Multi-level Language Descriptions

Andreas Prinz

Department of ICT, University of Agder, Grimstad, Norway
andreas.prinz@uia.no

Abstract. Language descriptions are a multi-level issue. In particular, the definition and handling of instantiation semantics connects three levels. This paper looks at two language workbenches without support for multi-level modelling and their handling of the multi-level part of language descriptions. From these observations, the importance of runtime instantiation in terms of an underlying machine is established.

1 Introduction

Levels in modelling and in programming are typically considered in a hierarchy that is given by instantiation. Generally, the main idea of multi-level modelling is to provide ways to not only influence the instances of a model, but also the instances of these, as observed in [5]. In the area of language description, this concerns instantiation semantics, which is about instantiation of the instances.

Therefore, instantiation semantics, also known as structural semantics, is a multi-level issue. The same is true for execution semantics, which is based on the instantiation semantics since executions are sequences of instance structures. This paper will not discuss execution semantics; the interested reader is pointed to [17] for more information.

The problem of instantiation semantics is approached by observing two tools, LanguageLab [9] and MPS [16]. Both tools are language workbenches, and not multi-level modelling tools. However, they handle instantiation semantics, which is a multi-level issue.

The experience with the tools is used to propose a framework for the discussion of instantiation semantics and multi-level modelling. The framework is used to revisit two old problems of non-multi-level approaches as discussed in [2]. First, using only two levels leads to ambiguous classification, in that it is difficult to place the concepts on the correct levels. An example from UML are the two concepts *Node* and *NodeInstance*, which should be on the same level, as they both belong to the same language, but need to be on different levels, as they describe aspects that are on different levels. The second problem concerns the replication of model elements, mostly due to the fact that instantiation is available between every two levels and this has to be described on each level.

This kind of problems has lead to the idea that the OMG architecture is broken and has to be fixed. Multi-level modelling [2] came to the rescue, allowing instantiation over several levels. Several approaches for multi-level modelling have been proposed, among them orthogonal (linguistic plus ontological) approaches

[3], deep instantiation (potencies and clabjects) [2], and powertype-based approaches [11]. They solve the problems with the architecture, but also bring a new level of complexity.

Our experience with the tools shows that language descriptions can be handled in the OMG architecture based on the concept of runtime environment.

This paper continues with Sect. 2 which explains levels and their relation to language descriptions. Section 3 looks into tool handling of instantiation semantics. In Sect. 4, underlying machines and instantiation are considered. Section 5 introduces runtime environments, leading to a discussion of the two problems in Sect. 6. Finally, the paper is summarized in Sect. 7.

2 Metamodelling and MDA

OMG has defined model-driven architecture (MDA) for using models in the development process, see [12]. MOF [7] and UML [18] are key languages of the MDA, but MDA is open for other languages. MDA is based on an understanding of a four-level hierarchy of abstractions as illustrated in the table below. M0 is

OMG Level	Examples	Grammar example	OCL example
M3 = meta languages	MOF	EBNF	MOF
M2 = languages	UML metamodel	Java grammar	OCL language
M1 = models	UML model	a program	a formula
M0 = instances	objects of UML classes	a run	a truth value

the lowest level. It contains concrete (runtime) objects. The next level (M1) includes the models that describe the M0 objects. On top of M1 there is a level describing how models are formed, which is a meta-model or language level, called M2. Finally, the architecture is closed with a level M3 (meta-language) that is supposed to both describe M2 as well as describing itself.

2.1 Definition and Use

The relation between levels is the definition-use relation, where the higher level provides the definition and the lower level includes the use. This is the instantiation relation (meta-relation) as for example stated in [8, 21]. As an example, the definition of a program on M1 is related to a use (a run) on level M0. In [3], this level-crossing relation is called linguistic instantiation. The second related dimension, called ontological instantiation, is discussed in Sect. 6.

