LEMP: a Language Engineering Model-driven Process

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

[1] Università di Bergamo, Italy
{angelo.gargantini,patrizia.scandurra}@unibg.it
[2] Università degli Studi di Milano, Italy
elvinia.riccobene@unimi.it

Abstract. In this paper, we propose LEMP as a model-driven process to develop a language endowed with a set of derived artifacts (syntax, interchange format, APIs, ...) and with a well defined formal semantics. The process exploits the Model Driven Engineering principles of metamodeling, model transformation and automatic generation of language processing tools. We describe the requirements to fulfill and the development steps of this language engineering life cycle, including the validation activities regarding the syntactic and semantic aspects. As a proof-of-concepts, we apply LEMP to the Finite State Machines and we report our experience in developing a language for the Abstract State Machine formal method.

Key words: software language, language engineering, model-driven engineering, metamodeling, language artifacts development, semantics specification and validation

1 Introduction

In the context of (software) language engineering, Model-driven Engineering (MDE) [5, 25, 36] is beginning to take a more prominent role, thanks to its basic concepts of *models*, which are considered as first class artifacts of the development process, and of automatic *model transformations*, which drive the overall design flow from requirements elicitation till final implementation toward specific platforms. We refer to *model-driven language engineering* when language development is carried out following the principles of the model-driven approach.

Among software languages, a distinction can be made between *programming languages*, which are used to develop software code running on a given platform and satisfying certain computational paradigms, and *modeling languages*, which are used for high level, platform-independent software design and that are increasingly being defined (as *domain-specific languages* or DSLs) for specific domains of interest.

Traditionally, programming language construction follows a well-defined path [2], which consists of (1) defining the language syntax by means of an EBNF grammar, (2) generating a parser, (3) defining a type-system, (4) developing algorithms that walk the abstract syntax tree and check the well-typedness of the

program. On the contrary, a modeling languages development process diverges from traditional language design [23] since modeling languages are usually introduced to model specific domain concepts, should be easy and fast to define, and should allow re-use of previously defined artifacts. Modeling language syntaxes are expressed in an abstract way by means of (usually object-oriented) models, called *metamodels*, so separating the abstract syntax and semantics of the language constructs from their different concrete notations. Language syntaxes should be suitable for different applications: *model transformations*, to change design abstraction level, compose modeling aspects, relate platform-independent models with platform specific ones; *design space exploration*, to generate optimized variants of models; and *correct-by construction design*, where expressive syntactic rules are used in conjunction with a suitable behavioral semantics to statically identify models with bad behavioral properties. Furthermore, a way to express and prove *structural* and *behavioral* formal *properties* of models should be provided. Structural properties concern the specification, representation, and manipulation of models as expressed in some (domain-specific) syntax. Behavioral properties focus on the specification and analysis of (domain-specific) execution semantics.

A process to engineer modeling languages must, therefore, address several aspects of a language: structure, constraints, textual and graphical representation, parser/compiler, transformational and executable behavior. Up to now, research usually faced on one aspect at a time: definition of (domain-specific) language metamodels [22, 38, 33, 12], development or derivation of concrete syntaxes (textual or graphical) from metamodels [31], model implementation, etc. And whilst many metamodelling environments (Eclipse/Ecore, GME/MetaGME, AMMA/-KM3, XMF-Mosaic/Xcore, etc.) exist that allow to cope with syntactic and transformation definition issues, very limited effort was spent for language semantics definition, which is usually given in natural language. However, a precise formalization of model behavior is strongly required for the purposes of verification and simulation [23], and it still remains an open issue in the context of modeling language definition [3].

The definition of a language development methodology covering all aspects mentioned above is still missing, and the lack of such a process is addressed in [29] as the reason why the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage. A well-established language engineering process, should organize all the development activities by setting precise (sequential or parallel) relations among them on the base of their input/output artifacts. The definition of such an engineering language process can take advantage of the traditional programming language development process, but it can not ignore the different goal, nature, and artifacts of modeling languages w.r.t. programming languages.

In this paper, we propose LEMP (Language Engineering Model-driven Process) as a process to develop a language endowed with a set of derived artifacts (syntax, interchange format, APIs, etc.) and a well defined formal semantics. The process exploits the MDE principles of metamodelling, model transforma-

tion and automatic generation of language processing tools. The novelty aspects of LEMP are (a) the completeness of the language development process from requirements to final artifacts implementation, and (b) a validation activity, regarding both syntactic and semantic aspects, to asses language correctness.

LEMP is the result of our experience in applying the principle of the model-driven approach to develop a unified abstract notation for the Abstract State Machine (ASM) formal method [6], and to tackle the problem of ASM tool inter-operability. This effort brought us to the development of the ASMETA (ASM mETA modeling) tool-set [4, 16], a set of tools around ASMs, and to the definition of a general framework to endow metamodel-based languages with a formal semantics [17].

