Autom Softw Eng
DOI 10.1007/s10515-009-0053-0

A semantic framework for metamodel-based languages

Angelo Gargantini · Elvinia Riccobene · Patrizia Scandurra

Received: 7 February 2008 / Accepted: 31 March 2009

Abstract In the model-based development context, metamodel-based languages are increasingly being defined and adopted either for general purposes or for specific domains of interest. However, meta-languages such as the MOF (Meta Object Facility)—combined with the OCL (Object Constraint Language) for expressing constraints—used to specify metamodels focus on structural and static semantics but have no built-in support for specifying behavioral semantics. This paper introduces a formal semantic framework for the definition of the semantics of metamodel-based languages. Using metamodelling principles, we propose several techniques, some based on the *translational approach* while others based on the *weaving approach*, all showing how the Abstract State Machine formal method can be integrated with current metamodel engineering environments to endow language metamodels with precise and executable semantics.

We exemplify the use of our semantic framework by applying the proposed techniques to the OMG metamodelling framework for the behaviour specification of the Finite State Machines provided in terms of a metamodel.

A. Gargantini · P. Scandurra (✉)
Dip. Ing. dell'Informazione e Metodi Matematici, Universitá di Bergamo, V.le Marconi, 5, 24044 Dalmine, Italy
e-mail: patrizia.scandurra@unibg.it

P. Scandurra
e-mail: scandurra@dti.unimi.it

A. Gargantini
e-mail: angelo.gargantini@unibg.it

E. Riccobene
Dip. di Tecnologie dell'Informazione, Universitá di Milano, via Bramante, 65, 26013 Crema, Italy
e-mail: elvinia.riccobene@unimi.it

Published online: 17 April 2009

Keywords Metamodelling · Model-based development · Model driven engineering · Formal methods · Abstract state machines · Language semantics · Semantic (meta-)hooking · Weaving behaviour

1 Introduction

Modelling is beginning to take a more prominent role in software development. Recent initiatives such as the MDA (Model Driven Architecture) (MDA Guide V1.0.1 2003) from the OMG (Object Management Group)—which supports various standards including the UML (Unified Modeling Language) (UML 2.1.2 2009)—, MDE (Model-driven Engineering) (Bézivin 2005), MIC (Model-integrated Computing) (Sztipanovits and Karsai 1997), Software Factories and the Microsoft DSL tools (Microsoft DSL Tools 2005), etc., promote a *model-based* approach to software development where models are first-class entities that need to be maintained, analysed, simulated and otherwise exercised, and mapped into programs and/or other models by automatic model transformations. *Modelling languages* offer designers modelling concepts and notations to capture structural and behavioural aspects of their applications. In contrast to *general-purpose* modelling languages (like the UML) that are used for a wide range of domains, some modelling languages are often tailored to a particular problem domain, and for this reason considered *domain-specific*.

Modelling languages themselves can be seen as artifacts of the model-based approach to (software) language engineering. Indeed, in a model-based language definition, the abstract syntax of a language is defined in terms of an object-oriented model, called *metamodel*, that characterizes syntax elements and their relationships, so separating the abstract syntax and semantics of the language constructs from their different concrete notations.

The definition of a language abstract syntax by a metamodel is well mastered and supported by many metamodelling environments (Eclipse/Ecore, GME/MetaGME, AMMA/KM3, XMF-Mosaic/Xcore, etc.). However, the same cannot be said for the semantics definition, which is the other important aspect in language design. Currently, metamodelling environments allow to cope with most syntactic and transformation definition issues, but they lack of any standard and rigorous support to provide the semantics of metamodels, which is usually given in natural language.

The definition of a means for specifying rigorously the semantics of metamodels is currently an open and crucial issue in the model-driven context. A presentation of existing approaches and techniques is given in Sect. 2.

What is required is a metamodelling environment sufficiently rich to express all syntactic and semantic aspects of a language. We believe this goal can be achieved by integrating metamodelling techniques with formal methods providing the requested and lacked rigour and preciseness. In general, metamodels semantics can be given with different degrees of formality by a mapping to a sufficiently well-known domain or target platform (like the JVM). However, incomplete and informal specification of a language makes precise understanding of its syntax and semantics difficult. Moreover, the lack of formally specified language semantics can cause a semantic mismatch between design models and tools supporting the analysis of models of the

language (Chen et al. 2005). These shortcomings can be avoided by the use of formal methods that can guarantee high formality and preciseness in semantics specification. A reasonable and desirable formal method to use for this scope should have the following important features: (i) it should be powerful enough to capture the principal models of computation and specification methods, (ii) it should be endowed with a metamodel-based definition conforming to the underlying metamodelling framework in order to allow automatic model transformation by model-based techniques, (iii) it should be executable in order to validate metamodels semantics, (iv) it should be able to work at different levels of abstraction, and (v) it should provide a refinement mechanism so that one can focus on a few concepts at a time and (possibly) deliver the language semantics specification at different refinement phases.

Currently, we have been studying the feasibility and the advantages of integrating metamodelling techniques and formal methods in the context of the ASM (Abstract State Machine) formalism (Börger and Stärk 2003) which, according to the requirements above and as better described in Sect. 5, seems to be a good candidate. We started by defining a metamodel for ASMs (Gargantini et al. 2006; AsmM 2006), and we have developed the ASMETA framework (AsmM 2006; Gargantini et al. 2007a) as an instantiation of the OMG metamodelling framework for ASM related concepts, to create and handle ASM models exploiting the advantages offered by the metamodelling approach and its facilities (derivatives, libraries, APIs, etc.) for building and integrating ASM tools.

In this paper, we address the issue of defining a formal semantic framework to express the semantics (possibly executable) of metamodel-based languages. We propose several techniques, some based on the *translational approach* while others based on the *weaving approach*. All these techniques show how the ASM formal method can be integrated, and in same cases *promoted* as a meta-language, with current metamodel engineering environments to endow languages with precise and executable semantics. These techniques imply a different level of automation, user freedom, possible reuse, and user effort in defining semantics, but they all share a common unifying formal framework.

As exemplification of the use of our semantic framework, we apply the proposed techniques to the OMG metamodelling framework for the behaviour specification of the Finite State Machines (FSMs) provided in terms of a metamodel. The choice of this toy example is intentional and due to the fact that we prefer the reader to concentrate on understanding our semantic techniques rather than a complex application case study. Note that the two main semantic techniques (meta-hooking and weaving) have been applied to define a precise and executable semantics of the SystemC UML profile (Gargantini et al. 2008; Scandurra 2005) —as part of the definition of a model-based SoC (System-on-Chip) design flow for embedded systems.

The remainder of the paper is organized as follows. Section 2 provides a description of related work along the lines of our motivations. Some background on the formal definition of a modelling language and the problem of defining a language formal semantics is given in Sect. 3. Section 4 describes the OMG metamodelling framework and the FSM metamodel that we used throughout the paper as case study. Section 5 provides basic concepts concerning ASMs. Section 6 presents the proposed ASM-based semantic framework to metamodel-based language definition, while the

subsequent Sects. 7, 8, 9, and 10 provide a detailed description of the techniques supported by the framework. Section 11 provides a comparison of the proposed techniques in relation to the level of the metamodelling stack they are applied to and shows how other related techniques existing in literature can be captured by those proposed in our framework. Finally, Sect. 12 concludes the paper.

2 Related work and motivations

The formal specification of the semantics of a modelling language is a key current issue for model-based engineering.

Initially, languages such as OCL (OCL 2006) (and its various extensions) allowed to specify structural semantics (*static semantics*) through invariants defined over the abstract syntax of the language. They are used to check whether a model conforms to the modelling language (i.e. it is well-formed) or not. The OCL can also be used to specify behavioural semantics through the definition of pre- and post-conditions on operations; being side-effect free, the OCL does not allow the change of a model state, but it allows describing it.

Some recent works have addressed the problem of providing executability into current metamodelling frameworks like Eclipse/Ecore (EMF 2008), GME/MetaGME (GME 2006), AMMA/KM3 (AMMA 2005), XMF-Mosaic/Xcore (XMF Mosaic 2007), etc., and thereby provided techniques for semantics specification natively with metamodels. Different solutions have been proposed that may be classified into tree main categories: (i) *translational semantics*, (ii) *weaving behaviour*, and (iii) *semantic domain modelling*.

The first approach (*translational semantics*) consists in defining a mapping (enacted by the use of model transformation engines) from the abstract syntax of the underlying language to the abstract syntax of another language which is supposed to be formally defined (i.e., for which a mapping to a semantic domain is defined). This translational semantics has been used, for example, in Chen et al. (2005, 2007) where a *semantic anchoring* to well-established formal models of computation (such as finite state machines, data flow, and discrete event systems) built upon AsmL (ASML 2001) is proposed, by using the transformation language GME/GReAT (Graph Rewriting And Transformation language) (Balasubramanian et al. 2006). The solution they propose to the semantic anchoring offers up predefined and well-defined sets of *semantic units* for future (conventional) anchoring efforts. However, we see two main disadvantages in this approach: first, it requires well understood and safe behavioural language units and it is not clear how to specify the language semantics from scratch when these language units do not yet exist; second, in *heterogeneous systems*, specifying the language semantics as composition of some selected primary semantic units for basic behavioural categories (Chen et al. 2007) is not always possible, since there may exist complex behaviours which are not easily reducible to a combination of existing ones.

Still concerning the translational category, two other experiments have to be mentioned: the semantics of the AMMA/ATL transformation language (Di Ruscio et al. 2006b) and SPL, a DSL for telephony services, (Di Ruscio et al. 2006a) have been specified in XASM (Anlauff 2000), an open source ASM dialect. A direct mapping

from the AMMA meta-language KM3 to an XASM metamodel is used to represent metamodels in terms of ASM universes and functions, and this ASM model is taken as basis for the semantics specification of the ATL metamodelled language. However, this mapping is neither formally defined nor the ATL transformation code which implements it have been made available in the ATL transformations Zoo or as ATL use case (Jouault et al. 2006); only the Atlantic XASM Zoo (XASM Zoo 2006), a mirror of the Atlantic Zoo metamodels expressed in XASM (as a collection of universes and functions), has been made available.

In Combemale et al. (2007), an attempt is given towards a generic framework to specify and effectively check temporal properties over arbitrary models in SIMPLEPDL, a process modelling language. A way is proposed to use a temporal extension of OCL, TOCL, to express properties, and a model transformation to Petri Nets and LTL formulae for both the process model and its associated temporal properties are specified.

The Moses project (Esser and Janneck 2001) provides a framework for supporting various visual formalisms (time Petri nets, process networks, statecharts, etc.) for modelling heterogeneous systems, and a simulation platform (made of built-in or external simulators) to execute such a mixture of models, possibly in a distributed fashion on several processing units. The Moses framework adopts a Graph Type Definition Language (GTDL) to define syntactical aspects of visual notations (vertex/edge types, their attribute structure, syntactical predicates, etc.), while the semantics is defined by pluggable user-defined interpreters/compilers. ASMs are also adopted as abstract specification language for rapid prototyping of interpreters/compilers. The goal is ambitious, but unfortunately the project seems out of date.

The second approach consists in (*weaving behaviour*) into metamodels, i.e. specifying an executable semantics directly on the abstract syntax of the language by *promoting* meta-languages for the semantics specification. Meta-programming languages like Kermeta (Muller et al. 2005), xOCL (eXecutable OCL) of the XMF-Mosaic metamodelling framework (XMF Mosaic 2007), or also the approach in Scheidgen and Fischer (2007), belong to this category. Inspired from the UML action semantics (AS 2001; UML 2.1.2 2009; fUML 2008), they all use a minimal set of executable primitives (create/delete object, slot update, conditional operators, loops, local variables declarations, call expressions, etc.) to define the behaviour of metamodels by attaching behaviour to classes operations.[1] Such action languages can be imperative or object-oriented. Although, they aim to be pragmatic, extensible and modifiable, some of them suffer from the same shortcomings of traditional UML-based action languages, i.e. they are a simplified version of real programming languages, and therefore a description written in one of such action languages has the same complexity of one (a program) written in a conventional programming language. Action languages that fall in this category, such as the xOCL language in the XMF toolkit (XMF Mosaic 2007) and the OMG QVT standard (QVT 2008), may be efficiently employed (like ordinary programming languages) in model repositories

[1]The MOF, for example, allows the definition (inside classes of a metamodel) of the name and the type signature of operations, but it does not allow the specification of the body counterpart which has to be described in text using an external action language.

and meta-programming environments for model management purposes to implement and execute queries, views, transformations, etc., on and between models, rather than used to specify the execution semantics of formalisms represented by metamodels. The adoption of more abstract action languages is desirable to reduce model complexity. Moreover, not all action semantics proposals are powerful enough to specify the model of computation (MoC) underlying the language being modelled and to provide such a specification with a clear formal semantics. As shown in Sect. 10 when we illustrate the weaving technique of our ASM-based framework, the ASMs formalism itself can be also intended as an action language but with a concise and powerful set of action schemes provided by different ASM rule constructors.

