamodelling and automatic generation of software artifacts from models. We describe the requirements to fulfill and the development steps of this model-driven process. As a proof-of-concepts, we apply it to the Finite State Machines and we report our experience in engineering a metamodel-based language and a toolset for the Abstract State Machine formal method.

1 Introduction

The growing complexity of modern systems, their high level of integrity, and the need to guarantee safety and security properties yet at the early stages of the system development can be addressed by using formal methods for system specification and analysis. Formal models can be used to understand if the system under development satisfies the given requirements and guarantees certain properties. By means of abstract models, faults in the specification can be detected as early as possible with limited effort.

However, the practical integration of a formal method within a system development process is often prevented by the lack of tools supporting its use during the different development activities: model editing, simulation, validation, verification, tests generation, etc. Furthermore, when tools are available, it is often the case that they have been developed to cover well only one aspect of the whole system development process, while, at different steps, modelers and practitioners would like to switch tools to make the best of them while reusing information already entered about their models. Tools are usually loosely-coupled, they have their own

*Correspondence to: Elvinia Riccobene, Dipartimento di Tecnologie dell'Informazione, Università degli Studi di Milano, via Bramante, 65 - 26013 Crema (CR)- Italy
‡This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA)

Received 13 April 1999
Revised 23 September 2002

notations and internal representation of models. This makes the integration of tools and the reuse of information hard to accomplish, so preventing a formal notation from being used in an efficient and tool supported manner during the entire software development life-cycle.

This was, for example, our experience with the Abstract State Machine [5] formal method which has been widely used as a system engineering method, from requirements capture to implementation, in different contexts: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, verification of compilation schema and compiler back-ends, etc. The increasing application of the ASM formal method for academic and industrial projects has caused a rapid development of tools around ASMs of various complexity and goals: tools for mechanically verifying properties using theorem provers or model checkers, and execution engines for simulation and testing. Since these ASM tools have been usually developed by different research groups, they are loosely coupled, and have syntaxes and internal representations of ASM models strictly depending on the implementation environment. This makes the encoding of ASM mathematical models not always natural and straightforward and makes the integration of these tools hard to accomplish.

In order to achieve the problem of developing a set of integrated tools around a formal method, in this paper we present a process, based on the Model-Driven Engineering (MDE), which allows developing a family of tools supporting different activities of the development process, from system specification to system analysis, and that are strongly integrated in order to permit reusing information about models during several phases of system life cycle. The process exploits MDE concepts and technologies, like metamodeling and automatic model-to-model and model-to-text transformation. It also facilitates software reuse since several software artifacts are reused by all the tools and it exploits several generation techniques and tools in order to automatically obtain several software artifacts starting from (meta)models.

The application of MDE principles in order to engineer software languages is well established in the context of domain-specific modeling languages [21]. It mainly consists in developing a model (called *metamodel*) to represent the modeling concepts of a language, their relationships, and their use and combination to build models (i.e. the abstract syntax of the language). We here propose to apply the same approach to formal notations as well. This (meta)modeling activity requires a certain effort and a deep understanding of the underlying formal notation. However, this effort is compensated later by the speed and the easiness with which software tools for the formal method can be developed. Indeed, one can automatically derive (through mappings or projections) from a metamodel-based abstract syntax, several different basic *artifacts* that can be reused. In particular, several language concrete notations and grammars can be easily derived or defined. They can be either human-comprehensible (textual and/or graphical) for editing models, and machine-comprehensible (like the XML Metadata Interchange format) for model handling by software applications. Software APIs for model representation in terms of programmable elements can be also easily obtained in a *generative* programming approach.

The proposed process is based on our experience in engineering a metamodel-based language for ASMs and in developing the ASMETA (ASM mETAmodeling) toolset [4, 11] which provides tools for developing, exchanging, and analyzing ASM models. ASMETA is also a framework

for developing new ASM tools and for integrating existing ones. ASMETA is currently used in teaching courses and has been employed in industrial projects.

However, the proposed process is general enough and applicable to any kind of formal method. Therefore, besides briefly reporting our experience in the development of the ASMETA toolset, as a proof-of-concepts, we show the application of the model-driven process to the Finite State Machines (FSMs) case study (all the material can be downloaded from the web site `www.asmeta.sf.net/lemp`). The choice of the FSMs is intentional and due to the fact that this formal method is well-known and concise enough so to permit a complete description and an easy understanding of all the phases of our model-driven process.

