

A Formal Foundation Supporting MDD --- ZOOM Approach

Hongming Liu, Lizhang Qin, Chris Jones, Xiaoping Jia
DePaul University
School of Computer Science
Chicago, IL
{jordan, lqin, cjones, xjia}@cs.depaul.edu

Abstract

Model-Driven Development (MDD) represents a positive step toward a general model-based approach to software engineering. The Object Management Group (OMG) offers a conceptual framework, called MDA that defines a set of standards in support of MDD. However, MDA lacks a formal foundation. We present a formal foundation supporting MDD based on ZOOM. We examine the benefits that such a formal foundation can provide. We look at the core components of ZOOM and how they differ from comparable approaches by including formal elements relating not just to application functionality, but also to application architecture and user interface requirements as well.

1. Introduction

Model-driven development (MDD) moves the development focus from third generation programming language code to models, specifically models expressed in the Unified Modeling Language (UML) and its profiles [1,2]. Instead of developers implementing a system using a programming language, MDD lets them describe the desired functionality using a set of models. MDD enables reuse at the business domain level, increases quality as models are successively improved, reduces costs by supporting automated software development processes, and increases software longevity. Because of the potential to change the way we develop applications, many researchers have been working on methodologies to support MDD [3,4].

In this paper, we propose a formal foundation based on ZOOM (Z-based Object-Oriented Modeling notation) model to support MDD. Our foundation is based on Z[5,6], a well-known formal notation. We begin by introducing some background on MDD, UML and meta-modeling. We explore the limitations of existing models and meta-models and introduce our proposed formal foundation for meta-modeling. We then explore the details of formal modeling using ZOOM as well as its supporting tools. Finally we compare our approach with other related work.

2. Background of MDD

Using models to design complex systems is common in software development. Model-driven development is the notion that we can construct a model that can then be transformed into a real system [5]. MDD aims to shift the focus of software development from coding to modeling. Instead of requiring developers to implement a system using a programming language, MDD lets them develop models of the desired system that can then be transformed into that system. Under MDD, the models tend to be reflective of the business domain rather than the technical domain.

To be complete, a model must capture all of the essential information about the subject. In the case of software, the models must capture information from three dimensions:

- The functional requirements
- The architectural requirements
- The target environment requirements

Functional requirements are the characteristics that most modeling techniques capture and are the primary concern of most developers and business modelers during systems design. While these requirements are critical for any model, they do not ensure a model's completeness.

Architectural requirements, sometimes called extra-functional requirements, or XFRs¹, are of equal importance and can be difficult to capture effectively. The XFRs stipulate the minimum service levels that an application must provide while exercising its functional requirements. Architectural requirements constrain the overall application solution space; we can ignore all solutions that don't conform to the constraints described by the XFRs.

The third dimension is concerned with the target runtime environment and its requirements. Such requirements force us to consider that any platform-independent model must ultimately be built to run on some physical platform. It does us no good to model an application that requires a large amount of memory when the application will be executed on a device such as a PDA.

Each dimension must be adequately addressed before the application can be successfully realized from a model. Failing to completely specify any one of these dimensions could result in an application that:

- Doesn't meet the user's needs (functional deficiencies)
- Doesn't meet minimum quality of service levels (architectural deficiencies)
- Can't be implemented in the target runtime environment (environmental deficiencies)

As we will attempt to show, the capture of each of these dimensions will result in a better model of the target application. This model can then be transformed into an application that meets all of the specified requirements.

2.1. Characteristics of MDD

Because of MDD's potential to dramatically change the way we develop applications, companies are already working to deliver supporting technology. However, there is no universally accepted definition of the requirements for an MDD infrastructure and many requirements for MDD support are still implicit [6]. To overcome this lack of standardization, we argue that an MDD infrastructure requires precise, analyzable, transformable and executable models.

Model-driven development's defining characteristic is that developers focus on building models rather than programs. A model is a coherent set of formal elements describing a system built for a purpose that is amenable to a particular form of analysis [5]. A model depicts a system's structure and behavior at a certain level of abstraction. At the lowest level, models use implementation technology concepts. Models at a higher level of abstraction focus more heavily on business domain concepts and thus are less sensitive to the technical domain and to evolutionary changes to that domain.