As a proof-of-concepts, we show the application of the LEMP activities by developing a language for the Finite State Machines (FSMs), and then we briefly report our experience in the development of the ASMETA tool-set. The choice of the FSMs is intentional and due to the fact that a language for FSMs is easy to understand, small enough that a complete description of the LEMP steps is feasible to be presented in a small room, and covers all language aspects mentioned above.

The remainder of this paper is organized as follows. Sect. 2 introduces basic concepts underlying ASMs. Sect. 3 presents the overall process of engineering a metamodel-based language by LEMP and shows its applycation to the FSMs case study. Sect. 4 reports our experience on applying the LEMP process to the ASM domain. Finally, related work and conclusions are given in Sect. 5.

2 Abstract State Machines

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

ASMs can be understood as pseudo-code or virtual machines working over abstract data structures. The notion of ASMs moves from a definition which formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, and in a structured and recursive way, *Structured or Turbo ASMs*, to a generalization where multiple agents interact *Multi-agent ASMs*. Appropri-

---

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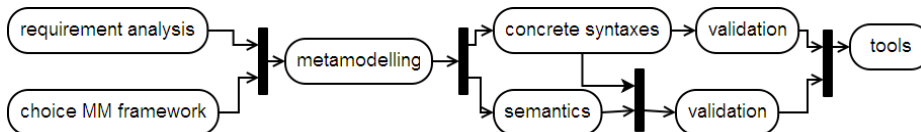
Fig. 1. The LEMP process

ate rule constructors also allow non-determinism (existential quantification) and unrestricted synchronous parallelism (universal quantification).

A complete mathematical definition of the ASM method can be found in [6], together with an overview of the great variety of its applications like: definition of industrial standards for programming and modeling languages, design and re-engineering of industrial control systems, modeling e-commerce and web services, design and analysis of protocols, architectural design, verification of compilation schema and compiler back-ends, etc.

3   A language engineering model driven process: LEMP

A language definition process should specify the steps a designer has to under-take in order to provide the essential ingredients of a language, namely (at least) a *syntax*, a *semantics* and a set of *tools* for language processing. The language engineering process we here present exploits the basic concepts of (a) metamod-eling, (b) model transformations, (c) automatic generation of code from models of the model-driven engineering approach to system development.

The overall LEMP (Language Engineering Model-driven Process) process for developing a metamodel-based language is depicted in Fig. 1. The diagram shows the process activities and makes explicit their dependency. LEMP consists of the following steps:

1. language requirements capture and analysis;
2. choice of a metamodeling framework and supporting technologies;
3. language design by metamodeling;
4. definition of language *concrete syntaxes*, i.e. metamodel derivatives (also called *language artifacts*), to handle – i.e. create, store, control, exchange, access, manipulate — models;
5. validation of the language concrete syntaxes;
6. language semantics definition;
7. validation of the language semantics;
8. development of language processing tools, exploiting the chosen metamod-eling framework and the language artifacts

The LEMP process may be iterative: often it has to come back to previous steps and make corrections. The activities of defining language concrete syntaxes and semantics, together with their subsequent validation steps, can, in princi-ple, be performed in parallel. However, a synchronization is required by LEMP

between the syntax definition and the semantic validation, since, as better explained in Sect. 3.7, the latter requires the availability of models to simulate that have to be expressed in terms of a concrete syntax.

3.1 Language requirements analysis

This design step consists in pointing out the language computation paradigm and the constructs representing the expressive power of the language. To this purpose, any official documentation of the language 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, or not.

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[2], the State Machine Compiler[3], the FSMCreator[4], just to name a few. A simple FSM language was also presented in [32]. 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 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

Many meta-modeling frameworks implementing the model-driven engineering principles exist in literature. Among them, the most commonly used are the OMG framework with the MOF (Meta Object Facility) as meta-language, the AMMA metamodeling platform, 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, and the Eclipse Modeling Framework (EMF).

In principle, the choice of a specific meta-modeling framework should not prevent the use of models in other different meta-modeling spaces, since model-transformations among meta-modeling framework should be theoretically supported by the environments. However, although in theory one could switch framework later, a commitment with a precise metamodeling framework has to be done very early in the development process. Indeed, at the light of our experience, model transformations (model-to model, model-to-text, etc.) are not supported by all metamodeling environments in the same way, and problems can arise upon context change. We suggest that the choice of the metamodeling frame-

---

[2] http://hthreads.csce.uark.edu/wiki/FSMLanguage
[3] http://smc.sourceforge.net
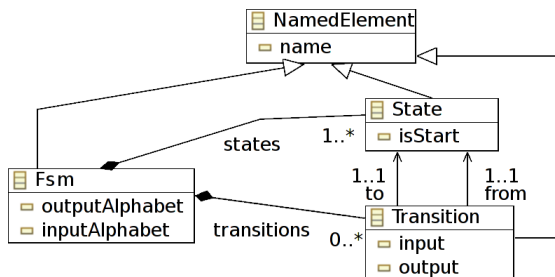[4] http://www.jugend-weinheim.de/fsm

Fig. 2. FSM metamodel

work should be prescribed by the language artifacts one likes to generate from the metamodel. For example, if the language designer is interested into developing a concrete notation in textual form, he/she must prefer a framework where model-to-text transformations are well-mastered.