The remainder of this paper is organized as follows. The fundamental concepts of the MDE approach for software development are briefly presented in Sect. 2. Sect. 3 presents the overall process of engineering a toolset around a formal method and shows its application to the FSMs case study. Sect. 4 reports our experience in applying the proposed model-driven process to the ASM domain. Lessons learned and benefits gained from our experience in applying this process to ASMs are discussed in Sect. 5. Finally, related work and conclusions are given in Sect. 6 and 7, respectively.

2 Model-Driven Engineering

Model-Driven Engineering (MDE) technologies, with a greater focus on architecture and automation, provide high levels of abstraction in software 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 methodology and it is intended as a way to endow a language or a formalism with an abstract notation, so separating the abstract syntax and semantics of the language from its different concrete notations.

Although the foundation constituents of the MDE as a paradigm are still evolving, some implementations of the MDE principles can be found in different meta-modeling/programming frameworks. The most commonly used are the OMG framework with the MOF (Meta Object Facility) as meta-language, the AMMA metamodeling platform with the KM3 meta-language, the Xactium XMF Mosaic initiative, the Software Factories and their Microsoft DSL Tools, the Model-integrated Computing (MIC), the Generic Modeling Environment for domain-specific modeling, the Eclipse Modeling Framework (EMF) with the Ecore meta-language, and the Eclipse subproject openArchitectureware.

The MDE methodology for engineering software languages is well established in the context of domain-specific languages [21]. The development process of a DSL consists, more or less, of the following four main activities: defining the DSLs core language model to reflect all relevant domain abstractions, defining the behavior of DSL language elements, defining the DSLs concrete syntax(es) by specifying symbols for language model elements and defining DSL production/composition rules, integrating DSL artifacts with the platform/infrastructure by mapping the different artifacts to the target platform. This last activity produces transformations, integration tests, and platform extensions for the DSL.

This model-driven development process can be adapted to a formal method with the overall goal of engineering a language and a set of integrated tools around it. How this adaptation is possible is described in detail in the following section.

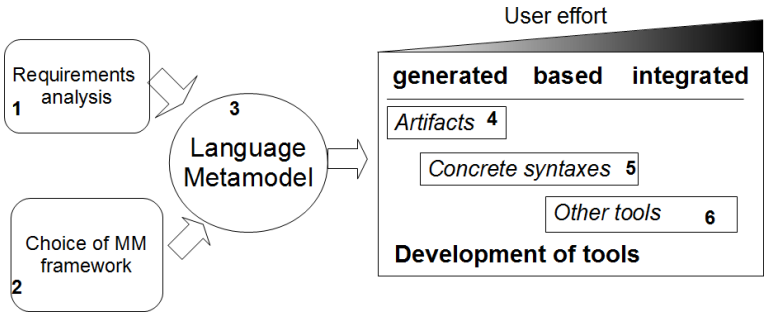


Figure 1. Development process

3 Model-Driven Process for Toolset Development

We here describe the steps of the model-driven process (see Fig. 1) a designer may undertake in order to engineer a set of integrated tools for a formal method, namely tools for model editing, exchange, validation, and verification. The process might require feedback loops and iterative development. To exemplify the process, we show the application of each step to the FSM case study.

3.1 Requirements capture and analysis

During this step (**1** in Fig. 1), concepts and constructs representing the expressive power of the formal method should be clearly pointed out. To this purpose, any official documentation should be taken in consideration, and, if language dialects already exist, it should be make clear if their characteristics have to be included in the new language.

FSM. Many mathematical models for FSMs exist in literature. We choose their formal representation as the tuple $(\Sigma, \Gamma, S, S_0, \tau)$ where Σ is the input alphabet (a finite, non-empty set of symbols), Γ is the output alphabet (a finite, non-empty set of symbols), S is a finite, non-empty set of states, $S_0 \subseteq S$ is a set of initial states, $\tau \subseteq S \times \Sigma \times \Gamma \times S$ is the transition relation such that $(s_j, i, o, s_k) \in \tau$ if the machine is in the state s_j and receives the input i, then it produces the output o and moves to the state s_k.