A particularly desirable property of these models is that a modeler can navigate between the levels of abstraction in some uniform way. That is, that we can "drill down" through the different abstraction layers. This implies that we have a uniform set of modeling constructs that can scale from the very coarse-grained to the very fine-grained and back again. Therefore, we need modeling elements that facilitate the navigation between abstraction layers, but that are

¹ These are also referred to as non-functional requirements, or NFRs.

sufficiently powerful to be lossless in terms of content when performing such navigation. This ability to losslessly migrate between abstraction layers means that an appropriate level of detail or *view* can be chosen based on a target audience. Thus a business process expert might be provided a low-level model while an executive steering committee might interact with the same model at a much higher-level of abstraction.

A core premise behind MDD is that applications will be generated directly from their corresponding models. Thus one of MDD's key challenges is the transformation from high-level models to platform-specific applications. This is particularly true when we consider that not only must the functional requirements be captured and transformed appropriately, but also the extra-functional requirements that act as systemic constraints. Finally, the entire solution must be able to be realized for a particular runtime environment. To make these transformations a reality, each model must have a defined meaning. More particularly, each model needs to have a precise and analyzable formalism.

Having mere formal knowledge of models is not enough to gain a thorough understanding of the system being developed. One of the fundamental ways that developers learn is through experimentation, that is, through model execution. MDD should support the execution of a model in such a way that its behavior can be observed and analyzed. This technique is called model *animation*. There are precedents for such animation. For example, most IDEs offer debugging tools that allow application code to be interrogated, evaluated, and experimented upon during execution. We propose similar tools, but at a higher level of abstraction than code.

2.2. Meta-Modeling

Model-driven development automates the transformation of models from one form to another. We express each model, both source and target, in some language. We can define a modeling language's syntax and semantics by building a model of the modeling language – a so-called *meta-model*. A meta-model is the explicit specification of an abstraction. It uses a specific language for expressing this abstraction, such as the Object Management Group's (OMG) Meta-Object Facility (MOF). A meta-model defines a set of concepts and the relations between these concepts and is used as an abstraction filter in a particular modeling activity [7]. The notion of the meta-model is related to the notion of an ontology used in knowledge representation communities. We can extract models from a system by using specific meta-models or ontologies.

The OMG recently announced its Model-Driven Architecture (MDA) initiative, which provides a conceptual framework for defining a set of standards in support of MDD [8,9]. A key MDA standard is the Unified Modeling Language (UML), along with several other technologies related to modeling, such as MOF, CWM. The MOF provides a formal definition of the syntax and structure of UML models [10].

Although widely used, UML is only one of the meta-models on the software development landscape. Because of the different and incompatible meta-models being defined, there was a need for a global integration framework. The solution is a language for defining meta-models, that is, a meta-meta-model. In OMG's MDA framework, this is the role of the MOF, an abstract language and framework for specifying, constructing, and managing technology neutral meta-models. It is a unique and self-defined meta-meta-model. As a consequence, a layered architecture has now been defined as shown in Figure 1.

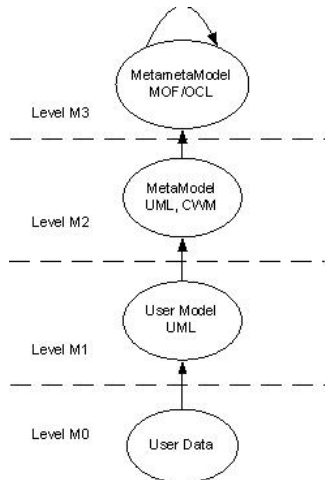


Figure 1. OMG Meta-Modeling Layered Architecture

This layered architecture has the following levels:

- M3: the meta-meta-model level (contains only the MOF)
- M2: the meta-model level (contains any kind of meta-model, including the UML meta-model).
- M1: the model level (any model with a corresponding meta-model from M2)
- M0: the concrete level (any real situation represented by a given model from M1)

One way to look at the architecture is to compare it to programming languages. Level M3 corresponds to a meta-grammar (for example, the EBNF notation). Level M2 corresponds to a grammar. Level M1 corresponds to a program. Level M0 corresponds to one given dynamic execution of a program. In the OMG modeling stack, MOF is at level M3. It is self-defined and allows for the definition of meta-models at level M2. The UML meta-model is one well-known example. It allows the definition of UML models at level M1. A given UML model describes a real phenomenon at level M0, with entities and events that are unique in time and space.