The third approach (*semantic domain modelling*) suggests defining a metamodel even for the "semantic domain"—i.e. to express also concepts of the run-time execution environment—, and then using well-formedness OCL rules to map elements of the language metamodel into elements of the semantic domain metamodel. It is used, for example, by the OMG task forces for the *CMOF Abstract Semantics*—see Chap. 15 in (MOF 2006)—and for the OCL (OCL 2006). A similar technique, also used for the OCL semantics, involves set theory to formulate the semantic domain in terms of an object model (a formalization of UML class diagrams, not of the OCL metamodel) and system states for the evaluation of OCL expressions. This last was originally proposed in Richters (2001), and then used also in Flake and Müller (2004) where new components to the object model and system states have been introduced and the ASM formalism was used to formalize the evaluation of OCL constraints.

We essentially investigate the two major approaches, *translational semantics* and *weaving behaviour*, and for both categories, we propose some techniques showing how the ASM formalism can be exploited as specification language to endow metamodel-based languages with a rigorous and executable description of their semantics.

Concerning the UML language, probably the most well-known example of metamodel-based modelling language, many papers on the semantics of UML exist in literature. However, specifying a formal semantics of UML is still an open problem as demonstrated by the existence of the international UML 2 Semantics Project (Broy et al. 2007), that has, among its several major objectives, that of specifying a definitive and complete formal semantics foundation for the UML 2 standard. In fUML (2008), an attempt can be found to define an executable subset of standard UML (the Foundational UML Subset) that can be used to define the semantics of modelling languages such as the standard UML or its subsets and extensions.

Through several case studies, ASMs have shown to be a formal method suitable for system modelling and for describing the semantics of modelling/programming languages. Among successful applications of the ASMs in the field of language semantics, we can cite UML and SDL-2000, programming languages such as Java, C/C++, and hardware description languages (HDLs) such as SystemC, SpecC, and VHDL—complete references can be found in Börger and Stärk (2003). Concerning the ASM application to UML to provide precise and rigorous semantics of metamodel-based languages, we can mention, to name a few, the work in Ober (2000), Börger et al. (2000), Cavarra et al. (2004), Jürjens (2002), Compton et al. (2000). More or less, all these approaches define an ASM model able to capture the semantics of a particular

kind of UML graphical sub-language (statecharts, activity diagrams, etc.). It would be desiderable to generalize the approach to any metamodel-based language by providing a general framework where ASMs can be completely integrated into a metamodelling environment to allow building well-formed and well-understood metamodels in a uniform and systematic way. The only attempts in this direction are the work in Chen et al. (2005, 2007), Di Ruscio et al. (2006a, 2006b) already discussed above.

The choice of ASM as formal support for metamodel-based language semantics specification is intentional and due to the fact that this formalism owns all the characteristics of preciseness, abstraction, refinement, executability, metamodel-based definition, that we identified in Sect. 1 as the desirable properties for this goal. Of course, the proposed goal is achievable by any other formal language sharing the same characteristics.

3 Model-based language specification

Regardless of their general or domain specific nature, modelling languages share a common structure: they usually have a concrete syntax (textual, graphical, or mixed); they have an abstract syntax; they have a semantics which can be implicitly or explicitly defined, and may be executable. Formally, a language L is defined (Chen et al. 2005; Clark et al. 2001) as a five-tuple $L = \langle A, C, S, M_C, M_S \rangle$, consisting of abstract syntax A, concrete syntax C, semantic domain S, syntactic mapping M_C and semantic mapping M_S.

The abstract syntax A defines the language concepts, their relationships, and well-formedness rules available in the language. The concrete syntax C defines a specific notation used to express models, which may be graphical, textual, or mixed. The syntactic mapping, M_C: $C \rightarrow A$, assigns syntactic constructs to elements in the abstract syntax.

The language semantics is defined (Harel and Rumpe 2004) by choosing a semantic domain S and defining a semantic mapping M_S: $A \rightarrow S$ which relates syntactic concepts to those of the semantic domain. The semantic domain S and the mapping M_S can be described in varying degrees of formality, from natural language to rigorous mathematics. It is very important that both S and M_S are defined in a precise, clear, and readable way. The semantic domain S is usually defined in some formal, mathematical framework (transition systems, pomsets, traces, the set of natural numbers with its underlying properties, are examples of semantic domains). The semantic mapping M_S is not so often given in a formal and precise way, possibly leaving some doubts about the semantics of L. Section 6 presents our approach to the definition of S and M_S.

In a model-based approach, the abstract syntax A of L is described by a *meta-language ML*, which itself has an abstract syntax A_{ML}. A is called the *metamodel* of L, while A_{ML} is the *meta-metamodel*. In this paper, we focus on languages whose abstract syntax is defined by in terms of a metamodel. The metamodel of a language describes the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models. 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 C_i and their syntactic mappings M_{C_i}, 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 M_S. Therefore, a metamodel could be intended as a standard representation of the language notation.

According to the definition in Kurtev et al. (2006), a terminal model m written in a language L has the language metamodel A as its *reference model*. The reference model of a metamodel A is the meta-metamodel A_{ML}. The relation between a model and its reference model is called *conformance*, denoted in the sequel by ω.

4 Running case study

In this section, a running example is introduced. It is extensively used throughout the next sections to illustrate the proposed semantics specification techniques. The presentation of the case study includes an overview of the chosen metamodelling framework, which in our case is the OMG/MOF (MOF 2006) framework, and a description of the metamodel of the language or formalism being specified, which in our case is the Finite State Machines. Clearly, we are concerned here with issues related to metamodels conforming to MOF, but the approach can be applied to any other metamodelling framework.

4.1 The OMG metamodelling framework

In the OMG metamodelling stack—the OMG calls M0 the real world, M1 its models, M2 the metamodels and M3 the self-defined meta-metamodel—, the MOF (Meta Object Facility) is the meta-language to define metamodels, namely the *ML*, and it is self-descriptive, i.e. the MOF metamodel is defined using concepts defined in MOF (meta-circularity).

OMG offers several transformation languages, which take as input a model conforming to a given metamodel and produce as output another model conforming to a given metamodel. Transformation processes can be completely automatized by means of *transformation engines* for model-to-model transformation languages such as those provided by the Eclipse Modeling Project (M2M project 2007), like the ATL engine (Jouault et al. 2006), as implementation of the OMG Queries/Views/Transformations (QVT 2008) standard.

4.2 FSM case study

To show the application of our different techniques to provide semantics to a metamodel-based language, we choose the Finite State Machines (FSMs) as simple case study.

Figure 1 shows the metamodel of an FSM. The metamodel describes the modelling elements for specifying a model of behaviour composed of a finite number of states, transitions between those states, and events. The model of the state machine described here is a *Mealy machine* which generates an output event based on its current state and input. One of the states is chosen as initial state. Moreover, the metamodel allows

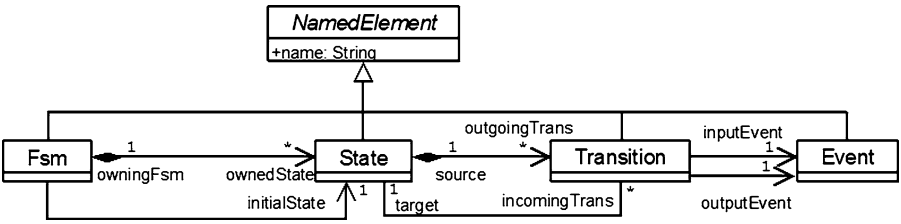
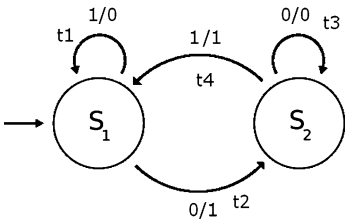


Fig. 1 A class diagram for the FSM metamodel

Fig. 2 A Finite State Machine



the description of both deterministic and non-deterministic (for each pair of state and input event there may be several possible next states) FSMs.

Figure 2 shows an FSM accepting binary strings. State S_1 is designated as initial state. The machine emits the output event 1 when it changes state, while it emits the output event 0 when it remains in the current state as effect of firing a self-transition.

5 The Abstract State Machines and the ASMETA toolset

Abstract State Machines can be seen an extension of FSMs (Börger 2005), where unstructured control states are replaced by states comprising arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them, while the *transition relation* is specified by "rules" describing the modification of the functions from one state to the next.

The notion of ASMs formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*. Furthermore, it supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synchronous/Asynchronous Multi-agent ASMs*. Appropriate rule constructors also allow non-determinism (**choose** or existential quantification) and unrestricted synchronous parallelism (universal quantification **forall**).

A complete mathematical definition of the ASM method can be found in Börger and Stärk (2003), together with a presentation of the great variety of its successful application in different fields such as: definition of industrial standards for programming and modelling languages, design and re-engineering of industrial control systems, modelling e-commerce and web services, design and analysis of protocols,

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

A number of ASM tools have been developed for model simulation, model-based testing, verification of model properties by proof techniques or model checkers—see GASM (2008) for a list.

Although the ASM method comes with a rigorous scientific foundation, the practitioner needs no special training to use it since it can be understood correctly as pseudo-code or Virtual Machines working over abstract data structures. In Sect. 5.1 we quote this *working* definition of the ASMs, which is useful for our purposes.

In addition to its mathematical-based foundation, a metamodel-based definition for ASMs is also available. An ASM metamodel called *AsmM* (*Abstract State Machines Metamodel*) (Riccobene and Scandurra 2004; Gargantini et al. 2006, 2007b; AsmM 2006) provides an abstract syntax for an ASM language in terms of MOF concepts. On the base of the AsmM and exploiting the advantages of the metamodelling techniques, a general framework, called ASMETA *tool set* (AsmM 2006), for a wide inter-operability and integration of new and existing tools around ASMs (ASM model editors, ASM model repositories, ASM model validators, ASM model verifiers, ASM simulators, ASM-to-Any code generators, etc.) has been developed. The AsmM and the ASMETA tool-set are presented in Sect. 5.2.

5.1 Abstract State Machines

Based on Sect. 2.2.4 of Börger and Stärk (2003), an ASM, viewed as pseudo-code over abstract data structures, can be defined as the tuple (*header*, *body*, *main rule*, *initialization*).

The *header* contains the *name* of the ASM and its *signature*,[2] namely all domain, function and predicate declarations (domain and function classification is described in Sect. 5.2.1).

The *body* of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations (definitions) of transition rules. Basically, a transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* are a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed[3] when *Condition* is true. An ASM M is therefore a finite set of rules for such guarded multiple function updates. State transitions of M may be influenced in two ways: *internally*, through the transition rules, or *externally* through the modifications of the environment. A *computation* of M is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of M, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the transition rules which are enabled in S_n. The

[2] For multi-agent ASM, the header contains also the machine *import* and *export clauses*, namely all names for functions and rules which are, respectively, imported from another ASMs, and exported from the current one. We assume that there are no name clashes in these signatures.

[3] 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 only in the new interpretation of the function f.

body of ASM may also contains definitions of *axioms* for invariants one wants to assume for domains and functions of the ASM.

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

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

The notion of module is also supported. An ASM *module* is an ASM without a main rule and a characterization of the set of initial states.

5.2 The AsmM metamodel and the ASMETA toolset

The AsmM metamodel (Riccobene and Scandurra 2004; Gargantini et al. 2006, 2007b; AsmM 2006) is a complete meta-level representation of ASMs concepts based on the OMG's MOF 1.4 (MOF 2002). Metamodelling representation results in class diagrams. Each class is also equipped with a set of relevant *constraints*, OCL invariants written to fix how to meaningfully connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams. AsmM is also publicly available—see (AsmM 2006)—as expressed in the meta-languages AMMA/KM3 (Jouault and Bézivin 2006) and in EMF/Ecore (EMF 2008) thanks to the ATL-KM3 plugin (Jouault et al. 2006) which allows model transformations both in the EMF and MOF modelling spaces.

The complete AsmM metamodel is organized in one package called ASMETA containing 115 classes, 114 associations, and 150 OCL class invariants, approximatively. The ASMETA package is further divided into four packages as shown in Fig. 3.