Many notations for FSMs exist, as the FSMLanguage[†], the State Machine Compiler[‡], the FSMCreator[§], just to name a few. A simple FSM language was also presented in [18]. To represent the above mathematical model, our notation denotes a FSM by: its name, an input and an output alphabet whose symbols are simple characters, a set of named states, and a set

[†]http://hthreads.csce.uark.edu/wiki/FSMLanguage
[‡]http://smc.sourceforge.net
[§]http://www.jugend-weinheim.de/fsm

of named transitions. A state can be an initial state. A transition has a source state, an input character that triggers the transition, an output character, and a target state.

3.2 Choice of a metamodeling framework and supporting technologies

The choice of a specific metamodeling framework (step **2** in Fig. 1) should not in principle prevent the use of models in other metamodeling frameworks, since model transformations (model-to model, model-to-text, etc.) are supported by almost all metamodeling environments. In practice, metamodeling environments do not support all kinds of model transformations in the same way and problems may arise when changing technologies. Therefore, the choice of a metamodeling framework should consider the language artifacts one likes to generate from the metamodel. For example, if one is interested into a concrete textual notation, a framework supporting model-to-text transformations should be selected.

FSM. As metamodeling framework, we selected the EMF since it is based on the extensible, open-source Eclipse framework, it is becoming the standard de-facto MDE platform, and it provides a great variety of supporting technologies and tools.

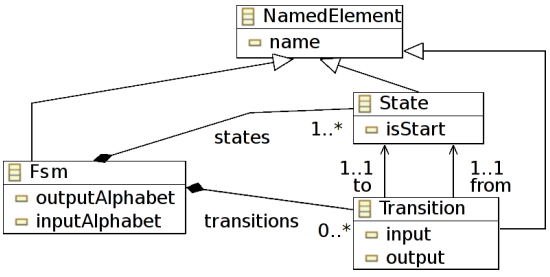
3.3 Design of a specification language for the formal method by metamodeling

During this step (**3** in Fig. 1), the abstract syntax of a specification language is defined in terms of a *metamodel* describing the vocabulary of modeling concepts, the relationships existing among those concepts, and how they may be combined to create models. Possible constraints on the metamodel elements are expressed by the Object Constrain Language (OCL) [17]. Precise guide lines exist (e.g. [21]) 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 further tool development and remains the reference blueprint of the overall development process.

FSM. Figure 2 shows the metamodel for the FSMs using the meta-language EMF/Ecore. Fsm, State, and Transition are subclasses of NamedElement since they all share their attribute name that is the identifier. A Fsm has an input alphabet and an output alphabet as string attributes (since input/output symbols are characters), and consists of a (non-empty) set of states and of a set of transitions. A state can be the starting state. Each transition is labeled by an input and (possibly) and by an output character and is associated with its source and target states.

3.4 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 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.



[tb]

Figure 2. FSM metamodel

3.4.1 Development of language artifacts

From the language metamodel, several *language artifacts* are generated (step **4** in Fig. 1) for model handling – i.e. model creation, storage, exchange, access, manipulatation –, 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 XML-ware for model serialization and interchange, and Java-ware for model representation in terms of programmable objects (through standard APIs).

FSM. The Java APIs and the XMI and XML persistence providers are automatically generated by the EMF/ecore tools.

3.4.2 Definition and validation of language concrete syntax(es)

Language concrete notations (textual, graphical or both) can be introduced (step **5** in Fig. 1) 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 [13] allows defining text syntax for languages described by an Ecore metamodel and it generates an ANTLR grammar file. TCS [14] (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) [23]. Other tools, like the Xtext by openArchitectureWare (now integrated in the Eclipse Modeling Process) [25], 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.

Once 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 of the language metamodel) to check that all of them are covered 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

model set	description	coverage class	coverage line	model set	description	coverage class	coverage line
I1	only FSM declaration	15	24.1	I2	missing states and transitions	25	35.4
C	only states without transitions	55	43.3	R	realistic examples	100	64.5
E1	basic errors	100	65.3	E2	complex errors	100	73.5

Table I. Parser coverage

the parser accordingly. This validation activity is also useful to provide confidence that the metamodel correctly captures concepts and constructs of the underlying formal method.

FSM. In [9], we defined general rules on how to derive a context-free EBNF grammar from a metamodel, and we also provided guidance on how to automatically assemble a script file and give it as input to the JavaCC parser generator to generate a parser for the EBNF grammar of the textual notation. This parser is more than a grammar checker: it is able to process models conforming to their metamodel, to check for their well-formedness with respect to the OCL constraints of the metamodel, and to create instances of the metamodel through the use of the Java APIs.