FSM. As metamodeling framework, we commit with 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   Language design by metamodeling

In a model-driven approach, the abstract syntax of a language is defined in terms of a *metamodel* describing the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models. This step leads to an instantiation of the chosen metamodeling framework for a specific domain of interest.

A metamodel based abstract syntax definition has the great advantage of being suitable to derive from the same metamodel (through mappings or projections) different alternative concrete notations, textual or graphical or both, for various scopes like graphical rendering, model interchange, standard encoding in programming languages, and so on, still maintaining the same semantics. Therefore, a metamodel could be intended as a standard representation of the language notation.

FSM. Figure 2 shows the metamodel for the FSMs we propose. 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   Language concrete syntaxes

Whenever a language is specified in terms of a metamodel, it is possible to automatically (or semi-automatically) generate several concrete syntaxes – here

referred to us as *language artifacts*. Each one is defined by exploiting standard or proprietary projections from the metamodeling framework to other technical spaces [26], as for ex. the Javaware, XMLware, and grammarware spaces.

Language concrete notations can be classified as *human-comprehensible* (textual, graphical or both) for human use to edit models conforming to the metamodel, and as *machine-comprehensible* for model handling by software applications. An XMI (XML Metadata Interchange) format and Java APIs are examples of machine-comprehensible concrete syntaxes. The former allows serializing language models and it is necessary for an easy interchange of data and metadata between tools; the latter allow model representation in terms of programmable objects, and they are useful to develop language processing tools able to access and manipulate models in a model repository.

FSM. In [14], 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.

There exist several tools able to define (or derive) concrete grammars for metamodel-based languages. For example, EMFText [21] allows the user to define text syntax for languages described by an Ecore metamodel and it generates an ANTLR grammar file. TCS [24] (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)[5]. All these approaches in which the abstract syntax (metamodel) is defined before the concrete syntax are completely compatible with the LEMP approach. Other tools, like the Xtext by openArchitectureWare[6], feature the derivation of the language metamodel from its concrete textual grammar. They can be integrated with LEMP provided that the user compares the metamodel generated from the grammar with the metamodel developed at stage 3 to find inconsistencies. For

---

[5] http://www2.informatik.hu-berlin.de/sam/meta-tools/tef
[6] http://www.openarchitectureware.org/

our FSM language we have developed both a EMFText .cs file and a Xtext .xtxt file, which can be downloaded from the web site `www.ameta.sf.net/lemp`.

3.5   Syntax Validation

The first goal of the validation phase is to validate the abstract syntax obtained by the metamodeling activity. Besides a manual inspection of the metamodel, possibly performed by experts of the language, LEMP proposes to validate the abstract syntax by validating the concrete syntaxes to provide confidence that the metamodel correctly captures the language constructs. We believe that validating the abstract syntax without any concrete syntax requires a greater effort and is more error prone than developing some concrete syntaxes and use them. The validation is performed by instantiating several *terminal* models[7]. The best way to carry on this activity is to use a concrete textual syntax and its parser to read a model pool that would act as benchmark. Using the APIs and programmaticaly instantiate models in the language is another alternative. We assume that a textual syntax has been defined, a syntax checker has been introduced that checks the conformance of the model with respect to the metamodel constraints (for example by checking that the OCL constraints are satisfied or not) and the models are written as text files. During this activity it is important that the user collects the information about the coverage of language constructs to check that all the language constructs (classes, attributes and relations) are actually used by the examples. Writing wrong models and checking that they are not accepted by the parser is important as well. The coverage evaluation can be performed by using a code coverage tool and instrumenting the parser accordingly.

FSM. We have developed several examples of simple FSMs to check the validity of our grammar. Table 3 reports two different FSMs taken from the literature [17, 32]. The entire set of benchmarks we used contains about 15 machines, some simple other complex, some corrected while others are incomplete or contain syntactical errors. Table 1 reports the coverage measured by the EclEmma Java tool while parsing our examples.

model set	description	coverage	model set	description	coverage
I1	only FSM declaration	24.1	I2	missing states and transitions	35.4
C	only states without transitions	43.3	R	realistic examples	64.5
E1	basic errors	65.3	E2	complex errors	73,5

Table 1. Parser coverage

---

[7] According to the definition in [27], a *terminal model* is a model written in the language L and *conforming* to the language metamodel.

```
// taken from [32]                    // taken from [17]
// determines if a binary number has an   // emits 1 when it changes state, 0
// odd or even number of zero digits.      // when it remains in the current state
fsm evenFsm                           fsm myFSM
inputAlphabet "01"                    inputAlphabet "01"
// Char 'e' for even and 'o' for odd  outputAlphabet "01"
outputAlphabet "eo"                   states
states                                start s1 s2
start even odd                        transitions
transitions                           s1 − 1 / 0 −> s1 ;
t1: even − 0 / o −> odd ;             s1 − 0 / 1 −> s2 ;
t2: even − 1 / e −> even ;            s2 − 0 / 0 −> s2 ;
t3: odd − 0 / e −> even ;             s2 − 1 / 1 −> s1 ;
t4: odd − 1 / o −> odd ;
```

Fig. 3. Example of FSMs

3.6   Language semantics

The definition of a means for specifying language semantics rigorously and natively within their metamodels is currently an open and crucial issue in the MDE context. We believe this goal can be achieved by integrating metamodeling techniques with formal methods providing the requested and lacked rigor and preciseness [17]. LEMP provides a way to define metamodel-based language semantics (possible *executable*) by using a formal *semantic framework* based on the ASM formal method [17].