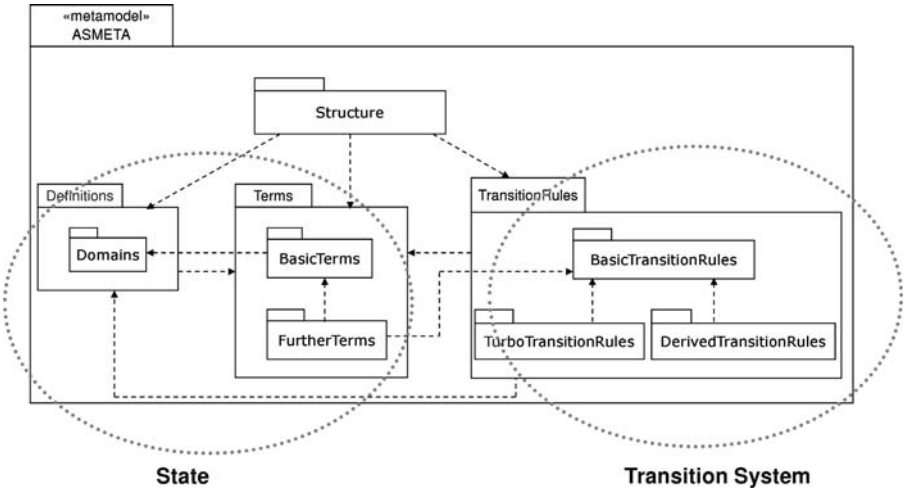


Fig. 3 Package structure of the AsmM metamodel

Each package covers different aspects of the ASMs. The dashed gray ovals in Fig. 3 denote packages representing the notions of *State* and *Transition System*, respectively. The `Structure` package defines architectural constructs (modules and machines) required to specify the backbone of an ASM model. The `Definitions` package contains all basic constructs (functions, domains, constraints, rule declarations, etc.) which characterize algebraic specifications. The `Terms` package provides all kinds of syntactic expressions which can be evaluated in a state of an ASM. The `TransitionRules` package contains all possible transition rules schemes of Basic and Turbo ASMs. All *derived* transition rules[4] are contained in the `DerivedTransitionRules` package. All relations between packages are of type *uses*.

Figure 4 shows the backbone of an ASM according to the definition given in Sect. 5.1. Further technical details on the AsmM can be found in the Sect. 5.2.1 below. We present here only a very small fragment of the AsmM whose complete description can be found in Gargantini et al. (2006), AsmM (2006).

A general framework, called *ASMETA tool set* (Gargantini et al. 2007a; AsmM 2006) has been developed based on the AsmM and exploiting the advantages of the metamodelling techniques. The ASMETA toolset essentially includes: a textual notation, *AsmetaL*, to write ASM models (conforming to the AsmM) in a textual and human-comprehensible form; a text-to-model compiler, *AsmetaLc*, to parse ASM models written in AsmetaL and check for their consistency with respect to the OCL constraints of the metamodel; a simulator, *AsmetaS*, to execute ASM models (stored in a model repository as instances of AsmM); the Avalla language for scenario-based validation of ASM models, with its supporting tool, the AsmetaV validator; the ATGT tool that is an ASM-based test case generator based upon the SPIN model checker; a graphical front-end, called ASMEE (ASM Eclipse Environment), which acts as IDE and it is an Eclipse plug-in.

5.2.1 Further AsmM concepts

This subsection presents further simplified fragments of the AsmM subset which will be used in Sect. 9.1.1 to define a mapping from MOF to AsmM. These fragments are reported in Fig. 5, 6, 7, 8 by the usual UML class diagram notation. They are related to ASM domains and functions for representing the data structures derived from the abstract syntax (metamodel) of a language expressed in MOF.

Axioms, domains, functions, and rule declarations are all represented by subclasses of the *abstract* class *`NamedElement`* (see Fig. 5). The expression of an `Axiom` is given by a term, specified by the association end `body`, which yields a boolean value when evaluated in a state of the ASM. An axiom must refer to at least one function or one domain.

In order to represent domains and distinguish between user-defined sets and domains with predefined types, in the AsmM we introduce the abstract class *`TypeDomain`* (see Fig. 6) for *type-domains* and the class `ConcreteDomain` for subdomains of type-domains, which are provided by the user. The association end

[4]The AsmM metamodel in Fig. 3 includes other ASM transition rule schemes derived from the basic and the turbo ones, respectively. Although they could be easily expressed at model level in terms of other existing rule schemes, they are considered "syntactic sugar" and therefore they have been included in the metamodel. Example of such rules are the case-rule and the (turbo) iterative/recursive while-rule.

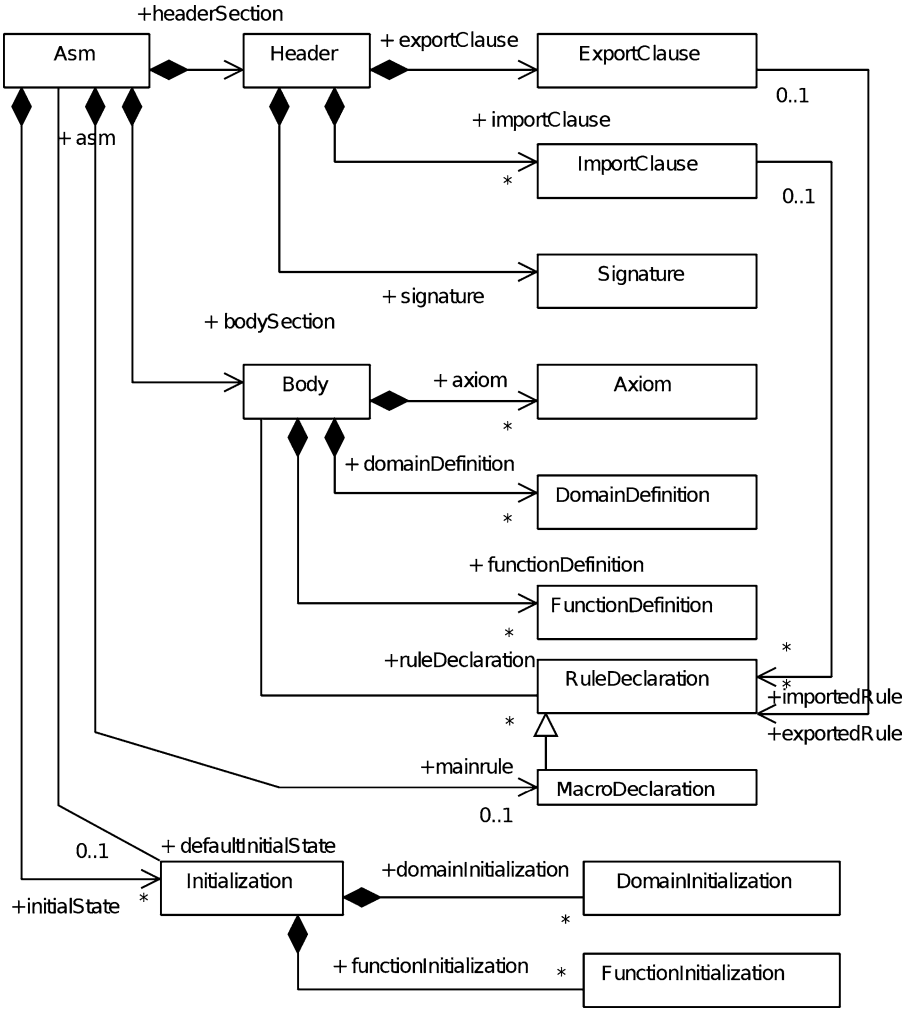


Fig. 4 Backbone

with role name `subsetof` between the classes *`Domain`* and `ConcreteDomain` binds a concrete domain to its corresponding parent-domain. The class *`Type-Domain`* is further classified in: *`BasicTypeDomain`*, for primitive data values; *`StructuredTypeDomain`* (see Fig. 7), representing type-domain constructors for structured data (like finite sets and tuples); `AbstractTypeDomain`, modelling user named domains whose elements have no precise structure. For both abstract and concrete domains, the boolean attribute `isDynamic` specifies if the domain is *static* (never changes) or *dynamic* (its content can change during the computation by effect of transition rules, typically by an *extend rule* which allows creating new objects dynamically).

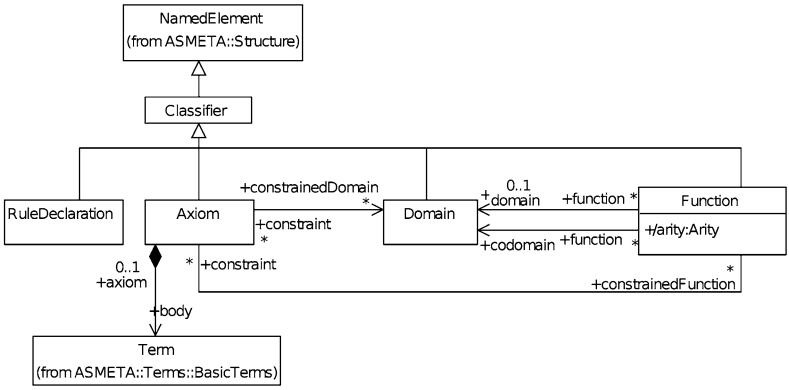


Fig. 5 AsmM named elements: axioms, rule declarations, domains, functions

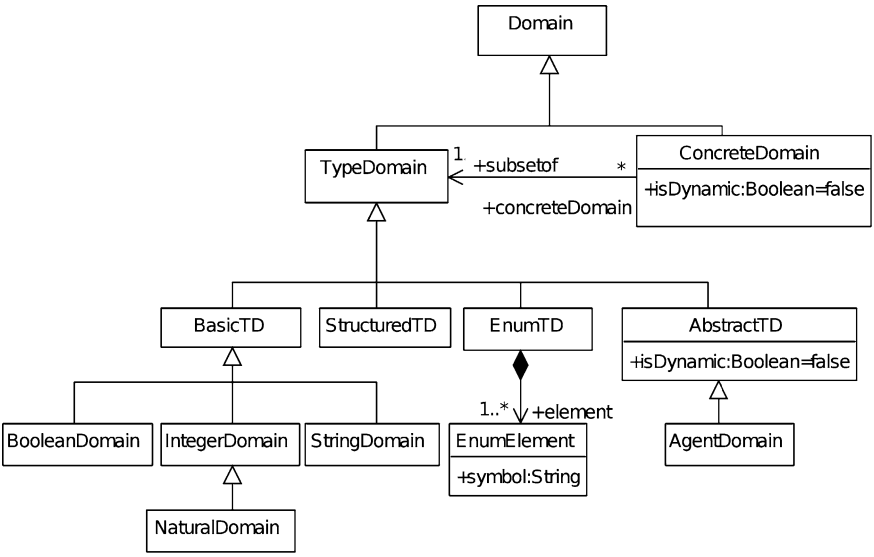


Fig. 6 AsmM domains

The abstract class `Function` and its hierarchy is detailed in Fig. 8. It models the notion of function and function classification in ASMs. *Derived* functions are functions coming with a specification or computation mechanism given in terms of other functions. *Basic* functions are classified into *static* (never change during any run of the machine) and *dynamic* ones (may change as a consequence of agent actions (or *updates*). Dynamic functions are further classified into: `monitored` (only read), *controlled* (read and write), *shared* and *output* (only write) functions.

ASM rule constructors are represented by subclasses of the class `Rule`. Figure 9 shows a subset of basic forms of a transition rule under the class hierarchy rooted by the class `BasicRule`: update rule, conditional rule, skip, do in parallel (block rule), extend, etc.

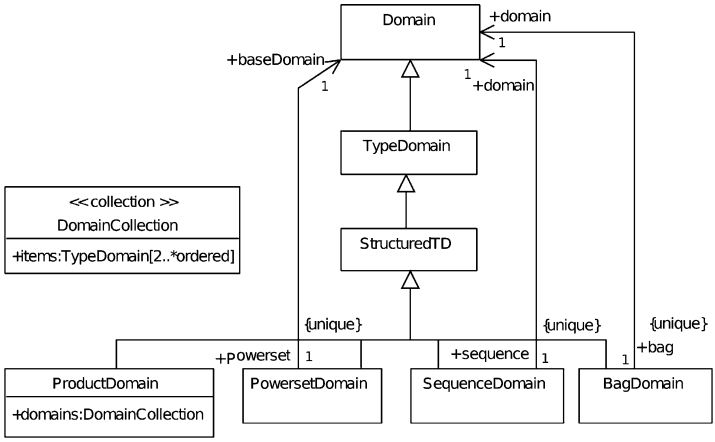


Fig. 7 AsmM structured domains

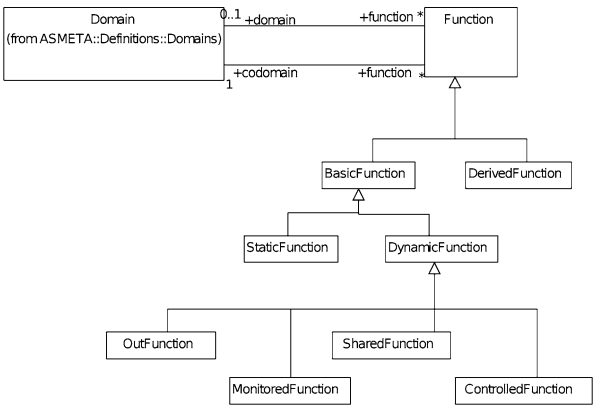


Fig. 8 AsmM functions classification

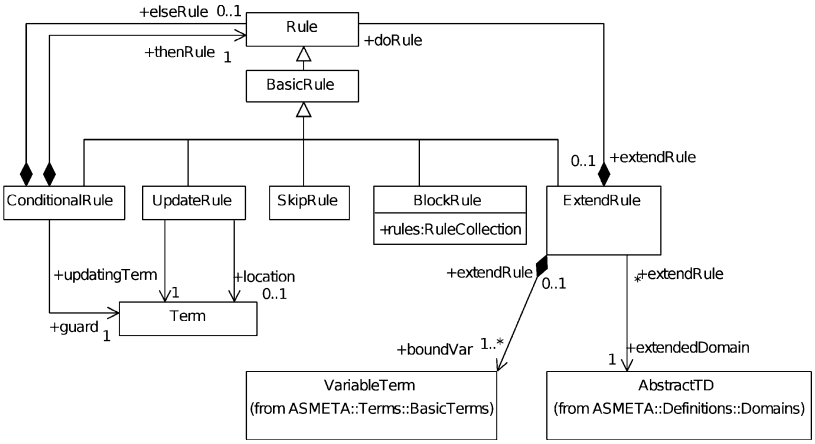


Fig. 9 Basic rule forms (a subset)

6 A semantic framework for metamodel-based languages

Recall from Sect. 3 that a language L is defined as $\langle A, C, S, M_C, M_S \rangle$, where A is the abstract syntax, S is the semantic domain, and M_S the semantic mapping. In this section we assume that A is defined by a metamodel and we introduce a way to precisely define S and M_S.