By using these mapping rules, the following EBNF grammar has been derived from the FSM metamodel in Fig 2 to represent the lexical and syntactical structure of FSM text files:

```
Fsm = "fsm" id "inputAlphabet" string "outputAlphabet" string
      "states" (State)+ "transitions" (Transition)*
State = ["start"] id
Transition = id ":" id "-" char ["/" char] "->" id ";"
```

The terminal symbol *char* represents single characters, *string* represents a string of characters, and *id* represents identifier strings starting with a letter.

For our FSM language we have developed both a EMFText .cs file and a Xtext .xtxt file. The ANTLR parser was also generated by EMFText.

Concerning the syntax validation, we have developed several examples of simple FSMs to check the validity of our grammar. Figure 3 reports two different FSMs taken from the literature [12, 18]. The entire set of benchmarks we used contains about 15 machines, some simple other complex, some correct while others are incomplete or contain syntactical errors. Table I reports the coverage of code in terms of classes and lines measured by the EclEmma Java tool while parsing our examples.

3.5 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 (step **6** in Fig. 1).

FSM. For the FSM language, we have developed two small tools. The first tool is a simple editor *generated* starting from the definition of the grammar by the Xtext Eclipse plugin. The

```
// taken from [19]
// determines if a binary number has an
// odd or even number of zero digits.
fsm evenFsm
inputAlphabet "01"
// Char 'e' for even and 'o' for odd
outputAlphabet "eo"
states
start even odd
transitions
t1: even - 0 / o -> odd ;
t2: even - 1 / e -> even ;
t3: odd - 0 / e -> even ;
t4: odd - 1 / o -> odd ;
```

```
// taken from [12]
// emits 1 when it changes state, 0
// when it remains in the current state
fsm myFSM
inputAlphabet "01"
outputAlphabet "01"
states
start s1 s2
transitions
s1 - 1 / 0 -> s1 ;
s1 - 0 / 1 -> s2 ;
s2 - 0 / 0 -> s2 ;
s2 - 1 / 1 -> s1 ;
```

Figure 3. Example of FSMs

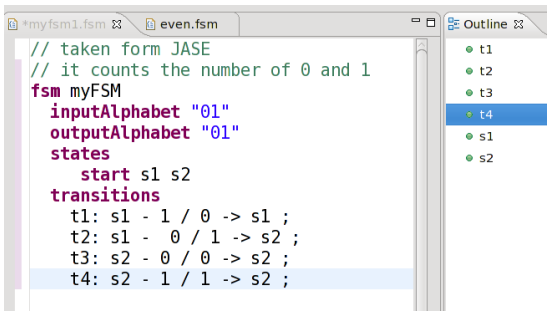


Figure 4. FSM editor

result is shown in Fig. 4. The second tool is a small exporter to the Graphviz tool in order to have a visual pretty-printer for FSMs; it is *based* on the Xpand Eclipse plugin. Fig. 5 shows the Xpand script, the resulting Graphviz .dot file obtained from the *even* FSM, and the resulting picture.

4 The Abstract State Machine and the ASMETA case study

Here we report our experience in engineering a metamodel-based language and a toolset for the Abstract State Machines (ASMs) [5]. ASMs are an extension of FSMs, where unstructured "internal" control states are replaced by states comprising arbitrary complex data. ASM *states* are multi-sorted first-order structures, i.e. domains of objects with functions and predicates

Listing 1. Xpand pretty printer

```
«IMPORT FSM»
«DEFINE Root FOR fsm::FSM»
«FILE "fsm_gv.dot"»
digraph «name» {
  rankdir=LR;
  node [shape = circle]
  «FOREACH states AS a» «a.name»
  «ENDFOREACH» ;
  «FOREACH transitions AS t»
    «t.from.name» -> «t.to.name»
      [label = "«t.name» : «t.input»/
                           «t.output»"];
    «ENDFOREACH»}
  «ENDFILE»
«ENDDEFINE»
```

Listing 2. .dot file

```
digraph even {
  rankdir=LR;
  node [shape = circle] even odd;
  even -> odd [label = "t1 : 0/o"];
  even -> even [label = "t2 : 1/e"];
  odd -> even [label = "t3 : 0/e"];
  odd -> odd [label = "t4 : 1/o"];
}
```

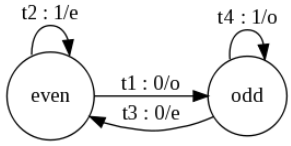


Figure 5. GraphViz exporter for FSMs

defined on them. The *transition relation* is specified by "rules" describing the modification of the functions from one state to the next. ASMs allow modeling single and multi-agents (synchronous and asynchronous) systems, non-determinism and unrestricted synchronous parallelism.