Definition and use refer to roles, not to absolute properties. This means that the same entity can be both use and definition depending on context, which leads to the notion of clabjects in [1]. This way all the four OMG levels are spanned by the definition-use relation, connecting two adjacent levels thereby raising an arbitrary number of levels, see also [14]. With the definition-use relation only crossing one level boundary, MDA is not a multi-level approach per se.

The definition-use relation is also known as type-element pattern, where a definition (type) gives rise to a number of uses (elements). The connection between definition and use is provided by a semantic function, associating the definition with a set of possible uses. The definition-use relation also appears between compile time (definition) and runtime (use). Typically, the definition is read-only while the use is read/write.

2.2 Language Descriptions

Formal language descriptions on OMG level M2 have the aspects structure, syntax, and semantics ([14]).

Language *structure* defines the concepts of the language and their relation to each other, maybe as a MOF class diagram. This way, the structure aspect allows describing all possible instances in a *generative* way. In addition, constraints *restrict* the possible instances by rejecting some of them as invalid.

The *syntax* aspect can have different kinds of presentation, e.g. *textual*, *graphical*, and *tabular* presentations and a mixture of them. It is often defined by the specification of a presentation domain and a one-to-one mapping between presentation domain and structure.

The *semantics* (meaning) can be given in several different ways. For example, semantics can define language instance *execution* (operational semantics), or define a *mapping* into another language (transformation semantics). An important part of operational semantics is instantiation semantics, i.e. the way how instances of the language are instantiated.

Each language aspect requires a meta-language for its description, which amounts to a potentially large number of meta-languages. MOF is an example of a meta-language for structure, while OCL is an example for constraints.

In this paper, instantiation semantics is most important. It is defined by the language itself (on M2) and might depend on the actual program (on M1). It describes how to create objects at runtime (M0). This way it is a multi-level issue. In fact, none of the other language aspects are multi-level, as structure, syntax, and transformation do only cross one level boundary.

With instantiation semantics, the language bridges three levels, see [15].

1. the language specification, or the meta-model,
2. the user specification, or the model, and
3. objects of the model.

After the two-level definition-use, this introduces a three-level pattern that is repeated in MDA: M2-M1-M0, M3-M2-M1, M3-M3-M2.

3 Levels and Instantiation in Tools

3.1 MPS

Meta-Programming System (MPS) [16] is a tool to define domain-specific languages. It is geared towards professional developers and is often used in a Java

environment. It is written in Java, but can be used with other languages as well. The mbeddr project [20] has applied MPS to programming of embedded devices in a C/C++ environment.

The general philosophy of MPS can be explained in connection with the OMG four-level architecture. The meta-languages (M3) in MPS are predefined by the platform and fixed. Being a professional tool, MPS has a large variety of meta-languages to describe all aspects of languages.

The first main activity in MPS is the description of a language on M2 using the M3 languages. This language description is translated by MPS into MPS internal Java code thereby creating an IDE for the language described. An example is the definition of a Petrinet language with the MPS meta-languages for structure, constraints, text syntax, transformation, intentions, and behaviour.

The second main activity in MPS is the definition of programs or specifications on M1 in the IDE for the language defined on M2. MPS provides a projectional editor starting from the internal objects and showing either a default view or a language-defined view. If the programs written should be executable, a transformation has to be provided, typically into Java. If this is the case, then user programs are transformed into standard Java programs.

Finally, MPS does not care for M0, as it relies on the built-in execution and instantiation as given by Java (or C++, if C++ code is generated). This way, M0 need not be handled in MPS.

For the levels M3-M2 and M2-M1, MPS uses its built-in instantiation semantics, which is accessible using an internal language called S (for structure). For instantiation between M1 and M0, the instantiation semantics of the generated code is used.

3.2 LanguageLab

LanguageLab [9] is an educational tool for language description theory. Only a few languages exist in LanguageLab and they are related to teaching or to the platform itself. LanguageLab provides very simple meta-languages for structure, textual syntax and transformation, thereby being much simpler than MPS.

An example is the basic structure meta-language, which is defined using the basic meta-languages for structure, text syntax and transformation.