A language can be considered well-defined if its semantic mapping M_S leads from a clear and expressive syntax A to a well-defined and well-understood semantic domain S (Harel and Rumpe 2004). No other constraints over S and M_S are given, as long as they are clearly defined. The notion of semantic mapping is widely adopted in the field of programming language semantics: "*The general idea of semantics through translation is an important and pervasive one in the study of programming languages*" (Gunter 1992). The way in which this mapping and the semantic domain S are defined, differentiates the various kinds of semantics: operational, denotational, axiomatic, and so on. The mapping M_S must be intended with a broad meaning: in operational semantics, for example, it is an abstract interpretation of the constructs of L.

Sometimes, in order to give the semantics of a language L, another helper language L', whose semantics is clearly defined and well established, is introduced. The helper language is defined itself as $L' = \langle A', C', S', M'_C, M'_S \rangle$ and we assume this time that M'_S and S' are already well-defined.

In this case, L' can be exploited to define the semantics of L by (1) taking S' as semantic domain for L too, i.e. $S' = S$, (2) by introducing a *building function* $M : A \to A'$ which associates an element of A' to every construct of A, and (3) by defining the semantic mapping $M_S : A \to S$ which defines the semantics of A as follows:

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

In this way, the semantics of L is given by translating terminal models m of L to models of L'. Note that the function M can be applied to terminal models conforming to A in order to obtain models conforming to A', as shown in Fig. 10. In this way, the semantic mapping $M_S : A \to S$ associates a well-formed terminal model m conforming to A with its semantic model $M_S(m)$, by first translating m to m' conforming to A' by means of the M function, and then applying the mapping M'_S which is already well-defined.

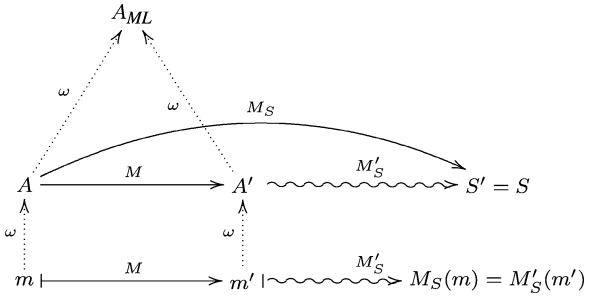


Fig. 10 The building function M

The M function *hooks* the semantics of L to the S' semantic domain of the language L' and, therefore, the problem of giving a semantic mapping of L is reduced to define the building function M.

The definition of M is accomplished by exploiting different techniques, some based on the *translational approach* while another based on the *weaving approach*.

In the translational approach, the building function M is defined as

$$M : A \rightarrow A'$$

and transforms an input model m conforming to A into a model conforming to A'. There exist three different ways of defining M, depending on the abstraction level of the metamodelling stack—see the MDE model organization in Kurtev et al. (2006)—we are working at. We classify them as: *semantic mapping*, at model level; *semantic hooking*, at metamodel level; *semantic meta-hooking*, at meta-metamodel level. Going up through the metamodelling levels, the three techniques allow increasing automation in defining the model transformation, and essentially differ in increasing reuse and decreasing dependency of the final ASM with respect to the terminal model which all techniques are, in the end, applied to.

In the weaving approach (Muller et al. 2005), L' is *promoted*[5] as meta-language and integrated with the metamodel engineering environment to weave behavioral semantics directly into metamodels. To this purpose, the current meta-metamodel A_{ML} and the A' are weaved together, by establishing precise *join points* to combine the two metamodels into a new meta-metamodel A_{ML^+} which adds to A_{ML} the capability of specifying behaviour by L'. Then, the metamodel A of L is weaved with the intended behavioural semantics, by exploiting the join points established between A_{ML} and A'. Let A^+ be the new metamodel. The building function M is defined as

$$M : A^+ \rightarrow A'$$

which associates a model conforming to A' to a terminal model conforming to A^+.[6]

6.1 ASMs for the semantic framework

We investigate in this paper the use of the ASM language as helper language L' and we present how to define the building function M accordingly. The ASM language is defined, in the ASMETA framework, by the tuple

$$\langle AsmM,\ C_{AsmM},\ S_{AsmM},\ M_{S_{AsmM}},\ M_{C_{AsmM}} \rangle \tag{1}$$

where $AsmM$ is the ASM metamodel presented in Sect. 5.2, C_{AsmM} is intended as AsmetaL (or also any other concrete notation associated to AsmM), S_{AsmM} is the first-order logic extended with a logic for function updates and for transition rule constructors formally defined in Börger and Stärk (2003), $M_{C_{AsmM}}$ is the syntax mapping

[5] According to Thirioux et al. (2007), the promotion operation maps a model into a reference model, i.e. a meta-model in turn is used as meta-metamodel to build metamodels conforming to it.

[6] Note that if a terminal model m conforms to a metamodel A, then m also conforms to the weaved metamodel A^+.

given by the AsmetaLc compiler, $M_{S_{AsmM}}$ is the semantic mapping which associates a theory conforming to the S_{AsmM} logic to a model conforming to *AsmM*.

By instantiating A' with *AsmM*, the translational techniques presented in the previous section consist in defining a building function $M : A \rightarrow AsmM$ and they are described in more details in the following Sects. 7, 8, 9. The weaving technique consists in (1) weaving the current meta-metamodel A_{ML} and the *AsmM* together into a new meta-metamodel A_{ML^+}, (2) modifying the metamodel A to obtain the metamodel A^+ and (3) defining the building function $M : A^+ \rightarrow AsmM$. This technique is presented in Sect. 10.

A concrete example is provided applying each technique to the metamodelling framework OMG/MOF and to the FSM modelling language. The results of this activity are FSM semantic models which can be made available in a model repository either in textual form using AsmetaL or also in abstract form as instance model of the AsmM metamodel.

7 Semantic mapping

By semantic mapping, a unique building function $M: A \rightarrow AsmM$ is manually defined at will by the language designer by using a model transformation language. M maps concepts of the A space into *AsmM*concepts.

A transformation language must be provided with a suitable tool framework which allows to automatically apply M to any model m conforming to A (see Fig. 11). The aim of this transformation operation is to create an ASM (i.e. a model conforming to the AsmM metamodel) on the base of the elements of the source model m and their tying elements in the source metamodel A. Elements of m are mapped to abstract data structures (signature, domain/function definitions, axioms), transition rules, and initial state of the resulting machine. Given two different models m_1 and m_2 both conforming to A, the resulting ASMs substantially differ in structure, rules and initial states, and the common parts are limited. We say the resulting ASM strongly resembles the terminal model m.

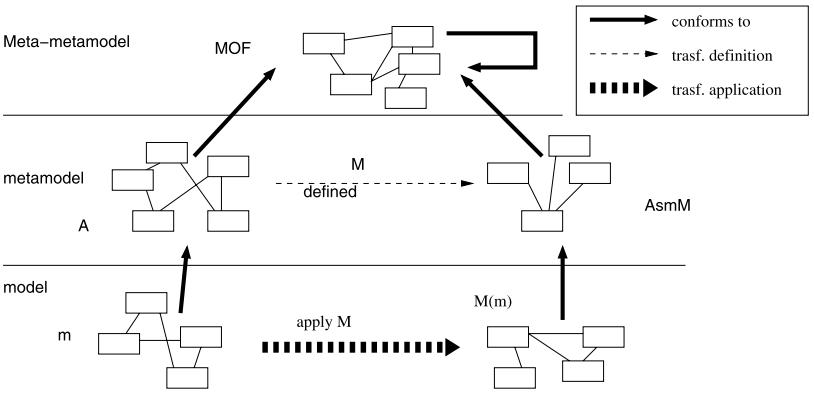


Fig. 11 Semantic mapping

Listing 1 Mapping FSM transitions into ASM conditional rules

```
lazy rule mapTrans{
from tr: FSM!Transition
to out: AsmM!RuleDeclaration
          (name <− 'r_'+tr.source.name+"_"+tr.inputEvent.name,
           ruleBody <− Sequence {createCondRule(tr)}) }
lazy rule createCondRule{
from tr: FSM!Transition
to out: AsmM!ConditionalRule(
           guard <− createGuard(tr)
           thenRule <− createAction(tr))
}
lazy rule createGuard {
// create the guard: currentState==tr.source.name and input==tr.inputEvent.name
...
}
lazy rule createAction {
// two update rules: currentState:= tr.target.name; output:= tr.outputEvent.name
...
}
```

No guidance on how to map static or behavioural parts of A into concepts of AsmM is given. However, some patterns can be defined: one is to map behavioural concepts of A into ASM rules. It is applied in the example below.

7.1 Semantic mapping for FSM

An example of semantic mapping M for the basic deterministic FSMs follows. M maps each state and each event of a terminal FSM model into a String constant. A controlled variable *currentState* of type String in the resulting ASM model takes values among these constants. A monitored variable *input* of type String is added to the ASM model to denote the current input. Each transition is mapped into a rule declaration: the rule name is formed by the name of the source state followed by the input, while the rule body is a conditional rule. This last has as guard a condition to determine if the *currentState* is the source state of the underlying transition and the input event of the transition matches the present *input*; if this guard is satisfied, appropriate update rules allows setting the *output* to the output event and the current state to the next state, accordingly.

M can be implemented as a model transformation using, for example, the ATL language. Listing 1 shows a fragment of such ATL transformation relating to the mapping of FSM transitions into ASM conditional rules. The resulting ASM for the terminal FSM model given in Fig. 2 is reported in Listing 2 using the AsmetaL notation.

For non-deterministic FSM, the semantic mapping M should be defined so to capture the non deterministic choice of all possible firing transitions. Therefore, the code reported in Listing 2 should be modified accordingly.

Listing 2 ASM for a terminal FSM model by mapping

```
asm FSM_mapping
import StandardLibrary
signature:
   controlled currentState: String
   monitored input: String
   out output: String

definitions:
   rule r_s1_1 = if currentState = "s1" and input = "1" then
                        par currentState:= "s1"
                             output:= "0"
                        endpar
                     endif
   rule r_s1_0 = if currentState = "s1" and input = "0" then
                     ...
   rule r_s2_1 = ...
   rule r_s2_0 = ...
   //run all transition rules
   main rule r_Main = par r_s1_1[] r_s1_0[] r_s2_1[] r_s2_0[] endpar
default init s0:
          function currentState = "s1"
```

8 Semantic hooking

While in the semantic mapping the emphasis is on *models*, semantic hooking focus on the language itself trying to endow L with a semantics by means of a unique ASM for any model written in L. By using this technique, designers *hook* (or *anchor*) to the language metamodel A an abstract state machine Γ_A, which is an instance of *AsmM* and contains all data structures modelling elements of A with their relationships, and all transition rules representing behavioural aspects of L. Γ_A does not contain the initialization of functions and domains, which will depend on the particular instance of A. The function which adds the initialization part is called ι (see Fig. 12). Formally, the building function M is given by

$$M(m) = \iota_A(\Gamma_A, m)$$

for all m conforming to A, where:

- Γ_A: *AsmM*, is an abstract state machine which contains only declarations of functions and domains (the signature) and the behavioural semantics of L in terms of ASM transition rules;
- ι_A: $AsmM \times A \rightarrow AsmM$, 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.