By following the steps of our model-driven design process, we have provided ASM with a set of tools, the ASMs Metamodeling (ASMETA) toolset [4] (see Figure 6), useful for the practical use of this formal method in systems development life-cycle. We have defined concrete syntaxes useful to create, store, access, validate, exchange and manipulate ASM models, and we have built a general framework suitable for developing new ASM tools and for the integration of existing ones [11]. We also have defined a general framework to rigorously specify executable semantics of metamodel-based languages [12].

Requirements capture and analysis. We started collecting all material available on the ASM theory and tool support. As official documentation about the ASM theory, we took [5], but we also considered to include constructs (i.e. particular forms of domains, special terms, derived rule schemes) from other existing notations (like XASM, ASM-WB, AsmGofer, and AsmL) for encoding ASM models.

Choice of a metamodeling framework and supporting technologies. As metamodeling framework, we initially chose the OMG MDA/MOF framework, the mainstream at the time we started. Later, we moved the ASMETA framework to the EMF-Ecore meta-environment for reasons of user support and provided features.

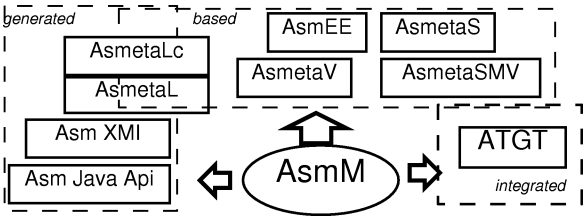


Figure 6. The Asmeta Toolset

Design of a specification language for the formal method by metamodeling. The *Abstract State Machines Metamodel* (AsmM) results into class diagrams representing all ASM concepts and constructs and their relationships. AsmM is available in both MDR/MOF and EMF/Ecore formats, but only the latter is actively maintained. The complete metamodel is organized in one package called `ASMETA` containing 115 classes, 114 associations, and 150 OCL class invariants, approximatively.

Development of *language artifacts*. By exploiting projections from EMF to other technical spaces, from the AsmM we developed in a generative[¶] manner (see Fig.6): an *XMI* interchange format for ASMs, and Java *APIs* to represent ASMs in terms of Java objects. Both formalisms are useful to speed up the tooling activity around ASMs.

Definition and validation of language *concrete syntax(es)*. By exploiting general rules on how to derive a context-free EBNF grammar from a metamodel [9], we derived a platform-independent *textual notation*, AsmetaL (see Fig.6), to write ASM models and a *text-to-model compiler*, AsmetaLc, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints. The AsmetaL and its compiler AsmetaLc can be considered in between generated and based tools, since they were partially derived from the metamodel. By encoding a great number of ASM specifications,we validated the capability of AsmetaL to encode in a natural and straightforward way ASM mathematical models, and ASM specifications written in other ASM notations. We also evaluated the coverage of the metamodel by instrumenting the AsmetaLc compiler with the EclEmma tool. We assured all the metamodel constructs were covered at least once.

Development of other tools. Regarding other ASMETA tools (see Fig.6), as *based* tools, we developed during the last years: a simulator, *AsmetaS* [10], to execute ASM models; a validator, *AsmetaV* [6], with its language *Avalla* to express scenarios, for scenario-based validation of ASM models; a model checker *AsmetaSMV* [3] for model verification by NuSMV; and a graphical front-end, *ASMEE* (ASM Eclipse Environment), which acts as IDE and it is an

[¶]These activities sometimes required some customization of the standard techniques made available from the EMF framework.

Eclipse plug-in. As *integrated* tools, we have modified the *ATGT* [8] tool that is an ASM-based test case generator based upon the SPIN model checker.