3. A Formal Foundation for Meta-Modeling

MOF provides the modeling and interchange constructs that support MDA, but it does have limitations. MOF is weak in terms of its semantics. MOF semantics are defined in terms of the MOF specification. Although a constraint language, OCL, is provided to define part of the MOF semantics, it does so in a disjointed manner. Because MOF itself lacks a formal foundation, nothing based on MOF will have such a foundation. Another limitation of MOF is evident when we consider a model's XFRs. For a model to be complete, all relevant characteristics and constraints must be captured and considered. However, few meta-models provide for the capture and verification of XFRs and MOF is no exception.

To address these limitations, we need a meta-modeling language that provides both formal syntax and semantics. This language must be extensible so that it can adapt to address issues such as XFRs. We propose an alternative approach to MOF that uses the ZOOM as its meta-modeling language. Not only does ZOOM have a formal foundation, but it is consistent with UML. ZOOM can be extended to include the appropriate information about the XFRs. While MOF could also be so extended, it lacks the formalism that ZOOM provides.

3.1. ZOOM

ZOOM stands for Z-based Object-Oriented Modeling. It is based on the formal specification notation Z [11,12], which is in turn based upon set theory and mathematical logic. Although widely used to specify software systems, one of Z's deficiencies is that its specification is limited to mathematical logic and does not provide useful mechanisms to support OO modeling such as classes or inheritance. ZOOM extends Z to support these object-oriented concepts. ZOOM brings formal foundations to object-oriented notations by providing textual and graphical representations of models that are consistent with UML. It brings formal semantics to models and provides a way to formally specify constraints such as preconditions, postconditions and invariants. To make it easy for practitioners to use, ZOOM adopts a syntax that is similar to commonly used programming languages such as Java or C++.

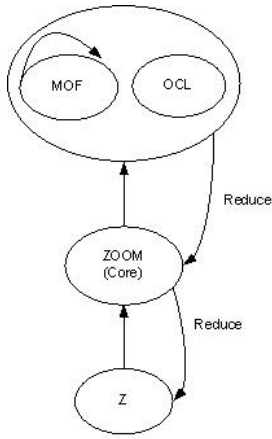


Figure 2. Z, Core-ZOOM and UML

As shown in Figure 2, Core ZOOM(ZOOM-C) is based on Z, and can be used to describe UML semantics. Similarly, UML semantics can be formally reduced to ZOOM and thus to Z. This provides a formal foundation to UML meta-modeling.

ZOOM can be extended to support structural models, behavioral models and user interface models. To distinguish it from these extensions, we call the part of ZOOM that can be used as a meta-modeling language ZOOM-C. Our goal is to use ZOOM-C to act as the semantic meta-model for UML.

3.2. ZOOM as a Meta-Modeling Language

ZOOM-C describes modeling elements with formally specified preconditions and postconditions. A specification in ZOOM-C consists of a set of declarations, state definitions, user roles and use cases. To demonstrate that ZOOM can be used to formally describe a meta-model, Figure 3 shows the ZOOM specification of the core of the MOF model.

```
typedef MM = ModelElement <=> ModelElement; //a relationship between ModelElements
typedef MN = ModelElement -> NameSpace; //a function map from ModelElement to NameSpace
MM Depends_On; //declare a global variable of type MM
MN Contain; //declare a global variable of type MN
struct ModelElement { //struct definition
    Public String name, annotation;
    Public : noaccess String qualifiedName;
    Set[ModelElement] dependent, provider;
    NameSpace container;
    Invariant { //invariant definition
        dependent == DependsOn.image(this);
        provider == DependsOn.invert().image(this);
        container == Contain.invert(this);
    }
}
struct NameSpace extends ModelElement {
    Set[modelElement] containElement { containElement == contain.invert(this) }
}
```

Figure 3. ZOOM Specification of core structure of MOF model

This example shows some of the characteristics of ZOOM. The syntax of ZOOM is similar to Java. It is object-oriented. It is strongly typed with a semantically rich type system that supports inheritance and generic types.

4. Formal Object-Oriented Modeling Using ZOOM

ZOOM addresses many aspects of a software system including the static and dynamic aspects, as well as the user interface. Unlike UML, which separates the system into several loosely related views, ZOOM provides a mechanism to merge those views together along with their formal semantics.