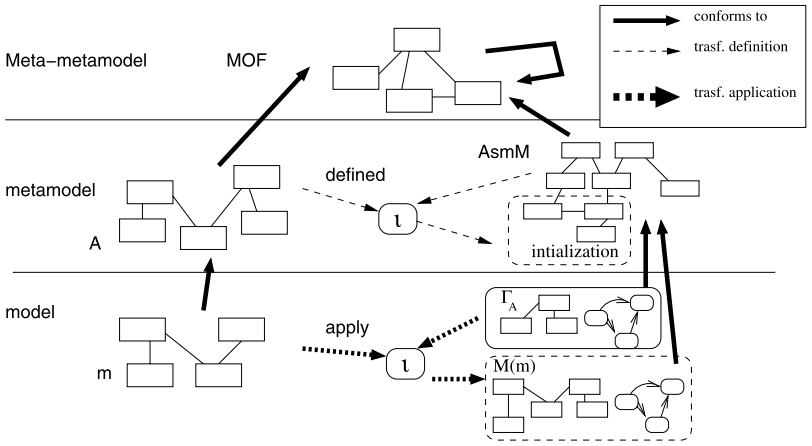


Fig. 12 Semantic hooking

8.1 Semantic hooking for FSM

The Γ_{FSM} models the semantics for the FSM language presented in Sect. 4.2. Listing 3 reports a possible Γ_{FSM} resembling that defined in Chen et al. (2005). It introduces abstract domains for the FSMs themselves, transitions, states, and events. Two controlled functions store the initial and current state of a FSM, while the output function returns the output of a FSM. The behaviour of a generic FSM is given by two rules *r_doTransition* and *r_fsmReact*. The main rule executes all machines in the *Fsm* set.

One has also to define a function ι_{FSM} which adds to Γ_{FSM} the initialization necessary to make the ASM model executable. Any model transformation tool can be used to automatize the ι_{FSM} mapping by retrieving data from m and creating the corresponding ASM initial state in the target ASM model. Listing 4 shows part of this mapping using the ATL model transformation language, where `m_FSM` is Γ_{FSM}, and `myFSM` is any terminal FSM model m. Essentially, for each class instance of the terminal model `myFSM`, a static 0-ary function is created in the signature of the ASM model `m_FSM` in order to initialize the domain corresponding to the underlying class (see the ATL rule `initializeDomains`). Moreover, class instances with their properties values and links are inspected to initialize the ASM functions declared in the ASM signature (see the ATL rule `initializeFunctions`). For example, for the automaton shown in Fig. 2, the provided ι_{FSM} mapping would automatically add to the original Γ_{FSM} the initial state partially shown in Listing 5. The initialization of the abstract domains `FSM`, `Transition`, `State` and `Event`, and of all functions defined over these domains, are added to the original Γ_{FSM}.

9 Semantic meta-hooking

By semantic hooking, a language designer hooks to each language with abstract syntax A its ASM Γ_A modelling the language semantics. The semantic meta-hooking

Listing 3 Γ_{FSM}

```
asm FSM_hooking
signature:
  abstract domain Fsm
  abstract domain State
  abstract domain Transition
  abstract domain Event

  //Functions on Fsm
  controlled initialState: Fsm −> State
  controlled currentState: Fsm −> State
  monitored input: Fsm −> Event
  out output: Fsm −> Event

  //Functions on Transition
  controlled source: Transition −> State
  controlled target: Transition −> State
  controlled inputEvent: Transition −> Event
  controlled outputEvent: Transition −> Event

definitions:
    rule r_doTransition($m in Fsm, $t in Transition) = par
                    output($m) := outputEvent($t)
                    currentState($m):= target($t)
                  endpar

    rule r_fsmReact($m in Fsm, $e in Event) =
      let ($s = currentState($m)) in
        choose $pt in Transition with source($pt)= $s and inputEvent($pt)= $e do
            r_doTransition[$m,$pt]
      endlet

  //run all Fsm machines
    main rule r_Main = forall $fsm in Fsm do r_fsmReact[$fsm,input($fsm)]
```

technique aims at automatically deriving (part of) Γ_A (and hence part of the mapping M) by exploiting concepts in the source meta-model and in the meta-metamodel.

Essentially, signature, domain and function definitions, and axioms, which do not depend on the terminal model, are induced by the source metamodel. This resulting algebra is then endowed with (operational) semantics (i.e. ASM transition rules); this requires a certain human effort in order to capture in terms of ASM transition rules the behavioural aspects of the underlying language. Finally, by navigating a specific terminal model m, the initial state (values for domains and functions of the signature) is determined. Formally, we define the building function $M : A \rightarrow AsmM$ as

$$M(m) = \iota(\omega(m))(\tau_A(\gamma(\omega(m))), m)$$

for all m conforming to A, where:

- $\gamma : A_{ML} \rightarrow AsmM$ provides signature, domain and function definitions, and axioms of the final machine $M(m)$ from the metamodel $\omega(m)$,

Listing 4 ι_{FSM} mapping for the FSM

```
module FSM2AsmM_iota;
-- Metamodels: FSM.xmi (Ecore), AsmM.xmi(MDR)
-- Models: IN1 myFSM.xmi (Ecore), IN2 m_FSM.xmi (MDR)
create OUT:AsmM from IN1:FSM, IN2:AsmM;
-- This rule returns an AsmM!Asm along with its associated AsmM!Initialization
-- from a terminal model myFSM:FSM!Fsm and the original m_FSM:AsmM!Asm
rule Fsm2Asm {
 from myFSM: FSM!Fsm, m_FSM: AsmM!Asm
 to new_header: AsmM!Header (
      importClause <- m_FSM.headerSection.importClause,
      exportClause <- m_FSM.headerSection.exportClause,
      signature <- thisModule.initializeDomains(myFSM,
                        m_FSM.headerSection.signature)),
     out: AsmM!Asm (
      name <- m_FSM.name,
      headerSection <- new_header,
      bodySection <- m_FSM.bodySection,
      mainrule <- m_FSM.mainrule,
      initialState <- thisModule.initializeFunctions(myFSM,new_header.signature))
}
-- This rule returns an AsmM!Signature from the original ASM signature m_FSM_s
-- with new function symbols for class instances of the terminal model myFSM
lazy rule initializeDomains {
     from myFSM: FSM!Fsm, m_FSM_s: AsmM!Signature
     to out: AsmM!Signature (
      function <- Sequence{
        FSM!Fsm->allInstances()->collect(e |
            thisModule.createDomainElement(e,m_FSM_s)),
        FSM!State->allInstances()->collect(e |
            thisModule.createDomainElement(e,m_FSM_s)),
        FSM!Transition->allInstances()->collect(e |
            thisModule.createDomainElement(e,m_FSM_s)),
        FSM!Event->allInstances()->collect(e |
            thisModule.createDomainElement(e,m_FSM_s))} ->flatten() )
}
-- This rule generates an AsmM!Initialization from a terminal model myFSM
-- and the new ASM signature new_m_FSM_s
lazy rule initializeFunctions {
     from myFSM: FSM!Fsm, new_m_FSM_s: AsmM!Signature
     to out: AsmM!Initialization (
        functionInitialization <- Sequence{
          thisModule.initFunctOnFsm(myFSM,new_m_FSM_s),
          thisModule.initFunctOnState(myFSM,new_m_FSM_s),
          thisModule.initFunctOnTransition(myFSM,new_m_FSM_s)}->flatten())
}
-- This rule generates a 0-ary AsmM!StaticFunction corresponding to
-- a class instance 'e' of the terminal model myFSM
lazy rule createDomainElement {
     from e: FSM!NamedElement, new_m_FSM_s : AsmM!Signature
     to out: AsmM!StaticFunction (
              name <- e.name,
              arity <- 0,
              codomain <- new_m_FSM_s.domain->select(D| D.name = e.oclType().name))
}...
```

Listing 5 ASM for a terminal FSM model by hooking

```
asm FSM_hooking
signature:
    ....
  static myFsm: Fsm
  static t1,t2,t3,t4: Transition
  static s1,s2: State
  static e1,e2: Event
    ....
default init s0:
  //Functions on Fsm
  function initialState($m in Fsm) = at({myFsm −> s1},$m)
  function currentState($m in Fsm) = at({myFsm −> s1},$m)

  //Functions on Transition
  function source($t in Transition) = at({t1−>s1,t2−>s1,t3−>s2,t4−>s2},$t)
  function target($t in Transition) = at({t1−>s1,t2−>s2,t3−>s2,t4−>s1},$t)
  function inputEvent($t in Transition) = at({t1−>e1,t2−>e0,t3−>e0,t4−>e1},$t)
  function outputEvent($t in Transition) = at({t1−>e0,t2−>e1,t3−>e0,t4−>e1},$t)
```

- $\tau_A : AsmM \rightarrow AsmM$ provides the definition of the semantics of L by adding to the ASM the transition rules capturing the behavioural aspects of L,
- $\iota : A_{ML} \rightarrow (AsmM \times A \rightarrow AsmM)$ is an HOT (High Order Transformation)[7] and establishes, for a metamodel A, the transformation $\iota(A)$ which computes the initial state of the final machine $M(m)$ by extracting initial values for data structures of the machine from the source modelling elements in m.

The final ASM $M(m)$ is built step by step by the mappings above. Note that mappings γ and ι are *universal* (independent from the language L), i.e. once defined for a meta-language ML (which fixes the metamodelling environment) they are applicable to all metamodels conforming to ML. τ_A is a *refining transformation* which can be intended as a modelling activity carried out "once for all" for the metamodel A and is reusable for all models m conforming to A. Moreover, the application of mappings γ, τ_A, and ι can be automatized by writing them in terms of a model transformation language and then executed by a model transformation engine. We present a concrete implementation of all these mappings in Sect. 9.1 for the OMG metamodelling environment.

9.1 Meta-hooking in the OMG framework

We here show how to implement the meta-hooking technique in the context of the OMG MOF.

[7] An HOT is a transformation taking as input or producing as output another transformation.

First, **step 0**, the MOF core concepts (*Essential MOF*) used to define metamodels are mapped into ASMs concepts by defining the function γ: $A_{MOF} \rightarrow AsmM$; γ may be a partial function. In practice, this is effectively done by defining a set of mapping rules between the MOF metamodel and the AsmM metamodel, as described in the Sect. 9.1.1 below, where we introduce a suitable $\gamma_{A_{MOF}}$ function which can be reused making this step optional.

In **step 1**, by applying these mapping rules $\gamma_{A_{MOF}}$ to a metamodel A conforming to MOF, we obtain a model conforming to AsmM made of the signature symbols (essentially domains and functions) representing classes and relations of the source A metamodel, domain and function definitions, and possible axioms as direct encoding of OCL constraints on A.

The next **step 2** consists into defining the refining function τ_A, so endowing the ASM obtained from the previous step 1 with the ASM transition rules capturing the behavioural semantics of the given language L. The final result of this modelling activity is the ASM machine $\tau_A(\gamma(\omega(m)))$ capturing the complete operational semantics of the language L.

Note that this step could require the refinement of the given ASM signature by adding, for example, *new* functions necessary to describe behavioural facets within the body of the transition rules. Moreover, for better readability and for facilitating the navigation through modelling elements of the source metamodel, many other *new* ASM (derived) functions may be introduced in terms of composition of ASM functions already existing in the given signature. In order to guarantee the assumption that the complete signature of the resulting ASM is generated by γ, while τ_A can define only transition rules, we need to add these new functions to the original metamodel A in terms of MOF constructs (essentially, class properties or association ends) or OCL query/operations. In this way, the ASM generated at step 1 by applying γ to any terminal model, will contain all the functions used by the transition rules introduced by τ_A at step 2.

Note also that in order to guarantee the execution of each instance of the "concrete root class",[8] here referred to as C_r, which may be created in any execution scenario of the source metamodel, the τ_A activity implies also the introduction of a main rule (the entry point for executing the target ASM model) which usually has the following form:

$$\textbf{main rule } r\_main = \textbf{forall } e \textbf{ in } C_r \textbf{ do } r\_run(e)$$

where the rule named *r_run* is supposed to be the rule to invoke on each element of C_r for "executing".

Finally, **step 3**, to make the ASM machine executable for a specific terminal model m conforming to A, an initial state of the ASM must be provided. To this aim, one should (a) define an HOT ι, (b) apply ι to $A = \omega(m)$ in order to obtain the transformation $\iota(A)$, (c) apply $\iota(A)$ to the ASM obtained at step 2 and to the terminal model m

[8] At the top of the inheritance hierarchy of a metamodel structure there is usually an abstract class called `NamedElement` which is the root of all modelling elements that have a name. We prefer the use of the term "concrete root class" throughout the paper to denote, instead, a non abstract class (possibly direct or non-direct sub-class of `NamedElement`) which may be used as starting point for exploring the whole metamodel (see e.g. the `Fsm` class in the FSM metamodel in Fig. 1).

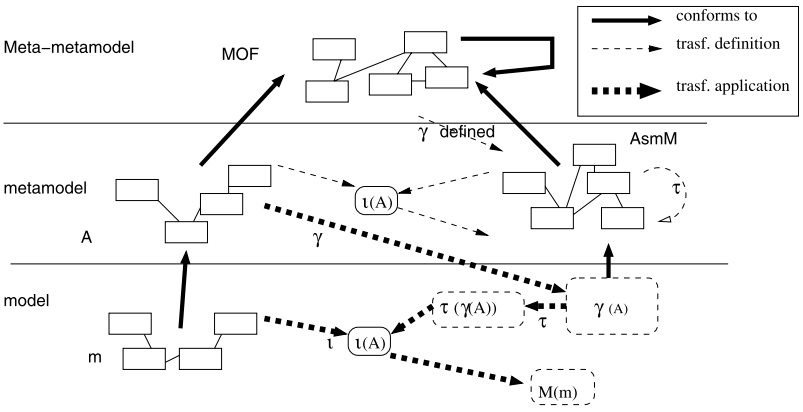


Fig. 13 Semantic meta-hooking in the OMG framework

in order to map elements of *m* into initial values for the ASM domains and functions. Figure 13 depicts the overall scenario. Step 3(a) can be done once for all and the HOT ι can be reused for any metamodel conforming to MOF. Section 9.1.2 introduces a suitable ι and Sect. 9.2 shows a concrete example of application of such mapping to the FSM language.