5 Lesson learned

On the basis of our experience in developing the AsmM/ASMETA toolset, we believe that a process for building a set of integrated tools around a formal method can gain benefits from the application of a model-driven approach and the use of the MDE automation means. Compared to that one for domain-specific languages definition, the model-driven process applied to a formal method mainly differs for its final goal and, therefore, for the complexity of its activities. For domain-specific languages, it is often the case that one is only interested into defining a language for domain models development. Therefore, the main focus of the process is to abstractedly represent, in terms of a metamodel, the *relevant* domain concepts and their relations in order to define a language for building models. The semantics for this language is usually provided in natural language. For a formal notation, the definition of a metamodel, which requires a certain effort and a deep understanding of *all* the formal concepts and constructs, is only the starting point of a long and complex activity of developing tools for model manipulation and analysis. Indeed, model development is a classical activity regarding the use of a formal method, and by itself it does not require the definition of an abstract syntax of the formal notation. Therefore, developing tools for model handling should be the final goal of a model-driven process for formal methods. Furthermore, while some tools can be easily generated from the metamodel – those that provide a complementary way to represent a model, as tools for model serialization or Java representation –, other tools, especially those for model validation and verification, should correctly capture the semantics of the formal method. This is not trivial to guarantee. It requires a semantic validation process, which is addressed as future work as process improvement.

In general, some significant benefits can be identified in exploiting the model-driven approach for engineering a toolset for a formal method. First, a metamodel may serve as *standard interlingua* establishing a common terminology to discriminate pertinent elements to be discussed, and therefore, helps to communicate understandings, especially if – as in the case of ASMs – the underlying formal method is still evolving and the community is too much heterogeneous to easily come to an agreement on the further development of the method itself.

Second, the metamodel-based approach has the advantage of being suitable to derive from the same metamodel (through mappings or projections) several artifacts (concrete syntaxes, interchange formats, APIs, etc.) that are useful to create, access, transform, manage and interchange models in a model-driven development context, settling, therefore, also a flexible object-oriented infrastructure for tools development and inter-operability. If a designer wants to develop a new tool, he/she can reuse many artifacts already developed in order to ease the development process and to obtain the interoperability with other tools. These features helped us to develop new tools like AsmetaS, AsmetaV, AsmetaSMV with a very limited effort.

Third advantage, we believe important for formal notations, is that people often claim that formal methods are too difficult to put in practice due to their mathematical-based foundation. In this respect, an abstract and (possibly) visual representation, like the one provided by a Ecore-compliant metamodel, delivers a more readable view of the modeling primitives offered by a formal method, especially for people, like students, who do not deal

well with mathematics but are familiar with the Eclipse platform. The AsmM can be, therefore, considered a complementary approach to [5] for the presentation of ASMs.

Furthermore, we like to remark that, although the task of defining a metamodel for a formal notation is not trivial and its complexity closely matches that of the modeling language being considered, the effort of developing from scratch a new EBNF grammar for a complex formalism, like ASMs, would not be less than the effort of defining a EMF-compliant metamodel for ASMs, and then deriving a EBNF grammar from it. Moreover, one has to consider the great possibility of being able to derive, from the same metamodel, different alternative concrete notations, textual or visual or both, for various goals like graphical rendering, model interchange, standard encoding in programming languages, etc.

Finally, metamodeling allows to settle a "global framework" to enable otherwise dissimilar languages (of possibly different domains) to be used in an interoperable manner in different *technical spaces* [15], namely working contexts with a set of associated concepts, knowledge, tools, skills, and possibilities. Indeed, it allows establishing precise *bridges* (or *projections*) among different metamodel-based languages through automatic model transformations. The same techniques can be exploited to achieve inter-operability among different formal notations, once that metamodels for those notations exist.

6 Related work

Concerning the metamodeling technique for language engineering, 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 [20] 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 [24] 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 transformations. Within the Graph Transformation community, using the concepts of graph transformations and metamodeling, the transformation language GReAT (Graph Rewriting And Transformation language) [2] has been designed to address the specific needs of the model transformation area of the Model Integrated Computing. It is supported by a suite of tools that allow the rapid prototyping and realization of transformation tools. A metamodel for the ITU language SDL-2000 has been also developed [7]. 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 we propose in this paper.

Within the ASM community, a number of notations and tools have been developed for the specification and analysis [1]. 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, 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 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 a model-based development process, and, a part 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 for our purposes.

Regarding the derivation of concrete grammars for metamodels, several tools exist: EMFText [13] working for Ecore metamodels, TCS [14] (Textual Concrete Syntax) for metamodels written in KM3, TEF (Textual Editing Framework) for EMF-based metamodels, etc. Viceversa, Xtext [25] allows to derive a language metamodel from the language concrete textual grammar. An overview of textual grammars and metamodel is given in [16].