LanguageLab favours a relative approach and does not use absolute levels as MDA. In particular, all the levels M3, M2, M1, and M0 are similar in LanguageLab. This is achieved by a modular approach, where the main entity is a module, which can be a language or a specification or a meta-language. Modules come with two sets of interfaces: provided interfaces (lower interfaces) in the sense of definitions, that are available for use, and required interfaces (upper interfaces), which are uses of definitions on the next upper level. This way, LanguageLab mirrors the definition-use relation.

LanguageLab is implemented in Java, but the user has no access to the implementation. The user only sees the languages provided. This is achieved by the definition of a platform, which is the underlying abstract runtime machine of LanguageLab. This platform provides instantiation, presentation and mapping

primitives and thus allows languages to be defined and executed. LanguageLab uses the same platform instantiation between any two adjacent levels.

3.3 Bootstrapping

Both MPS and LanguageLab claim to be bootstrapped, i.e. the meta-languages given in them are defined in the platform itself.

LanguageLab has two parts of the tool, the platform and the languages. The semantics of the languages is provided by mapping them to the elements of the platform. So an editor description is mapped to a runtime editor in the platform. Normally, LanguageLab modules are defined by other LanguageLab modules, but it is not a problem to define them by themselves. In this case, the upper module is read-only, while the lower one is read/write. For bootstrap, the meta-languages are defined using a read-only version of themselves, which is then translated to the platform primitives. Execution in the platform is given by the underlying LanguageLab machine.

MPS bootstrapping is more tricky. The MPS meta-languages are translated into Java. This Java code is related to several interfaces and stubs for the MPS tool platform. When these interfaces are used, the platform ensures the correct result. In addition to the interfaces, MPS also uses the specifications in the languages for cross-referencing and similar purposes.

In order to achieve bootstrapping in MPS, the previous generated code is stored and used for generating the new version of the code. Then, the new code is compiled and a new version of the platform is provided. Again, the old version of the module is used in a read-only way in order to produce the new one. The trick is to generate the same code as before, in particular the same concept identifiers, because otherwise it would not be possible to connect the old instances of concepts to the new generated concepts.

4 Machines: MOF-VM

Semantics, in particular instantiation semantics, is based on an underlying mechanism that provides basic execution and instantiation. This mechanism could be very low level as in machine code, where an indication of a memory area leads to the provision of an actual memory (simple instantiation). It could also be more high level as in a virtual machine that features class instantiation. In this paper, the basis is a mechanism that provides general object-oriented instantiation with the name MOF-VM (see also [10]). This is similar to the underlying instantiation in MPS and LanguageLab, see also Fig. 1. MOF-VM is a virtual machine that has instantiation semantics like MOF, which means that classes defined in the MOF-VM can be instantiated into objects. This way, all existing instances are internally MOF-VM objects, including the objects on M2, M1, and M0.

Figure 1 shows the underlying MOF-VM instantiation on the right-hand side. A class *RT_ActiveClass* is instantiated and yields an object *:RT_ActiveClass* with the attribute *name* being *"Factory"*. That instantiation is internal.

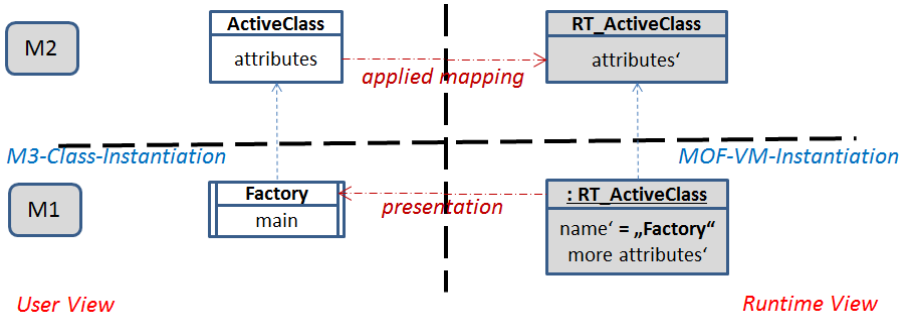

Fig. 1. MOF-VM instantiation versus language instantiation.