A ZOOM model has three integrated parts: structural models, behavior models and user interface (UI) models. An event model, processed through an event-driven framework, integrates them, as shown in Figure 4.

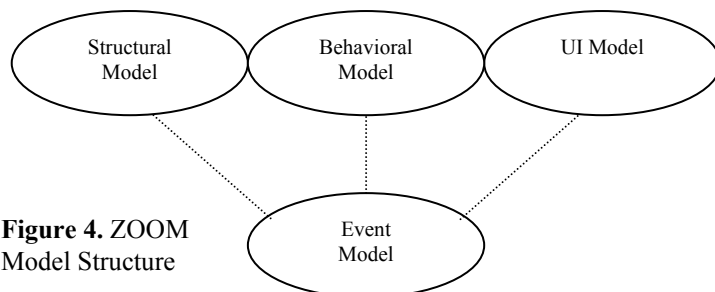


Figure 4. ZOOM Model Structure

One advantage of this separation of concerns is that we overcome the difficulties of specifying the UI

together with the structural model; the UI typically has a tight association to a specific platform while the other models can be platform-independent.

Each ZOOM model has dual representations including a textual specification and a graphical view. The graphical representations are consistent with common UML diagrams, such as class, use case and state diagrams, but also include formal semantics and syntax. This enables the use of popular tools to design and maintain ZOOM models. Modelers will appreciate the ability to use available tools to construct their models and to then add formal specifications to those models.

4.1. Structural Model

The *structural model* defines the functional part of the system using object-oriented concepts. Structural models consist of a series of diagrams along with the formal specifications described by ZOOM-C, including method preconditions and postconditions as well as class invariants.

Structural models provide not only the class relations and hierarchies, but also a precise specification of the functionality of each entity. Figure 5 provides a simple example for an *Address* class, which is part of the structural model for a larger system.

```
module zoom.examples.addressbook;

class AddressBook {
  String title;
  List[AddressBookEntry] entries;
  delegates entries; // delegates the operations to entries
  Set[AddressBookEntry] search(SearchCriterion criterion) {
    requires {
      criterion != null;
    }
    ensures {
      search! = { AddressBookEntry entry in entries |
                 entry.matchAddressBookEntry(criterion) @ entry }
    }
  }
}
```

Figure 5. ZOOM Notation Example

4.2. Behavior Model

The *behavioral model* is the central communication mechanism that links the structural models and UI models. It uses a formalized state diagram to specify the dynamic aspects of a system. This state diagram uses ZOOM-C to define the behavior of the system. Graphically, the representation of the behavioral model is consistent with a UML state chart. This approach makes it easier for practitioners to understand and design the behavior models.

In the behavior model, each action and its corresponding result is specified by pre- and post-conditions in a way that is similar to that used in the structural models. It supports both nested state and concurrent state change definitions. Figure 6 shows an example of a behavior model.

```

fsm MachineserviceStat() {
  state inService {
    fsm MachineopStat () {
      state standingBy;
      state turningOnMotor;
      state accelerating;
      state running;
      state decelerating;
      transition standingBy to turningOnMotor : operatorStartsMachine(runSpeed)
        {safetyBrake.turnOff(), mainMotor.turnOn(runSpeed, out runSpeedOK)};
      transition turningOnMotor to standingBy : standBy [runSpeedOK != TRUE]
        {mainMotor.turnOff(), safetyBrake.turnOn()};
      transition turningOnMotor to accelerating : accelerate [runSpeedOK == TRUE];
    }
  };
  state inRepair;
  state waitingForRepair;

  transition inRepair to inService : returnToService(fixDate);
  transition waitingForRepair to inRepair : beginMaintenance(maintDate);
  transition inService to waitingForRepair : removeFromService(reason)
    {maintShop.notify(self, reason)};
}

```

Figure 6. Behavioral Model Example

4.3. User Interface Model

The *user interface* (UI) model is the third component of a ZOOM model. In order to specify UI designs in a formal way, we define a hierarchical description framework in which designers can use predefined UI components to construct the structure and layout of a user interface. For each UI component, designers can specify related visual attributes and the events that can be generated or consumed by the component. The graphical view of the UI model provides a visual representation and is easy for designers to learn.