9.1.1 From MOF to AsmM: γ mapping rules

The γ mapping rules are in Table 1. Essentially, a **class** *C* is mapped to a dynamic abstract domain *C*. Each domain represents the set of objects that can be created by the corresponding class. If the class is *abstract*, the corresponding abstract domain is declared to be static.

Primitive types (`Integer`, `String`, `Boolean`, and `UnlimitedNatural`) and **enumerations** are mapped respectively into the AsmM basic type domains they represent.

Properties (either attributes or references) are mapped into ASM functions. The multiplicity and the adornments *ordered* and *unique* are captured by the codomain types of the corresponding functions. Moreover, although we assume that terminal models are well-formed according to the OCL constraints defined on their metamodel, additional ASM axioms may be added to ensure correct size of collections corresponding to each instance of a class. All target ASM functions are supposed to be *controlled*, unless a further adornment `isReadOnly` or `isDerived` is used for the property; in this last case the corresponding ASM function is *monitored* or *derived*, respectively. We do not take into account of other property adornments provided by MOF such as *subsetting*, *derived union*, *redefinitions*, *specialization*—see (MOF 2006) for more details—as they are not present in other meta-languages and imply complex semantic issues.

Navigable association ends are considered the same as attributes and references. Therefore, they are mapped into two dynamic controlled ASM functions or one dynamic controlled ASM function depending on whether the association is bi-

Table 1 γ mapping: from MOF to AsmM

MOF	AsmM
A non-abstract class C	A dynamic `AbstractTD` domain C
An abstract class C	A static `AbstractTD` domain C
An `Enumeration`	An `EnumTD` domain
A primitive type	A basic type domain
`Boolean`	`BooleanDomain`
`String`	`StringDomain`
`Integer`	`IntegerDomain`
`UnlimitedNatural`	`NaturalDomain`
An attribute a of a class C, type T and multiplicity 1	A function $a : C \rightarrow T$
An attribute a of a class C, type T, multiplicity > 1, and *ordered*	A function $a : C \rightarrow T^*$ where T^* is the domain of all finite sequences over T (`SequenceDomain`)
An attribute a of a class C, type T, multiplicity > 1, *unordered* and *unique*	A function $a : C \rightarrow \mathcal{P}(T)$ where $\mathcal{P}(T)$ is the mathematical powerset of T (`PowersetDomain`)
An attribute a of a class C, type T, multiplicity > 1, *unordered* and *not unique*	A function $a : C \rightarrow B(T)$ where $B(T)$ is the domain of all finite bags over T (`BagDomain`)
A reference or a navigable association end	See attribute
A *generalization* between a child class *C1* and a parent class *C2*	A `ConcreteDomain` C_1 subset of the corresponding domain C_2
OCL constraints	Axioms (optional)
OCL operations/queries	Static functions

directional navigable or not, respectively. The name of the functions must reflect the role name of the corresponding ends.

A **generalization** between a class C_1 that inherits from a class C_2 is expressed by declaring a concrete domain C_1 (`ConcreteDomain`) as subset of the corresponding domain C_2. Currently, we put the restriction on the number of types a given object type can be a subtype of; i.e. multiple inheritance is not supported.

Constraints representing further static information in the form of OCL **well formedness rules** can be encoded in ASM axioms over the ASM domains corresponding to the context classes;[9] however, if not required, structural constraints can remain expressed in OCL, as we assume that input models are well-formed. Similarly, OCL **operations/queries** can be mapped into static ASM functions, as well.

Note that, here we choose not to model the MOF notion of `Operation` on classes and of `default` value for a property. These notions are used in Sect. 10 in the weaving approach.

[9]Both OCL and ASM can be used for writing static integrity constraints, since they employ first order predicates to express logical conditions.

Table 2 $\iota(A)$ mapping: from A to AsmM

A	AsmM
An object *obj* instance of a class C	A static 0-ary function declaration $obj : C$
A property value $p = v$ (an attribute value or reference value or link end) of an object *obj* with type *Type*, and multiplicity *mult*	A pair term $obj \rightarrow v$ in a map term $obj_i \rightarrow v_i$ used to initialize the function $p : C \rightarrow Type$ where *Type* can be a simple domain T or $\mathcal{P}(T)$ or T^* or $B(T)$ depending on the multiplicity *mult*, ordering and uniqueness

9.1.2 From A to AsmM: ι mapping rules

For any language metamodel A, the ι HOT returns a transformation $\iota(A)$ with the mapping rules sketched in Table 2 to be used for any terminal model m conforming to A. Essentially, for each class instance of the terminal model, a static 0-ary function is added to the signature of the ASM model in order to initialize the domain corresponding to the underlying class. Moreover, model objects with their property values (attributes values ore reference values or links values) are inspected to initialize the ASM functions declared in the ASM signature.

A model transformation engine (such as ATL) can be used to automatize the ι HOT rules by retrieving data available at terminal model level and creating the corresponding initializing elements in the target ASM model.

9.2 Meta-hooking for FSM

According to the meta-hooking technique described before, the complete semantic specification of the FSM formalism is provided in terms of an ASM model derived in tree steps: (1) the γ mapping in Table 1 is applied to the FSM abstract syntax in Fig. 1 to express it in terms of an ASM signature; (2) the operational semantics of the FSM formalism is then defined by ASM transition rules as form of pseudo-code operating on the abstract data derived from step 1; finally, (3) the initial state of the semantic model for a terminal FSM model m given in input is provided by the mapping $\iota(FSM)$ in Table 2.

1. *ASM signature for the FSM metamodel* From the class diagram in Fig. 1, classes `Fsm`, `State`, `Transition`, and `Event` are mapped into dynamic concrete ASM domains as subsets of the abstract static domain *NamedElement*, together with the (controlled) functions derived from their properties. This is done by applying the function γ of Table 1. While γ is defined at MOF level and it does not depend on the abstract syntax A which it is applied to, the result of applying γ strongly depends on A and the obtained ASM signature contains functions induced by the specific language. Listing 6 shows the ASM signature in AsmetaL resulting by applying γ to the FSM metamodel.

2. *ASM transition system for the FSM metamodel* The second step consists into defining the semantics of the FSM by means of the function τ_A which captures the

Listing 6 Abstract syntax of the FSM in AsmetaL

```
asm FSM_meta_hooking
import StandardLibrary
signature:
    abstract domain NamedElement
    dynamic domain Fsm subsetof NamedElement
    dynamic domain State subsetof NamedElement
    dynamic domain Transition subsetof NamedElement
    dynamic domain Event subsetof NamedElement

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

    //Functions on Fsm
    controlled ownedState: Fsm -> Powerset(State)
    controlled initialState: Fsm -> State

    //Functions on State
    controlled owningFsm: State -> Fsm
    controlled incoming: State -> Powerset(Transition)
    controlled outgoing: State -> Powerset(Transition)

    //Functions on Transition
    controlled source: Transition -> State
    controlled target: Transition -> State
    controlled inputEvent: Transition -> Event
    controlled outputEvent: Transition -> Event
```

behavioural aspects in terms of ASMs transition rules. For this purpose, the ASM signature obtained in the previous step 1 is further enriched of concepts relating to the execution semantics: the current state, the input event to consume at each step, and the output event fired by the machine. These concepts are back-annotated in the original FSM metamodel by adding to the original `Fsm` class one readonly attribute *input* of type `Event`, and two attributes *currentState* of type `State` and *output* of type `Event`. The function γ must be applied again to obtain a signature which includes these new functions.

We complete the specification with the following ASM transition rules which formalize the behaviour of any FSM terminal model *m*. The rule *r_run* is responsible for the execution of transitions by invoking the *r_fire* rule. Non-deterministically, in each step this machine: reads (consumes) an event of the input stream, produces an output event, and then proceeds to the next control state. Note that both the non-deterministic version and the deterministic version of FSMs can be simulated by the same rules.

Listing 7 shows the ASM transition rules written in AsmetaL for the FSM metamodel, together with the new functions added to the signature by back-annotating them in the source metamodel and applying again the γ mapping. The main rule `r_main` guarantees the execution of each FSM instance (object of the concrete root class `Fsm`).

Listing 7 Transition rules for the FSM in AsmetaL

```
asm FSM_meta_hooking
import StandardLibrary
signature:
    ...
    //Added functions
    controlled currentState: Fsm −> State
    monitored input: Fsm −> Event
    controlled output: Fsm −> Event
    ...
definitions:
    ...
    rule r_fire ($t in Transition) =
    let ($m = owningFsm(source($t))) in
      par
        output($m) := outputEvent($t)
        currentState($m):= target($t)
      endpar
    endlet

    rule r_run ($m in Fsm)=
     choose $t in outgoing(currentState($m))
                 with inputEvent($t)=input($m)
     do r_fire[$t]

    main rule r_main = forall $m in Fsm do r_run[$m]
```

3. ***ASM initial state for a terminal FSM model*** The final step consists in endowing the ASM model shown in Listings 6 and 7 with an initial state for a specific automaton (as model instance of the FSM metamodel).

This initial state is obtained as result of the mapping $\iota(FSM)$, defined by the HOT ι (see Sect. 9.1.2), applied to a given FSM terminal model to define the initial values of domains (including also the element `myFsm` of the `Fsm` domain for the underlying FSM) and of (dynamic) controlled functions of the ASM signature. For the FSM shown in Fig. 2, the provided $\iota(FSM)$ mapping would automatically produce the initial state shown in part in Listing 8 using the AsmetaL notation.

10 Weaving behaviour

The aim of this technique is to weave the behavioural aspects of a modelling language L into its language metamodel A.

Applying this technique demands the definition of a *weaving function* specifying how the current meta-metamodel A_{ML} and the *AsmM* are weaved together into a new meta-metamodel A_{ML^+} which adds to A_{ML} the capability of specifying behaviour by ASM transition rules.

Adding behavioural constructs into a meta-language requires identifying how behaviour can be attached to the structural constructs of the meta-language, namely

Listing 8 Initial state in AsmetaL for a terminal FSM model

```
asm FSM_meta_hooking
import StandardLibrary
signature:
    ...
//Initialize domains for a specific terminal model
static myFsm:Fsm
static s1:State
static s2:State
    ...
//Initialize functions for a specific terminal model
default init s0: //ASM initial state
 function currentState($m in Fsm) = initialState($m)

 //Functions on NamedElement
 function name($e in NamedElement) = at({myFsm->"myFsm",s1->"s1",
     s2->"s2",t1->"t1",t2->"t2",t3->"t3",t4->"t4",e1->"1",e0->"0"},$e)

 //Functions on Fsm
 function ownedState($m in Fsm) = at({myFsm->{s1,s2}},$m)
 function initialState($m in Fsm) = at({myFsm -> s1},$m)

 //Functions on State
 ...
```

precise *join points*[10] between data and behaviour must be identified (Muller et al. 2005). In case A_{ML} is object-oriented, as for example MOF, it might be convenient to use transition rules within meta-classes as class operations to hold their behavioural specification. Therefore, a join point must be specified between the meta-class `Operation` of MOF and the meta-class `RuleDeclaration` of the *AsmM*. The next section exemplifies this approach.

Once a weaving function has been established between the *AsmM* and the meta-metamodel by identifying suitable join points, the metamodel A of L can be weaved with the intended behaviour of L. Let A^+ be the resulting metamodel (see Fig. 14). The building function $M : A^+ \rightarrow AsmM$ is defined as $M(m) = \iota(\omega(m))(W(\omega(m))), m)$ for all m conforming to A, where:

- $W : A_{ML^+} \rightarrow AsmM$ maps a weaved metamodel A^+ into a metamodel conforming to the *AsmM* and provides the abstract data structure (signature, domain and function definitions, axioms) and the transition system of the final machine $M(m)$,
- $\iota : A_{ML^+} \rightarrow (AsmM \times A^+ \rightarrow AsmM)$ is an HOT establishing, for a metamodel A, the transformation $\iota(A)$ which computes the initial state of the final machine $M(m)$ by extracting initial values for data structures of the machine from the source modelling elements in m.

[10] Inspired from the Aspect-oriented Programming (AOP) paradigm, join points are intended here and in Muller et al. (2005) as places of the meta-metamodel where further (executability) aspects can be injected.