The challenges of tool integration are discussed in [22], where a software language engineering solution technique is presented that apply MDE principles to address tool interoperability.

7 Conclusions and future work

We here propose a model-driven process to develop a set of integrated tools around a formal method. We give a complete view of all steps, together with their dependency relations, necessary to define a metamodel-based language and manage the tooling activity around a formal notation. We show the application of this process to the simple case study of the Finite State Machine and to the more complex Abstract State Machine formal method.

This model-driven process, based on the MDE concepts of metamodeling and automatic generation of software artifacts from metamodels, has the advantage of being suitable to derive from the same metamodel several language artifacts useful for model exchange and as software support for further tool development. Therefore, it allows providing a flexible infrastructure for tools inter-operability and development. That is in sympathy with the *SRI Evidential Tool Bus idea* [19], and can contribute positively to solve inter-operability issues among formal methods, their notations, and their tools.

We have reached a satisfactory level in the ASMETA toolset development, since we are able to provide model editing and syntax correctness checking, model exchange, simulation and validation, model-based testing, and model checking. Our plan is now to test the applicability of this framework for system development in different application domains, as, for example, service-oriented system, embedded systems, etc. We are also trying to use the ASMETA framework in classes regarding system specification and analysis. On the forehead of the development, in a next future we plan to integrate tools for theorem proving, indeed we are working on translating AmetaL to the language of SAL (Symbolic Analysis Laboratory). Regarding the model-driven engineering process we are working on the following directions: (i) complementing the MDE process for language engineering with an ASMETA-based way

to provide precise executable semantics of metamodel-based languages [12], and (ii) trying to establish a way for tools validation, since tools for formal model analysis have to rigorously reflect the formal method semantics and this must be in some way guaranteed.

REFERENCES

1. Abstract State Machines tools. `http://www.eecs.umich.edu/gasm/tools.html`.
2. A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.
3. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Abstract State Machines, Alloy, B and Z, Second Inter. Conference, ABZ 2010*, volume 5977 of *LNCS*, pages 61–74. Springer, 2010.
4. The Abstract State Machine Metamodel website. `http://asmeta.sf.net/`, 2006.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
6. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A scenario-based validation language for ASMs. In *Abstract State Machines, B and Z, First Inter. Conference, ABZ 2008*, volume 5238 of *LNCS*, pages 71–84. Springer, 2008.
7. J. Fischer, M. Piefel, and M. Scheidgen. A metamodel for SDL-2000 in the context of metamodelling ULF. In *Fourth SDL And MSC Workshop (SAM'04)*, pages 208–223, 2004.
8. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using Spin to generate tests from ASM specifications. In *Abstract State Machines, Advances in Theory and Practice*, number 2589 in LNCS, pages 263–277. Springer, 2003.
9. 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.
10. A. Gargantini, E. Riccobene, and P. Scandurra. A metamodel-based language and a simulation engine for Abstract State Machines. *J. of Universal Computer Science*, 14(12):1949–1983, 2008.
11. A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *Third International Conference on Software Engineering Advances*, pages 373–378, 2008.
12. A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *Journal of Automated Software Engineering*, 16(3-4):415–454, 2009.
13. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *ECMDA-FA*, 2009.
14. 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.
15. I. Kurtev, J. Bézivin, and M. Aksit. Technical spaces: An initial appraisal. In *CoopIS, DOA'2002, Federated Conferences, Industrial track*, Irvine, 2002.
16. 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*, 7(4):423–441, 2008.
17. OMG. Object Constraint Language (OCL), v2.0 formal/2006-05-01, 2006.
18. M. Pfeiffer and J. Pichler. A comparison of tool support for textual domain-specific languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling (DSM' 08)*, pages 1–7, 2008.
19. J. M. Rushby. Harnessing disruptive innovation in formal verification. In *SEFM*, pages 21–30, 2006.
20. C. Snook, F. Fritz, and A. Illisaov. An EMF Framework for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference*, 2010.
21. M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15), October 2009.
22. Y. Sun, Z. Demirezen, F. Jouault, R. Tairas, and J. Gray. A model engineering approach to tool interoperability. In *SLE*, pages 178–187, 2008.
23. Textual Editing Framework. `http://www2.informatik.hu-berlin.de/sam/meta-tools/tef`.
24. The Maude System. `http://maude.cs.uiuc.edu/`.
25. Xtext. `http://www.eclipse.org/Xtext//`, 2010.