A language L has a well-defined semantics if a semantic domain S is identified and a semantic mapping M_S from the language metamodel A to S is provided [20]. As semantic domain S, we assume the semantic domain S_{AsmM} of the ASM language, namely the first-order logic extended with a logic for function updates and for transition rule constructors formally defined in [6]. Therefore, the semantic mapping $M_S : A \rightarrow S_{AsmM}$ which associates a well-formed terminal model m conforming to A with its semantic model $M_S(m)$, can be defined as

$$M_S = M_{S_{AsmM}} \circ M$$

where $M_{S_{AsmM}}$ is the semantic mapping (of the ASM language) that associates a theory conforming to the S_{AsmM} logic with a model conforming to $AsmM$ (the metamodel of the ASM language), and the function $M : A \longrightarrow AsmM$ associates an ASM to a terminal model m. The function M *hooks* the semantics of a metamodel to the S_{AsmM} domain and, therefore, the problem of giving the language semantics is reduced to define the function M as shown in Fig. 4. Formally, the *hooking function M* is given by

$$M(m) = \iota_A(\Gamma_A, m)$$
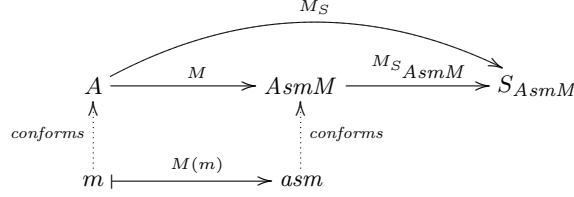
for all m conforming to A, where:

$$A \xrightarrow{\quad M \quad} AsmM \xrightarrow{\quad M_{S\,AsmM} \quad} S_{AsmM}$$

with M_S arc from A to S_{AsmM}, $conforms$ vertical arrows, and $m \xmapsto{\;M(m)\;} asm$

Fig. 4. The hooking function M

– Γ_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;

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

By exploiting this approach, the semantics of a metamodel-based language L is expressed in terms of ASM transition rules. The result of this activity is therefore an executable semantic model for the input language which can be make available in a model repository either in textual form using AsmetaL or also in abstract form as instance model of the AsmM metamodel.

FSM. Listing 1.1 reports a possible Γ_{FSM} for the FSM metamodel in Fig.2. It introduces an abstract domain for the FSMs themselves (the `Fsm` domain), transitions and states. Further signature elements (in addition to those inferred from the FSM metamodel), a controlled function `currentState` store the current state of a FSM, a monitored function `currentInput` provides the input events to a FSM, while the out function `currentOutput` returns the output of a FSM. The behavior of a generic FSM is given by the two rules: `r_run`, for non-deterministically choosing an *enabled* transition (i.e. a transition whose source state is the current state and the input event of the transition matches the present input) to fire, and `r_fire`, for the effective firing of the selected transition. The main rule executes all machines in the `Fsm` domain.

One has also to define a function ι_{FSM} which adds to Γ_{FSM} the initialization necessary to make the ASM model Γ_{FSM} executable. Any model transformation tool can be used to automatize the ι_{FSM} mapping. Fig. 5 shows the complete scenario using the ATL model transformation engine: for any terminal FSM model `myFSM`, the ASM model Γ_{FSM} is refined into the target ASM model Γ_{FSM}_`myFSM` by retrieving data from `myFSM` and creating the corresponding ASM initial state in the ASM model Γ_{FSM}. Essentially, for each class instance of the terminal model `myFSM`, a static 0-ary function is created in the signature of the ASM model Γ_{FSM} 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 *even* automaton shown in Fig. 3, the provided ι_{FSM} mapping would automatically add to the original Γ_{FSM} the initial state partially shown in Listing 1.2. The initialization of the abstract domains `Fsm`, `Transition`, and `State`, and of all functions defined over these domains, are added to the original Γ_{FSM}. All files (the ATL transformation, models and metamodels) involved in this transformation scenario are available for download at `www.ameta.sf.net/lemp`.