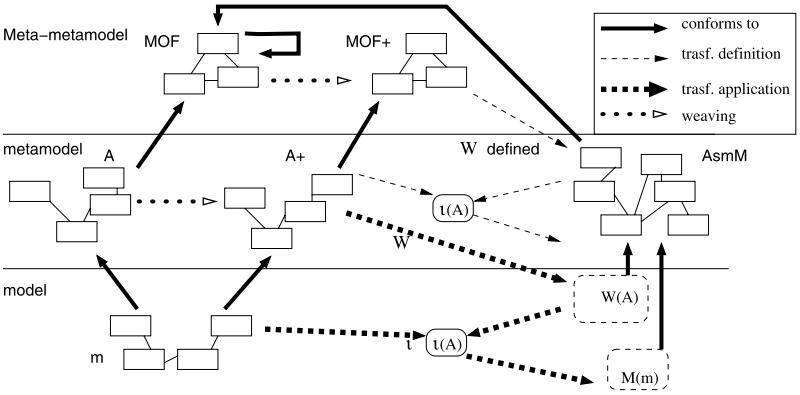


Fig. 14 Weaving in the OMG framework

The function W can be specified (once for all) as extension/redefinition of the γ mapping in Sect. 9. On those elements of the meta-language involved in the join points definition, the function W has to be defined in a way that associate them with the corresponding elements of the *AsmM* involved in the join points definition. The section below provides a concrete example of W. The HOT ι is similar to the ι operator of the meta-hooking in Sect. 9, since it provides the initialization depending on the terminal model given in input.

10.1 Weaving approach in the OMG framework

This section describes how the MOF meta-language can be enriched of behaviour specification capability in terms of ASM transition rules to define an executable meta-language, MOF$^+$. The process described here is directly applicable to any object-oriented meta-language, like the OMG MOFs (MOF 1.4, CMOF and EMOF), or Eclipse/ECore, or AMMA/KM3, etc.

We choose as *join point* the MOF class `Operation` to attach behaviour to operations of classes of a metamodel. Figure 15 shows how simply the composition may be carried out. The MOF `Operation` class resembles the *AsmM* `RuleDeclaration` class. The name `Operation` has been kept instead of `RuleDeclaration` to ensure MOF conformance, while the name `Parameter` changes in `VariableTerm`. Finally, the new property `isMain` has been added in order to designate, when set to true, a *closed* (i.e. without formal parameters) operation as (unique) main rule for model execution. Moreover, we assume that an operation cannot raise exceptions (i.e. the set of types provided by the association end `raisedException` is empty), since ASM rules do not raise exceptions.

The function W is defined as extension of the γ mapping reported in Table 1. Table 3 provides semantics to the `Operation` element of the weaved meta-language *MOF*$^+$, by associating it to the corresponding `RuleDeclaration` element of the *AsmM* involved in the join point definition. We assume, similarly to the use of `this` in the Java programming language, that in the definition of the rule body of an operation *op*, a special variable named $*this* is used to refer to the object that contains the

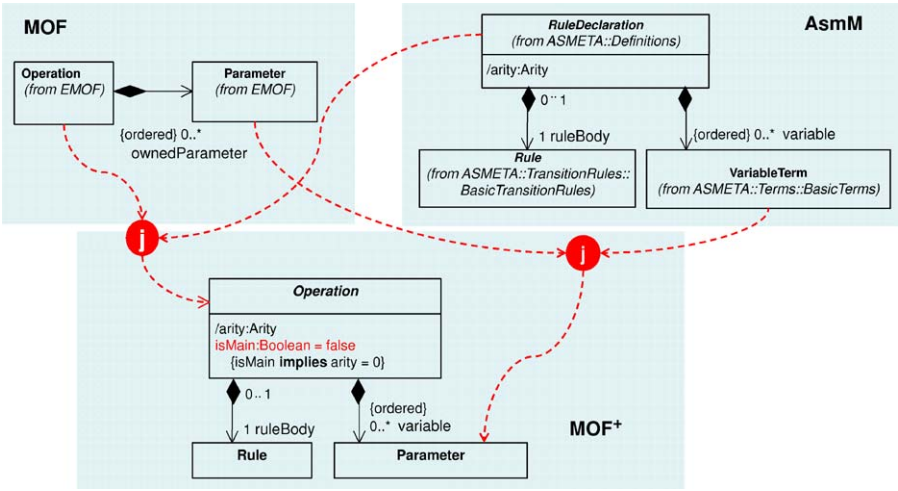


Fig. 15 Using operation bodies as join points between data and behaviour

Table 3 *W* mapping for `Operation`: from MOF^+ to *AsmM*

MOF^+	AsmM
An operation *op* of a class *C*, rule body *R*, arity *n*, and owned parameters $x_i : D_i$	A rule declaration $op(\$this \textbf{ in } C, x_1 \textbf{ in } D_1, \ldots, x_n \textbf{ in } D_n) = R$ of rule body *R*, arity $n+1$, and formal parameters $\$this \textbf{ in } C$ and $x_i \textbf{ in } D_i$
A closed operation *op* of a class *C*, rule body *R*, and with *isMain* set to true	The (unique) main rule declaration of form **main rule** $op =$ **for all** $\$this \textbf{ in } C$ **do** *R*

operation. The *W* function shown in Table 3 automatically adds the variable $\$this$ as formal parameter of the corresponding rule declaration.[11]

Since, the idea here is to extend the MOF to allow the definition of ASM transition rules working as "pseudo-code over meta-classes", a further join point is necessary in order to adorn class's properties (either attributes or references or association member ends of the metamodel) to reflect the ASM function classification (see Fig. 8 in Sect. 5.2.1). Figure 16 shows how this may be carried out. The MOF `Property` class resembles the *AsmM* `Function` class. Box MOF^+ presents the result of the composition process. The MOF class `Property` has been merged with the class `Function`. A further adornment `kind:PropertyKind` have been added to capture the complete ASM function classification. `PropertyKind` is an enumeration of the following literal values: static, monitored, controlled, out, and shared. Two OCL constraints have been also added stating, respectively, that a *read-only* (attribute `isReadOnly` is set to true) property can be of kind static or monitored, and that if

[11] Since operations are intended as ASM transition rules, within the body of an operation *op*, if *f* is a property (an attribute or a reference or an association end) in the same object, then $f(\$this)$ must be used as a full name for that property. If *anotherOp* is another operation in the same object, then $anotherOp(\$this, \ldots)$ must be used to call that operation.

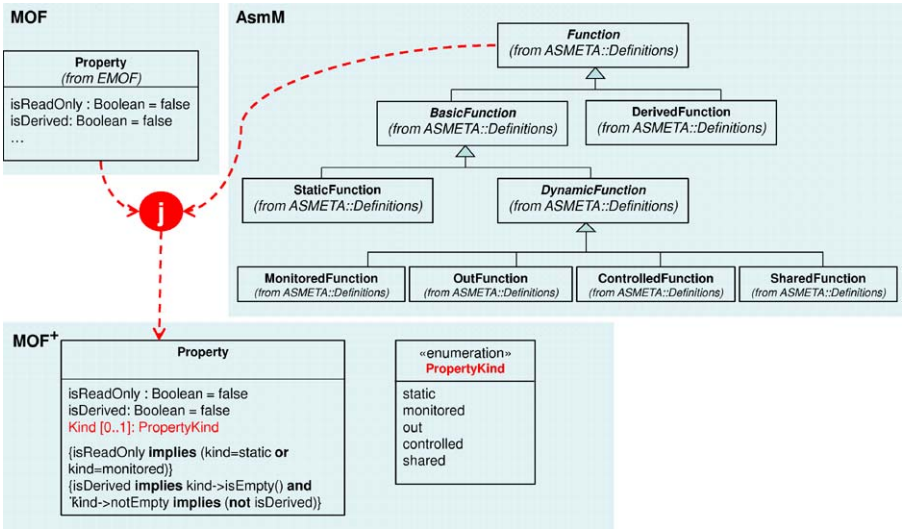


Fig. 16 Using properties as join point for ASM adornments

Table 4 W mapping for `Property`: from MOF^+ to *AsmM*

MOF^+	AsmM
A property p of a class C, kind k, type *Type*, and multiplicity *mult*	A function $p : C \rightarrow Type$ of kind k and with *Type* a simple domain T or $\mathcal{P}(T)$ or T^* or $B(T)$ depending on the multiplicity *mult*, ordering and uniqueness

a property is *derived* (attribute `isDerived` is set to true) then the attribute kind is empty.

Moreover, in order to merge the two statically-typed systems of the MOF and the *AsmM*, a MOF `Type` (a `Class` or a `DataType`) is merged with an ASM `Domain` (see Figs. 6 and 7 in Sect. 5.2.1). Hence, as explained in Table 1 for the γ mapping, the domain of the ASM function denoting a property has to be intended as the ASM domain (usually an `AbstractTD` type-domain) induced from the exposing class of the property, and the codomain as induced from the property's type.

Table 4 shows the W function as redefinition of the γ mapping reported in Table 1 to provide semantics to the `Property` element (an attribute or a reference or an association end) of the MOF^+ meta-language by associating it to the corresponding `Function` element of the *AsmM* involved in the join point definition.

10.2 Weaving for FSM

As an example, the ASM rules *r_run* (responsible for the FSM execution) and *r_fire* (responsible for the firing of a transition) shown in Listing 7 can be weaved in the FSM metamodel by attaching them as operations to the classes `Fsm` and `Transition`, respectively, as shown in Fig. 17. Moreover, the current state of the machine

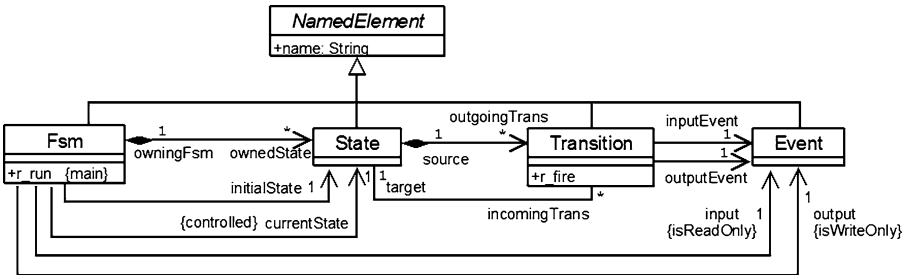


Fig. 17 FSM weaved metamodel

Table 5 Techniques comparison

	User effort	Technique provides
M	Define a map	Transformation language and engine
H	Define Γ_A and ι	Language to write Γ_A, transformation language and engine for ι
MH	Define τ_A	γ to obtain the initial ASM, further enriched by ι; transformation language and engine to apply transformations and HOT
WG	Weave A to obtain A^+ as instance of A_{ML^+}	Weaving operators with defined semantics to write metamodels conforming to A_{ML^+}; W to obtain the initial ASM, further enriched by ι; transformation language and engine to apply transformations and HOT

and the input/output events to consume/produce at each step can be provided in the metamodel as associations with appropriate ASM adornments. The notation `{controlled}` denotes that the property kind is set to *controlled*.

The application of the mappings W and ι to the finite state machine of Fig. 2 results into an AsmetaL specification which is not shown here but it is similar to that reported in Listings 6, 7 and 8 with all suitable adornments for functions.

11 A comparison among techniques

In this section, we compare our semantic techniques in relation to the level of automation each technique offers, to the degree of reuse and user effort required, and to the dependency of the final ASM capturing a terminal model semantics with respect to the terminal model itself. The availability of a common unifying framework makes this comparison easier to perform and comprehend, and it helps a language designer in choosing the best technique that fits his/her needs. Furthermore, all techniques are compared with respect to the classical topology of programming language semantics.

A brief comparison of all presented techniques in terms of user effort and what each technique provides to the user is reported in Table 5.

In the semantic mapping (M), the designer can map concepts of A to concepts of *AsmM* with great freedom. He/she must master the transformation language to obtain

an efficient (and sometimes complex) transformation and a well designed final ASM. The resulting ASMs of two different terminal models may differ on structures, rules, and initial states. Little guidance is given by the technique and the reuse is quite limited. A framework supporting this technique provides only a language to write the mapping and the engine running such transformation scripts.

The semantic hooking (*H*) tries to identify the common concepts of all models conforming to A and builds an ASM (Γ_A) which models the structural aspects of the language. The semantics of the particular model m is added by ι which takes the concrete specification as input data. The final ASM depends on the particular terminal model only in its initialization parts (initial elements in the domains, initial values for functions). This allows the language designer to reason about the language properties regardless the terminal model by using the ASM validation/verification techniques and tools.

With the meta-hooking (*MH*), most of the job is done automatically, having the designer only to define the τ_A function. The function γ, defined at meta-metamodel level, induces the right abstract data structure (signature, domain and function declarations, and axioms) for the meta-model A. In this case, the freedom of the designer is very limited as is the effort. When the user defines the transformation τ_A, he/she must take into account the resulting abstract data structure from γ and define τ_A accordingly.