One advantage of using a formal and platform-independent way of specifying the UI model is that it can be used to construct platform-specific user interfaces across a wide range of operating environments. For example, a single UI model could be used to generate the application for a desktop computer, a mobile device or even an embedded system.

4.4. Event Model

To coordinate the activities between the structural, behavioral and UI models, ZOOM provides an event model that is processed by an event-driven framework. Events are used in the behavioral model to specify the dynamic nature of the system. Upon receiving specific types of events, the behavioral model's finite state machine (FSM) will perform predefined operations. Events are categorized as shown in Figure 7.

1. Event
 - 1.1. UI Event
 - 1.2. FSM Event
 - 1.2.1. Standard Event
 - 1.2.2. User-Defined Event

Figure 7. ZOOM Event Hierarchy

When users interact with the system through the user interface, UI events are generated and passed to the FSM. The FSM performs a series of state transitions based on the structural models. The FSM may trigger additional events, which include both predefined standard events and user-defined domain specific events. Those events are then sent back to the FSM, thereby driving the system.

5. Support for Automation

The most exciting benefit of the MDD approach is that designers use high levels of abstraction to build applications. This level of abstraction is closer to the customer's view of the application instead of the technologist's view. However, to achieve widespread support with software development practitioners, MDD must provide them the tools to help them perform their tasks.

Because ZOOM provides a formal foundation in its modeling language syntax and semantics, it is feasible to develop a set of tools to help build more reliable applications through automated processes. ZOOM comes with a set of tools. Most of the main tools, such as the code generator, animator and unit test generator can accept many UML diagrams as input, including class, state, and use case diagrams. Currently we have developed several tool prototypes including a static analyzer, automated theorem prover, the ZOOM interpreter and a model animator. Other tools are still in the development phase.

5.1. Basic Language Support Tools

These tools provide the basic language level support for ZOOM modeling including: lexers, parsers, type checkers, UML mapping and model editing.

The UML mapping tools transfer existing UML models into a ZOOM model. ZOOM has a rich syntax, representing a superset of UML concepts, so each aspect of UML can be described in ZOOM. Since UML does not provide truly formal modeling, this tool will check and transfer existing models into partial models, and ask the modelers to provide the formal constraints.

The model editor supports the construction and refactoring of structural models. During this process, it automatically maintains the structure and logical consistency of the formal object design, such as automatically inheriting invariants, contracts and unit tests from a superclass. Another capability of the model editor is reverse-engineering, which allows most manually implemented changes to be maintained if the models are ever refactored.

5.2. Model Validation Tools

The goal of model validation is to exhibit the behavior of both the structural and behavioral models at an early stage. Developers can use these tools to validate and verify their models during the design phase, when only partial models are available.

The interpreter is used to exhibit the behavior of imperative implementations in the models. The imperative implementation specifies the dynamic nature of the models using a subset of ZOOM called ZOOM-E. ZOOM-E provides syntactic constructs similar to those in Java.

The animator provides a way for designers to verify their models based on their formal specifications. The animator "virtually" executes models without an implementation by introducing a default model implementation that is consistent with the formal specification. For example, the animator can hypothesize a method body based on the pre- and post-conditions defined for that method. Although the animator does not provide an optimized execution of the model, it does allow the designer to identify design deficiencies at an early stage.

5.3. Model Verification Tools

One tool that differentiates ZOOM from other software development environments is the Automated Theorem Prover (ATP). The ATP supports ZOOM's formal analysis, but can be used independently to do logical proving. The ATP in ZOOM is optimized for the ZOOM modeling language and provides a powerful reasoning mechanism for static analysis.

The static analyzer checks the consistency, validity and completeness of model components to ensure the model is both executable and reliable. The static analyzer can check the partial implementations provided by the developer, and verify their conformance to the specification.

The unit test generator creates suitable unit test cases based on different coverage criteria. This tool makes it possible for developers to concurrently develop both the models and their tests to help the verification and validation processes.

5.4. Transformation Tools

Transformation tools transform models from one level of abstraction to another.

The code generator creates code based on the ZOOM model. Traditional code generation focused mainly on constructing the skeleton of an application based on some templates and leaving “hooks” for developers to fill in. In the MDD world, code generation is based solely on the model with some customized parameters. Because of its formal characteristics, ZOOM can derive the code from the models’ formal specifications without interactive guidance. For example, ZOOM can generate a method body based on the formal pre- and post-conditions defined in the method’s models. This is one benefit of applying formal methods in MDD as it can provide more reliable code than before. In addition code generation also considers the XFRs and the target runtime environment that have been modeled. This allows “smart” code generation that can adapt its choice of implementation and algorithm to meet the specified performance or security constraints or to ensure that it can execute in the appropriate target environment.