Listing 1.1. Γ_{FSM}

```
asm gamma_FSM //Semantic hooking: gamma for the FSM metamodel
import StandardLibrary
signature:
//Signature derived automatically from the FSM metamodel:
      abstract domain NamedElement
      dynamic domain Fsm subsetof NamedElement
      dynamic domain State subsetof NamedElement
      dynamic domain Transition subsetof NamedElement

      //Functions on NamedElement
      controlled name : NamedElement−>String

      //Functions on Fsm
      controlled inputAlphabet: Fsm −> String
      controlled outputAlphabet: Fsm −> String
      controlled states: Fsm −> Powerset(State)
      controlled transitions: Fsm −> Powerset(Transition)

      //Functions on State
      controlled isStart: State −> Boolean

      //Functions on Transition
      controlled input: Transition −> String
      controlled output: Transition −> String
      controlled from: Transition −> State
      controlled to: Transition −> State

//Added signature:
      controlled currentState: Fsm −> State
      monitored currentInput: Fsm −> String
      out currentOutput: Fsm −> String

definitions:
 rule r_fire ($m in Fsm, $t in Transition) = par
                                              currentOutput($m) := output($t)
                                              currentState($m):= to($t)
                                     endpar

 rule r_run ($m in Fsm)= choose $t in transitions($m)
                         with input($t)=currentInput($m) and currentState($m)=from($t$)
                         do r_fire[$m,$t]

 main rule r_Main = forall $m in Fsm do r_run[$m]
```
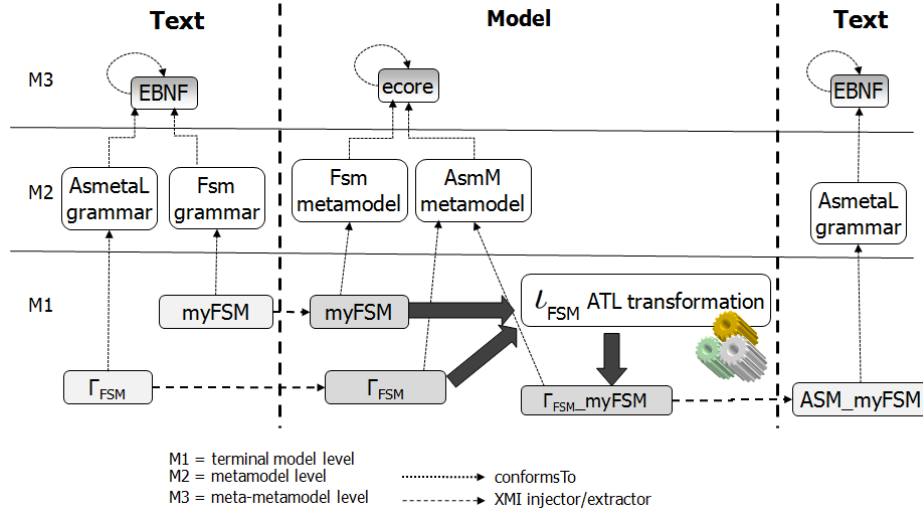
Listing 1.2. ASM for a terminal FSM model by hooking

```
asm ASM_evenFsm
signature:
   ....
   static evenFsm : Fsm
   static even:State
   static odd:State
   static t1:Transition
   static t2:Transition
```

Fig. 5. ι_{FSM} transformation scenario

```
      static t3:Transition
      static t4:Transition
      ....
default init s0:
 //Functions on NamedElement
 function name($e in NamedElement) = at({evenFsm−>"evenFsm",even−>"even",odd−>"odd",
                                         t1−>"t1",t2−>"t2",t3−>"t3",t4−>"t4"}, $e)

 //Functions on Fsm
 function inputAlphabet($m in Fsm) = at({evenFsm −> "01"},$m)
 function outputAlphabet($m in Fsm) = at({evenFsm −> "oe"},$m)
 function states($m in Fsm) = at({evenFsm−>{even,odd}},$m)
 function transitions($s in Fsm)= at({evenFsm−>{t1,t2,t3,t4}},$s)

 //functions on State
 function isStart($s in State) = at({even−>true,odd−>false},$s)

 //Functions on Transition
 function input($t in Transition) = at({t2−>"1",t1−>"0",t3−>"0",t4−>"1"},$t)
 function output($t in Transition) = at({t2−>"e",t1−>"o",t3−>"e",t4−>"o"},$t)
 function from($t in Transition) = at({t2−>even,t1−>even,t3−>odd,t4−>odd},$t)
 function to($t in Transition) = at({t2−>even,t1−>odd,t3−>even,t4−>odd},$t)

 function currentState($m in Fsm) = chooseone({$s in states($m) | isStart($s) : $s})
```

3.7   Semantics Validation

The ASM-based semantic framework supports language semantic validation by
exploiting the ASMETA tool-set (simulator, validator, etc.), a set of tools around
the ASMs (see Sect. 4 for details). Indeed, semantic validation is performed in
LEMP through the validation of the hooking function M presented in Section 3.6
by applying it to a collection of meaningful examples. The ASM models obtained
form the application of M to the examples can be validated in different ways
providing increasing degrees of confidence in the semantics correctness. A simple
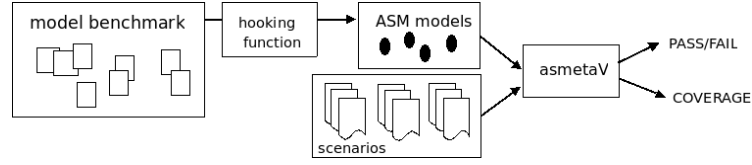
Fig. 6. Semantic validation by AsmetaV

validation can be performed by *randomly simulating* the ASM models with the ASM simulator (see Sect. 4) to check 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* [7] by the ASM validator (see Sect. 4). As shown in Fig. 6, 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 (see Sect. 4), or alternatively, if the language L has already a simulator L_S, these scenarios may be derived from the execution traces generated by L_S. 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.