In the weaving approach (*WG*), the user can work directly on the metamodel to add the rules (as operations) and functions necessary to model the semantics of A. However, when the designer adds properties and operations, he/she must be aware of the definitions of W, since operations must use functions and symbols later introduced by applying W (e.g. the symbol *$this*). The derivation of the final ASM for a terminal model conforming to A^+ is performed in a completely automatic way and possibly supported in a transparent way by an execution engine for the executable meta-language ML^+. The application of the mappings W and ι in the weaving approach is similar to that of the mappings γ and ι in the meta-hooking technique. The weaving technique has the great advantage of capturing both structural and behavioral aspects of a language at metamodel level.

Note that intermediate approaches are possible with respect to parts of the language. For example, the ι function of the hooking approach may introduce a new rule depending on the terminal model m when m contains a particular construct of L: in this case the approach would be a mix between mapping and hooking. The τ_A function may be defined to add functions to the signature obtained by $\gamma(A)$ (to avoid the back annotation of the metamodel A), making this approach a mix between hooking and meta-hooking.

Our techniques can also be compared with respect to the classical topology of the semantics for programming languages (Gunter 1992). The semantic mapping is similar to a *denotational* technique since it translates the original model into another language, in our case ASMs. The translation works as a compiler from L to *AsmM*. The semantic hooking and semantic meta-hooking are similar to *operational* techniques, since the model is here taken as data and interpreted by the machine (interpreter) defined for the language. The interpreter for the language is given by the Γ_A for the hooking and the τ_A for the meta-hooking. Both the techniques are still *translational*

since they translate the original model to a different representation in the target language of the ASM. This translation can be defined on the base of the meta-model or of the meta-metamodel making the difference between hooking and meta hooking. Finally, semantic weaving is also an operational technique, but here the model is used "as is", i.e. no translation to the target domain is needed, since when it is written it already contains the behavioral specification.

11.1 Classification of other approaches

The semantic anchoring approach proposed in Chen et al. (2005, 2007) clearly falls into the hooking category of our translational techniques. Our techniques can be intended as a means to specify the semantics from scratch for providing well understood and safe behavioural language units when they do not yet exist, while the semantic anchoring approach supposes to have in input a family of predefined semantic units.

The approach adopted in Di Ruscio et al. (2006a, 2006b), in the context of the AMMA framework, can be intended as a partial exemplification of our meta-hooking technique. Indeed, similar to our γ function, a direct mapping from the meta-language KM3 to an XASM metamodel is used to represent metamodels in terms of ASM universes and functions, and the semantics of the ATL transformation between the two metamodels is given in terms of ASM rules written in XASM. However, the approach is little formal and it is incomplete. Only the problem of mapping between metamodels to translate syntactic aspects of the terminal model is tackled, but this mapping is neither formally defined nor the ATL transformation code which implements it have been made available. Furthermore, a complete semantic definition of a language requires to reason about how to provide transition rules to capture the operational semantics and how to initialize the resulting model to make it executable, problems which are solved by our functions τ_A and ι, but are not taken in consideration in Di Ruscio et al. (2006a, 2006b).

The approach in Muller et al. (2005) for defining the executable Kermeta metalanguage, is similar to our weaving approach. They also choose the class `Operation` as join point to attach behaviour (specified with an action language) to operations of classes of a metamodel. In our case, as action language we adopt the ASMs and the ASM rule constructors, and therefore we compose the *AsmM* with the MOF with all the advantages deriving from the use of a formal method able to capture different various MOCs.

These observations provide evidence that the semantic framework proposed here is general enough to capture other existing approaches to define the semantics of metamodel-based languages which use as helper language L' the Abstract State Machines or any other formalism that can be intended as pseudo code over abstract data structures. Therefore, existing semantic techniques can be unified by our formal framework in a unique environment, allowing to switch from one technique to another with a possible reuse of model fragments (typically those involving transition rules and describing behavior) and model transformations.

12 Conclusions

In a language engineering process, which mainly uses metamodelling capabilities to implement families of languages either in specific application domains or in general purpose domains, the semantics specification is a necessary step for making languages semantically precise and of practical use in tool chains. Executability is an important property of the (specification) language used as meta-language for defining the semantics of these languages. Indeed, executable specifications describe how a system modelled by the executable specification language behaves without defining the implementation of the system behaviour.[12] Having formal operational specifications at a suitable level of abstraction would be useful for *semantics prototyping*: one can examine the semantics of a particular behavioural feature of the modelled language in an operational manner and with unnecessary details omitted.

In this paper, we explained how the Abstract State Machines (ASMs) (Börger and Stärk 2003) can be exploited as an executable specification language to endow metamodel-based languages with a rigourous and executable description of their semantics. We have shown how the formal framework of ASMs can be smoothly integrated with current metamodelling environments for language design by investigating two approaches: *semantic mapping*, *semantic hooking*, and *meta-hooking* techniques for the *translational* approach, and the *weaving behaviour* approach.

As future step, we plan to explore new techniques and evaluate the effectiveness of their joint-use with the ASM formal method for the semantics specification of metamodel-based languages. In particular, following the semantic domain modeling approach sketched in Sect. 2 which implies also the modelling, at metamodel level, of concepts of the *semantic domain*, we intend to extend the *AsmM* metamodel in order to express the underlying ASM semantic domain, namely concepts like terms values, locations, etc. of the run-time environment. This would lead to the specification of the *AsmM* semantics in terms of itself.

References

AMMA: The AMMA platform. http://www.sciences.univ-nantes.fr/lina/atl/ (2005)

Anlauff, M.: XASM—an extensible, component-based ASM language. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) Abstract State Machines, Theory and Applications, Proceedings of International Workshop, ASM 2000, Monte Verità, Switzerland, March 19–24, 2000. LNCS, vol. 1912, pp. 69–90. Springer, Berlin (2000)

AS: OMG. The Action Semantics Consortium for the UML. ad/2001-03-01. http://www.omg.org/ (2001)

ASML: The ASML language website. research.microsoft.com/foundations/AsmL/ (2001)

AsmM: The Abstract State Machine Metamodel website. http://asmeta.sf.net/ (2006)

Balasubramanian, D., Narayanan, A., vanBuskirk, C., Karsai, G.: The Graph Rewriting and Transformation Language: GReAT. In: International Workshop on Graph Based Tools (GraBaTs) (2006)

Bézivin, J.: On the unification power of models. Softw. Syst. Model. (SoSym) **4**(2), 171–188 (2005)

Börger, E.: The ASM method for system design and analysis. A tutorial introduction. In: Gramlich, B. (ed.) Frontiers of Combining Systems, Proceedings of the 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19–21, 2005. LNCS, vol. 3717, pp. 264–283. Springer, Berlin (2005)

[12] Clearly, executable specifications still represent abstract behaviour and not the actual implementation.

Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Berlin (2003)

Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) Abstract State Machines, Theory and Applications, Proceedings of the International Workshop, ASM 2000, Monte Verità, Switzerland, March 19–24, 2000. LNCS, vol. 1912, pp. 223–241. Springer, Berlin (2000)

Broy, M., Crane, M., Dingel, J., Hartman, A., Rumpe, B., Selic, B.: 2nd UML 2 semantics symposium: formal semantics for UML. In: Proc. of MoDELS 2006. LNCS, vol. 4364, pp. 318–323. Springer, Berlin (2007)

Cavarra, A., Riccobene, E., Scandurra, P.: Mapping UML into abstract state machines: a framework to simulate UML models. Studia Inform. Universalis **3**(3), 367–398 (2004)

Chen, K., Sztipanovits, J., Neema, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: Wolf, W. (ed.) EMSOFT 2005, September 18–22, 2005, Jersey City, NJ, USA, Proceedings of the 5th ACM International Conference on Embedded Software, pp. 35–43. ACM, New York (2005)

Chen, K., Sztipanovits, J., Neema, S.: Compositional specification of behavioral semantics. In: Lauwereins, R., Madsen, J. (eds.) Design, Automation and Test in Europe Conference and Exposition (DATE 2007) Proceedings, April 16–20, 2007, Nice, France, pp. 906–911. ACM, New York (2007)

Clark, T., Evans, A., Kent, S., Sammut, P.: The MMF approach to engineering object-oriented design languages. In: Workshop on Language Descriptions, Tools and Applications (2001)

Combemale, B., Crégut, P.G.X., Thirioux, X.: Towards a formal verification of process models's properties—SimplePDL and TOCL case study. In: 9th International Conference on Enterprise Information Systems (ICEIS) (2007)

Compton, K., Huggins, J., Shen, W.: A semantic model for the state machine in the Unified Modeling Language. In: Proc. of Dynamic Behavior in UML Models: Semantic Questions, UML 2000 (2000)

Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: A practical experiment to give dynamic semantics to a DSL for telephony services development. Tech. Rep. 06.03, LINA (2006a)

Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Tech. Rep. 06.02, LINA (2006b)

EMF: Eclipse modeling framework. http://www.eclipse.org/emf/ (2008)

Esser, R., Janneck, J.W.: Moses—a tool suite for visual modeling of discrete-event systems. In: Proc. IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 272–279 (2001)

Flake, S., Müller, W.: An ASM definition of the dynamic OCL 2.0 semantics. In: Proc. of UML'04, pp. 226–240. Springer, Berlin (2004)

fUML: OMG. Semantics of a foundational subset for executable UML models, version 1.0-Beta 1, ptc/2008-11-03 (2008)

Gargantini, A., Riccobene, E., Scandurra, P.: Metamodelling a formal method: applying mde to abstract state machines. Tech. Rep. 97, DTI Dept., University of Milan (2006)

Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based simulator for ASMs. In: Prinz, A. (ed.) 14th International ASM Workshop Proceedings (2007a)

Gargantini, A., Riccobene, E., Scandurra, P.: Ten reasons to metamodel ASMs. In: Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis. LNCS Festschrift. Springer, Berlin (2007b)

Gargantini, A., Riccobene, E., Scandurra, P.: (2008) A precise and executable semantics of the SystemC UML profile by the meta-hooking approach. Technical Report 110, DTI Dept., University of Milan

GASM: ASMs web site. http://www.eecs.umich.edu/gasm/ (2008)

GME: The Generic Modeling Environment (GME). http://www.isis.vanderbilt.edu/Projects/gme (2006)

Gunter, C.A.: Semantics of Programming Languages: Structures and Techniques. MIT Press, Cambridge (1992)

Harel, D., Rumpe, B.: Meaningful modeling: What's the semantics of "semantics"? IEEE Comput. **37**(10), 64–72 (2004)

Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy (2006)

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on object-oriented programming systems, languages, and applications, pp. 719–720. ACM, New York (2006)

Jürjens, J.: A UML statecharts semantics with message-passing. In: Proc. of the 2002 ACM Symposium on Applied Computing, pp. 1009–1013. ACM, New York (2002)

Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 602–616. ACM, New York (2006)

M2M project: Eclipse modeling project, model to model transformation (M2M) sub-project. http://www.eclipse.org/m2m/ (2007)

MDA Guide V1.0.1: OMG. The Model Driven Architecture (MDA guide v1.0.1). http://www.omg.org/mda/ (2003)

Microsoft DSL Tools: Microsoft DSL tools. http://msdn.microsoft.com/vstudio/DSLTools/ (2005)

MOF: OMG. The Meta Object Facility (MOF) v1.4, formal/2002-04-03 (2002)

MOF: OMG. Meta Object Facility (MOF), Core Specification v2.0, formal/2006-01-01 (2006)

Muller, P.A., Fleurey, F., Jezequel, J.M.: Weaving executability into object-oriented meta-languages. In: Proc. of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (2005)

Ober, I.: More meaningful UML Models. In: TOOLS—37 Pacific 2000, IEEE (2000)

OCL: OMG. Object Constraint Language (OCL), v2.0 formal/2006-05-01 (2006)

QVT: OMG, MOF Query/Views/Transformations, formal/08-04-03. http:www.omg.org (2008)

Riccobene, E., Scandurra, P.: Towards an interchange language for ASMs. In: Zimmermann, W., Thalheim, B. (eds.) Abstract State Machines. Advances in Theory and Practice. LNCS, vol. 3052, pp. 111–126. Springer, Berlin (2004)

Richters, M.: A precise approach to validating UML models and OCL constraints. PhD thesis, Universität Bremen, Germany (2001)

Scandurra, P.: Model-driven language definition: metamodelling methodologies and applications. PhD thesis, University of Catania, Italy (2005)

Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Model Driven Architecture—Foundations and Applications. Proccedings of the Third European Conference, ECMDA-FA 2007, Haifa, Israel, June 11–15, 2007. LNCS, vol. 4530, pp. 157–171. Springer, Berlin (2007)

Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Comput. **30**(4), 110–111 (1997). 10.1109/2.585163

Thirioux, X., Combemale, B., Crégut, X., Garoche, P.L.: A framework to formalise the MDE foundations. In: International Workshop on Towers of Models (TOWERS 2007), Zurich, Switzerland (2007)

UML 2.1.2: OMG. The Unified Modeling Language (UML), v2.2. http://www.uml.org (2009)

XASM Zoo: The Atlantic XASM Zoo. http://www.eclipse.org/gmt/am3/zoos/atlanticXASMZoo/ (2006)

XMF Mosaic: The Xactium XMF Mosaic. www.modelbased.net/www.xactium.com/ (2007)