The left-hand-side shows the language instantiation with *Factory* being an instance of *ActiveClass*. While the right bottom shows the default presentation coming from the platform, the left bottom shows the language-defined presentation. Both presentations show the same internal object. The last relation in the figure called *applied mapping* is explained in Sect. 5.

In Fig. 1, language instantiation is a composition of three relations: *applied mapping*, *MOF-VM instantiation*, and *presentation*. The MOF-VM instantiation is built-in and provides the level-crossing relation in the OMG architecture. By using the MOF-VM instantiation, language definitions do not need to cross levels in order to define their (linguistic) instantiation.

Instantiation in MOF as a language works in the same way - using the built-in MOF-VM instantiation. MOF-VM instantiation is related to the MDA platform definition and not the language definition of MOF. This means that apart from linguistic and ontological instantiation, there is also MOF-VM (runtime) instantiation, which is used to achieve linguistic instantiation. It is crucial to keep these three apart when trying to understand instantiation semantics.

An important point is the presentation in Fig. 1. The objects on the right-hand side (grey) do not have any presentation, because they are objects of the underlying machine (MOF-VM). The notation used in Fig. 1 is therefore just an ad-hoc presentation of these objects.

Even though MOF-VM provides execution and instantiation, it is abstract. There will be a *target platform* below MOF-VM used as concrete real machine for execution. There are known and proven techniques and methods to achieve this mapping onto a real machine, and this paper does not go into these details.

5 Runtime Environment

As discussed in Sect. 2, each language has to define how to instantiate its elements, e.g. what are instances of classes, modules, methods, and variables. The structures (possible states) existing at runtime are commonly called runtime environment (RTE). RTE states are purely structural and runtime state changes

are based on them, see also [19]. The read-only program itself is somehow available in the runtime state, such that the execution can refer to the program. The definition of RTE belongs to the language, i.e. level M2. But also the specification might influence the RTE, such that the definition has to be done on level M1. In order to sort out the levels, we distinguish several kinds of elements in the RTE as follows.

- *Global elements* are runtime elements coming from the language, e.g. predefined libraries, program counter, and exception storage. They are fixed by the language on M2, and are independent of the specific program.
- *Local elements* relate to language concepts and describe how these are instantiated at runtime. They are related to their respective concepts, but are fixed on the level of the language (M2), like storage areas for static variables. For any language concept, the instantiation can yield none (1:0), one (1:1), or many (1:n) runtime elements depending on the language.
- *Dependent elements* are similar to local elements, but they depend on the specific program. Objects of classes and stack frames for methods are two examples. They cannot be defined statically on level M2. Instead, on M2 a mapping from the language concepts to the runtime structures can be defined. Again, the mapping can be 1:0, 1:1 and 1:n.

In Fig. 1, the instantiation from M2 to M1 is defined a (meta-)language on M3. MOF-VM knows by means of *applied mapping* that it should use *RT_ ActiveClass* in order to instantiate *ActiveClass* (Fig. 1). The *applied mapping* (M2) is a use of the *defined mapping* (M3), see Fig. 2. All the different kinds of runtime elements can be captured with such a *defined mapping*.

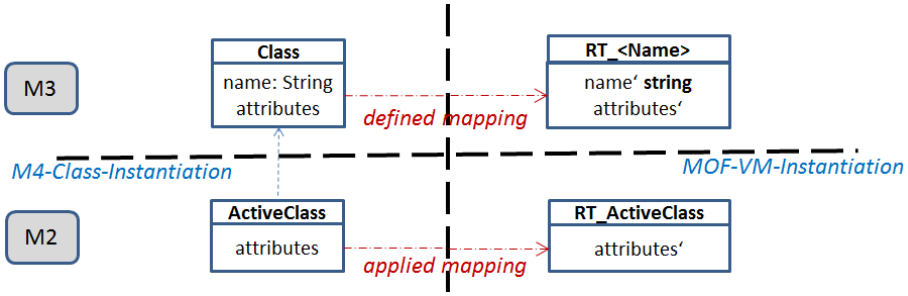

Fig. 2. Defined mapping versus applied mapping.