6. Comparison to Other Approaches

ZOOM provides both formal syntax and semantics. This formalization is based on Z, which is known for its ability to specify software functionality. While OMG’s MOF does provide a formal syntax for UML in the form of OCL[8,9], it lacks truly formal semantics. With its formal nature, ZOOM makes the MDD process more powerful and more convenient by allowing developers to focus on building models and not on the underlying formal mathematics.

ZOOM provides a high level of abstraction. The OMG approach to MDD still requires imperative languages to specify the functionality of the system. ZOOM uses a declarative language to specify its models. Through a set of tools that incrementally validate the behavior of these models, a complete and formal model emerges. This model can then be transformed across abstraction layers and ultimately into an implementation for a target platform.

ZOOM separates views from models, making it possible to formalize both the system’s functionality and user interface. This allows ZOOM to construct the same application for different platforms.

ZOOM provides a set of automated tools to support the design and development process. These tools take the advantage of the formal foundations of ZOOM. These tools conduct the model transformations and turn the models into real applications. They support iterative and incremental software development.

7. Conclusion

Model-driven development is a positive step toward a general model-based approach to software engineering, but current frameworks lack a formal foundation. Our formal foundation in support of MDD is ZOOM, a meta-model based on Z, which can be used to describe UML semantics. We believe that such a foundation strengthens and facilitates MDD by allowing a precise and analyzable way to model the structural, behavioral, and user interface aspects of an application.

We describe how formal semantics can positively influence the design process by allowing partial models to be constructed and verified before all of the analysis and design has been completed through a process called model animation. This approach supports iterative and incremental development by allowing the developer to refine and experiment on the model before committing to an un-tested solution. This animation is only possible because of the formal foundations within the ZOOM meta-model.

The ZOOM meta-model includes the static, behavioral and user-interface models as well as the communication mechanisms between them in the form of the event model and event-driven framework. We believe that allowing for the formal description of user interface requirements provides a significant benefit by decoupling its description from the other models and by allowing platform-specific user interfaces to be created from a platform-independent model.

The architectural and platform requirements provide significant value during model analysis and code generation. In particular, the extra-functional requirements impact the choice of implementation and algorithm during code generation.

ZOOM supports many tools. These tools include an automated theorem prover, model animator, static analyzer, and interpreter. Other tools such as the unit test generator are either planned or currently in development. These tools serve to make MDD in general, and ZOOM in particular, more accessible to the software development community.

References

- [1] J. Mukerji and J. Miller, Model Driven Architecture www.omg.org/cgi-bin/doc?ormsc/2001-07-01
- [2] D.S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, OMG Press, 2003
- [3] Object Management Group (2001) Model Driven Architecture (MDA), 9 July 2001 draft, edited by J.Miller and J. Mukerji, available at <http://doc.omg.org/ormsc/2001-07-01>
- [4] J. Poole Model-Driven Architecture: Vision, Standards And Emerging Technologies ECOOP 2001
- [5] J. Woodcock, J. Davies, Using Z Specification, Refinement, and Proof, Prentice Hall Europe 1996
- [6] J. B. Wordsworth Software Development with Z Addison-Wesley 1992
- [7] S. Mellor, A. Clark and T. Futagami Model-Driven Development IEEE Software No.5. Volume 20, 2003
- [8] Colin Atkinson and Thomas Kühne., Model-Driven Development:A Metamodeling Foundation IEEE Software September/October 2003 (Vol. 20, No. 5)
- [9] J Bezivin Towards a Precise Definition of the OMG/MDA Framework IEEE Proceeding of the 16th ASE 2001
- [10] Object Management Group (2002) text for an MDA Guide, June 2002, available at <http://doc.omg.org/ormsc/02-06-02>
- [11] Object Management Group and R. Soley, Model Driven Architecture 2000. OMG document available at <http://www.omg.org/mda>
- [12] Object Management Group (OMG). Meta-object Facility (MOF), Version 1.4. <http://www.omg.org/technology/documents/formal/mof.htm>.