FSM. A scenario example for the *even* FSM given in Fig. 3, is reported below. It was useful to discover a fault in a preliminary verision of Γ_{FSM}, namely the missing control of the currentState in the r_run rule.

```
scenario test1
load ASM_evenFsm.asm
check name(currentState(evenFsm)) = "even"; // check "even" as currentState
set currentInput(evenFsm) := "0" ; // provide "0" as input
step // perform an execution step
check name(currentState(evenFsm)) = "odd"; // check "odd" as the target state
```

3.8   Development and integration of tools

The metamodel and the language artifacts establish the foundations over which new tools can be developed. LEMP classifies the artifacts and tools in: *generated*, *based*, and *integrated*, as shown in Fig. 7, depending on the decreasing use of generative technologies the designer exploits to develop them. Generally, the effort required by the user increases, too.

Generated artifacts/tools are derivatives obtained (semi-)automatically by applying appropriate projections to the technical spaces Javaware, XMLware, and grammarware. The language concrete syntaxes presented in Sect 3.4 are considered derivatives.
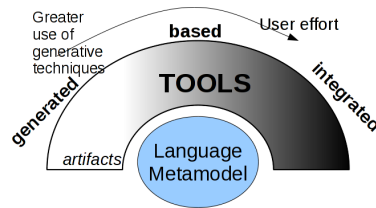
Fig. 7. Language processing tools

Based tools are those developed exploiting the metamodeling environment and related derivatives; they may use the APIs and other concrete syntaxes, but they also contain a considerable amount of code that has not been generated. An example of such a tool may be a tool that reads the language models in XMI format, uses the APIs to access the model data, but then it performs some operations written in Java code from scratch.
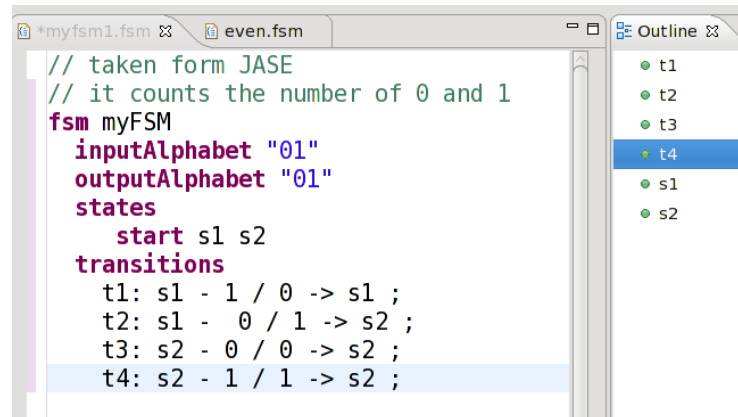
Integrated tools are external and existing tools that are connected to the language artifacts. The integration can be realized at different levels: a tool may use just the XML interchange format, other tools may take advantage of the APIs or of other derivatives. A goal of LEMP is to support an open and flexible architecture to make easier the development of new tools and the integration with other existing tools. If the 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.

FSM. For the FSM language, we have developed four artifacts: the Java APIs, the ANTLR parser, and two small tools. The first tool is a simple editor *generated* starting from the definition of the grammar by the Xtext eclipse plugin. The result is shown in Fig. 8. The second tool is a small exporter to graphviz developed in order to have a visual pretty-printer for FSMs; it is *based* on the Xpand eclipse plugin. Fig. 9 shows the Xpand script, the resulting graphviz .dot file obtained form the *even* FSM, and the resulting picture.

4   The ASMETA Case study

Here we report our experience in engineering a metamodel-based language for the Abstract State Machines. By following the steps suggested by the LEMP design process, we have developed the ASMs Metamodeling (ASMETA) framework [4]. 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 [16]. We also have defined a general framework to rigorously specify executable semantics of metamodel-based languages [17].

Language requirements analysis. We started collecting all material available on the ASM theory and tool support. As official documentation about the ASM theory, we took [6] and, in order to design a language that could serve as a

Fig. 8. FSM editor

standard interlingua for the ASM specific domain of interest, we considered to include constructs (i.e. particular forms of domains, special terms, derived rule schemes) taken from ASM dialects.

Choice of a metamodeling framework 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 [10] meta-environment for the reasons explained in Sect. 3.2.

Language design by metamodeling The *Abstract State Machines Meta-model* (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.

Language concrete syntaxes By exploiting projections from EMF to other technical spaces, several *concrete syntaxes* have been obtained automatically (or semi-automatically) in a generative[8] manner from the AsmM.