Similarly, the instantiation semantics of a language description (RTE structure) is defined at M2 as a mapping from the program on M1 to the MOF-VM classes on M1, which then get instantiated at M0 as MOF-VM objects.

6 Solving Multi-level Modelling Problems

Based on the concepts presented in Sect. 5, we revisit the two problems mentioned in the introduction: use of non-multi-level approaches leads to (1) ambiguous classification and (2) replication of model elements.

MPS and LanguageLab are not multi-level, but still they do not have these problems. What is their solution? For the problem (2), both tools use a relative approach, where languages are not placed on fixed levels, but definitions of languages are placed relative to their uses.

For the solution to problem (1), we have to look more closely into the three kinds of instantiation. Figure 3 shows the combination of the relations *defined mapping* (D), *applied mapping* (A), and *presentation* (P). On the left of the figure,

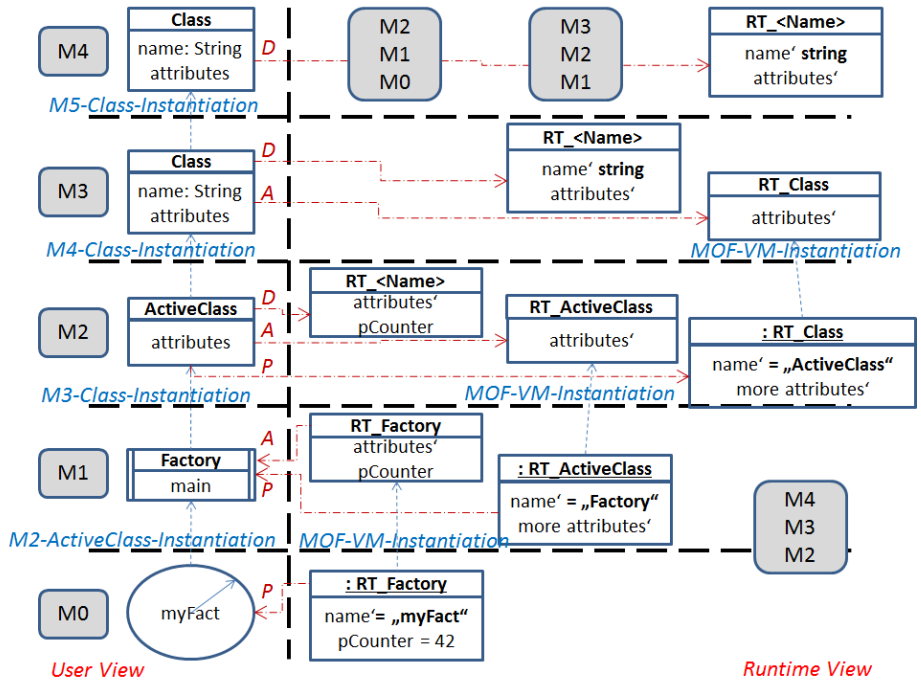

Fig. 3. Repeated Application of Language Definition.

the user view of instantiation is presented. On level M0, a dummy presentation of the runtime object for *Factory* is used, in order to make the figure complete. Most often, the objects on M0 have no language-defined presentation. To the right, there are the three instances of the language definition and use pattern stretching over three levels each: M2-M1-M0, M3-M2-M1, and M4-M3-M2. All level-crossing on the right part is MOF-VM instantiation. A similar pattern is

also used quite often in [13], and it is the natural way to think when using language-oriented programming [6].

The important part of Fig. 3 is the level M2, as here we have three relations at the same time. The same would be true for any higher levels, but Fig. 3 does not show all these relations. In particular, there is a presentation relation on each level, connecting the runtime view and the language view. These two are separate, therefore there is no ambiguous classification. Clabjects show these two in one presentation, which blurs the difference between them. BTW, clabjects would look differently on levels M1 and M0.

When it comes to expressing the mappings, there are many ways to do this. A simple way to define a mapping for local elements and maybe even for global elements are attributes with potency 2 in a deep modelling context.

A modelling language on M2 may provide as many ontological levels as needed, see also [4]. Its RTE has to define instantiation for each of them, indicating what elements can appear on level M0, based on runtime instantiation. This way ontological language levels can be captured with the MDA architecture.

7 Summary

This paper has reviewed the multi-level needs for language descriptions, in particular instantiation semantics. MOF-VM (runtime) instantiation as a basis for linguistic instantiation was found to be the backbone of the OMG four-level architecture and strict meta-modeling.

The definition of runtime environment is essentially the definition of instantiation semantics. RTE is not defined out of thin air, but related to an underlying machine, which is MOF-VM in this article. Several possible kinds of instantiation relations between specification and RTE were identified. They are specified for a language as a mapping between specification and MOF-VM, which is defined at language level and used at specification level.

There are three kinds of instantiation: linguistic, ontological, and runtime instantiation.

References

1. Colin Atkinson and Thomas Kühne. Meta-level independent modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, 2000.
2. Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *UML 2001-The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 19–33. Springer Berlin Heidelberg, 2001.
3. Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *Software, IEEE*, 2003.
4. Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. *Proceedings of ASE'01, Automated Software Engineering*, 2001.

5. Tony Clark, Cesar Gonzalez-Perez, and Brian Henderson-Sellers. A foundation for multi-level modelling. In *Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014.*, pages 43–52, 2014.
6. Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004. URL: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf, accessed 2015-06-05.
7. OMG Editor. OMG Meta Object Facility (MOF) Core Specification Version 2.4.2. Technical report, Object Management Group, 2014.
8. Jean-Marie Favre. Meta-model and model co-evolution within the 3D software space. In *Proceedings of ELISA 2003*, September 2003.
9. Terje Gjøsæter and Andreas Prinz. Languagelab 1.1 user manual. Technical report, University of Agder, 2013.
10. Terje Gjøsæter, Andreas Prinz, and Jan P. Nytun. MOF-VM: Instantiation revisited. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, pages 137–144, 2016.
11. Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1):72–90, 2006.
12. Anneke Kleppe and Jos Warmer. *MDA Explained.* Addison–Wesley, 2003.
13. Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, December 2014.
14. Liping Mu, Terje Gjøsæter, Andreas Prinz, and Merete Skjelten Tveit. Specification of modelling languages in a flexible meta-model architecture. In *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Companion Volume*, pages 302–308, 2010.
15. OMG Editor. *Unified Modeling Language: Infrastructure version 2.4.1 (OMG Document formal/2011-08-05).* OMG Document. Published by Object Management Group, http://www.omg.org, August 2011.
16. Vaclav Pech, Alex Shatalin, and Markus Völter. JetBrains MPS as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 165–168. ACM, 2013.
17. Andreas Prinz, Birger Møller-Pedersen, and Joachim Fischer. Object-oriented operational semantics. In *Proceedings of SAM 2016, LNCS 9959*, Berlin, Heidelberg, 2016. Springer-Verlag.
18. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Model Language Reference Manual, second Edition.* Published by Pearson Education, Inc., 2005.
19. Markus Scheidgen and Joachim Fischer. *Human Comprehensible and Machine Processable Specifications of Operational Semantics*, pages 157–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
20. Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. Mbeddr: Extensible languages for embedded software development. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 13–16, New York, NY, USA, 2014. ACM.
21. Paul Sammut Tony Clark, Andy Evans and James Williams. *Applied Metamodelling. A Foundation for Language Driven Development.* Xactium, 2004. Available at `http://www.xactium.com`.