As *machine-comprehensible* notations, we derived (a) an *XMI* interchange format for ASMs, and (b) Java *APIs* to represent ASMs in terms of Java objects. Both formalisms are useful to speed up the tooling activity around ASMs.

As *human-comprehensible* notations, we derived (c) a *textual notation*, called AsmetaL (ASMETA Language)[9], and its parser; (d) a *graphical notation*, to represent ASM models in a visual form, by means of the Eclipse Graphical

---

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

[9] The EBNF (Extended Backus-Naur Form) grammar of the AsmetaL textual notation can be found in [4]

Listing 1.3. Xpand pretty printer

Listing 1.4. .dot file

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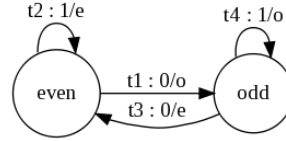
```
digraph even {
  rankdir=LR;
  node [shape = circle] even odd;
  even −> odd [label = "t1 : 0/o"];
  even −> even [label = "t2 : 1/e"];
  odd −> even [label = "t3 : 0/e"];
  odd −> odd [label = "t4 : 1/o"];
}
```


Fig. 9. GraphViz exporter for FSMs

Modeling Framework (GMF) technology [1] that allows to derive graphical model editors from metamodels.

Syntax Validation By encoding a great number of specifications (up to now we have about 140 ASM specification files in [4]), we validated the expressive power of AsmetaL, namely its capability of encoding in a natural and straightforward way not trivial ASM mathematical models. We also validated the capability of AsmetaL to textually represents ASM specifications written in different ASM dialects. Moreover, we evaluated the coverage of the metamodel by instrumenting the parser of AsmetaL with EclEmma and by parsing all the examples. We checked that all the metamodel constructs were covered at least once.

Language semantics definition and validation Following the semantic hooking approach described in Sect. 3.6, we have to specify an ASM Γ_{AsmM} containing declarations of functions and domains (the signature) and the behavioral semantics of the AsmM metamodel itself in terms of ASM transition rules. This work is still in progress. To validate the semantics, as suggested by LEMP, we have to apply the scenario-based approach for the validation of the semantics. Since the language semantics definition is not complete, the semantic validation activity is also in progress and we here report the way we are proceeding. First, we have to collected a set of AsmetaL examples encompassing all ASM constructs, and then we have to validate their semantics by building suitable scenarios. In order to build an extensive set of scenario specifying the expected behavior, instead of writing the scenario by hand, we plan to simulate the original examples with AsmetaS (the simulator of AsmetaL specifications, see Sect. 4) itself, and parse the log files produced by AsmetaS in order to obtain valid scenario files in the Avalla syntax. Then we will run the validator

with these scenarios and the input examples. In this way we would have checked the conformance of AsmetaS with the semantics of the ASM as defined by the hooking function.

Development and integration of tools Following the LEMP process and taking advantage of the metamodelling approach, we have developed a set of tools for ASMs – the ASMETA (ASM mETAmodeling) tool-set available in [4].

As *generated* tools, the ASMETA tool-set includes (among other things) a textual notation, *AsmetaL*, to write ASM models (conforming to the AsmM), and a text-to-model compiler, *AsmetaLc*, to parse AsmetaL models and check for their consistency w.r.t. the AsmM OCL constraints. As *based* tools, we developed: a simulator, *AsmetaS* [15], to execute ASM models; a validator, *AsmetaV* [7], with its language *Avalla* to express scenarios, for scenario-based validation of ASM models; and a graphical front-end, *ASMEE* (ASM Eclipse Environment), which acts as IDE and it is an eclipse plug-in. As *integrated* tool, we have modified the *ATGT* [13] tool that is an ASM-based test case generator based upon the SPIN model checker.

5 Related work and conclusions

The process LEMP has been defined with the aim to give a complete view of all steps necessary to engineer a metamodel-based language. It tries to establish all the steps, together with their dependency relations, to define a metamodel-based language and manage the tooling activity around the language.

Language engineering processes have been the object of study in many contexts of software engineering, see for example [11, 18]. 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. As already stated, 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.

Concerning the metamodelling technique for language engineering, we can mention the official metamodels supported by the OMG for the MOF itself, the UML, the OCL, etc. Formal methods communities like the Graph Transformation community [22, 38] and the Petri Net community [33], have also started to settle their tools on general metamodels and XML-based formats. A metamodel for the ITU language SDL-2000 has been also developed [12].

Regarding the derivation of concrete grammars for metamodels, several tools exist: EMFText [21] working for Ecore metamodels, TCS [24] (Textual Concrete Syntax) for metamodels written in KM3, TEF (Textual Editing Framework) for EMF-based metamodels, etc. Viceversa, Xtext by openArchitectureWare allows to derive a language metamodel from the language concrete textual grammar. An overview of textual grammars and metamodel is given in [31].

Concerning the lack of a framework to formally define metamodel-based language semantics that LEMP tries to overcomes, some recent works have addressed the problem of providing executability into current MDE frameworks. Work described in [35] and those supported by the XMF-Mosaic/Xcore [39] environment and Kermeta [30], are some examples in this direction.

Concerning the application of ASMs for specifying the executable semantics of metamodel-based languages, we can mention the *semantic anchoring* approach described in [8, 9]. Formal models of computation expressed in AsmL (an ASM dialect) are used to give executable semantics to domain specific languages. A further result [19] shows how to use the ASMs as pseudo-code to express behavioral semantics of a domain specific language and achieve model execution. They apply metamodel-based technologies for the creation of a language description for Sudoku.

On the aspect of language semantics validation that we consider an important feature of LEMP, in [34], Maude is exploited to perform simulation, reachability and model-checking analysis of Domain Specific Visual Language semantics given in a declarative way by the graph transformation technique.

The challenges of tool integration are discussed in [37], where the authors present a software language engineering solution technique that uses Model-Driven Engineering to address tool interoperability.

On the basis of our experience in developing the AsmM/ASMETA toolset, we believe a modeling language development can gain benefits from the use of MDE automation means. Indeed, the metamodel-based approach has the advantage of being suitable to derive from the same metamodel several language artifacts and a flexible infrastructure for language processing tools development and inter-operability. As future work, we plan to extend LEMP in order to support *model evolution activities* [28] such as model refinement, model refactoring, model inconsistency management, etc. Today, only limited support is available in model-driven development tools for these activities, but a lot of research is being carried out in this particular field, especially for language engineering, to establish synergies between model-driven approaches and many other areas of software engineering including software reverse and re-engineering, generative techniques, grammarware, ontologies, aspect-oriented programming, etc.

References

1. The Eclipse Graphical Modeling framework. `http://www.eclipse.org/gmf/`.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
3. M. Aksit, E. Kindler, A. McNeile, and E. Roubtsova, editors. *Behaviour Modelling in Model Driven Architecture*, CTIT Workshop Proceedings Series WP09-04, 2009.
4. The Abstract State Machine Metamodel website. `http://asmeta.sf.net/`, 2006.
5. J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.
6. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.

7. 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.
8. K. Chen, J. Sztipanovits, and S. Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *EMSOFT*, pages 35–43, 2005.
9. K. Chen, J. Sztipanovits, and S. Neema. Compositional specification of behavioral semantics. In *DATE*, pages 906–911, 2007.
10. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`, 2008.
11. J. Favre, D. Gasevic, R. Lämmel, and A. Winter. 4th international workshop on language engineering (ATEM 2007). In *MoDELS Workshops*, volume 5002 of *LNCS*, pages 28–33. Springer, 2007.
12. 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.
13. 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.
14. 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.
15. 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.
16. A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on Software Engineering Advances*, pages 373–378, 2008.
17. A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *Journal of Automated Software Engineering*, Online First, 2009.
18. D. Gasevic, R. Lämmel, and E. V. Wyk, editors. *Software Language Engineering, First International Conference, SLE 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*. Springer, 2009.
19. T. Gjøsæter, I. F. Isfeldt, and A. Prinz. Sudoku - a language description case study. In *Proc. SLE'08*, pages 305–321, 2008.
20. D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, 2004.
21. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and renement of textual syntax for models. In *ECMDA-FA*, 2009.
22. R. Holt, A. Schürr, S. E. Sim, and A. Winter. Graph exchange language. `http://www.gupro.de/GXL/index.html`.
23. E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Journal of Software and Systems Modeling*, 2008.
24. F. Jouault, J. Bzivin, 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.
25. S. Kent. Model driven engineering. In *Integrated Formal Methods: Third International Conference (IFM)*, LNCS 2335, pages 286–298. Springer-Verlag, 2002.
26. I. Kurtev, J. Bézivin, and M. Aksit. Technical spaces: An initial appraisal. In *CoopIS, DOA'2002, Federated Conferences, Industrial track*, Irvine, 2002.

27. I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *OOPSLA Companion*, pages 602–616, 2006.
28. T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *International Workshop on Principles of Software Evolution (IWPSE'05)*, 2005.
29. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
30. P.-A. Muller, F. Fleurey, and J.-M. Jezequel. Weaving executability into object-oriented meta-languages. In *Proc. of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.
31. 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.
32. 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.
33. Petri net markup laguage (pnml). `http://www.informatik.hu-berlin.de/top/pnml`.
34. J. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In *Software Language Engineering: First International Conference, Sle 2008 Toulouse, France, September 29-30, 2008. Revised Selected Papers*, page 54. Springer, 2009.
35. M. Scheidgen and J. Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA*. Springer, 2007. LNCS.
36. D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.
37. Y. Sun, Z. Demirezen, F. Jouault, R. Tairas, and J. Gray. A model engineering approach to tool interoperability. In *SLE*, pages 178–187, 2008.
38. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In *J. Padberg (Ed.), UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS*, 2001.
39. The Xactium XMF Mosaic. `www.modelbased.net/www.xactium.com/`, 2007.