A Representation-Theoretical Analysis of the OMG Modelling Suite

Cesar GONZALEZ-PEREZ and Brian HENDERSON-SELLERS
Department of Software Engineering,
Faculty of Information Technology,
University of Technology, Sydney, Australia

Abstract. Conceptual modelling in software engineering is dominated by the implicit semantics of UML, some other OMG products, and a bunch of related modelling tools. Although other alternatives to modelling are seen in academic circles, industry is dominated by this single approach. With no room to improve and no freedom to experiment, stagnation is guaranteed. The paper claims that a technology-free, purely theoretical analysis of models and modelling is necessary in order to find what real models are about, what the real modelling mechanisms behind software engineering are, and what kind of modelling infrastructure should serve as a foundation for future modelling technologies.

Keywords. Conceptual modelling. Representation theory. UML. Metamodelling.

Introduction

Modelling is usually linked to representing, describing and explaining concepts and systems. Similarly, the idea of a model is often related to the notions of reproduction, specification and description. Although these instinctive connections are useful for many every-day activities, software engineers often need to go beyond them and into a more formal understanding of models and modelling. Current trends such as Model-Driven Architecture (MDA, [9]) or other model-centric approaches are positioning modelling under the spotlight more than ever before.

This paper delves into what models are and what modelling means, approaching it from a technology-agnostic perspective so that unnecessary constraints are avoided. Works such as (Seidewitz, 2003) are valuable in their analyses, but their heavy reliance on specific solutions (the OMG suite of standards in the cited case) makes their value a small fraction of what it could have been. In our work, we present a purely theoretical discourse that establishes a basic conceptual set and a collection of modelling needs and, as an example of its application, analyse the OMG suite.
1. Models and Modelling

From a simplistic point of view, we could say that a model is a statement about a given subject under study (SUS), expressed in a given language. This definition is similar to that given by (Seidewitz, 2003) (“a set of statements about some system under study”), but it incorporates an explicit reference to the language the model is expressed in. If we accept this definition, then the sentence “today it’s raining” would be a model of the real world, since it is a statement about a given subject (my perceived real world), expressed in a given language (English). Since today it’s raining (I can verify it looking through the window), the statement expressed by the sentence is true, and so the prospective model would be a valid model. Figure 1 shows a model and the SUS that it represents.

![Figure 1. A model and the SUS that it represents. The arrow can be read as “represents”.
](image)

We could call the sentence “today it’s raining” a model, but it would be of little use. The value of models usually resides in our ability to reason about the SUS by looking at the model only; this is sometimes called abstraction. At the same time, we need to acknowledge that modelling is often performed to fight complexity, which usually appears in the form of SUS that are not simple monolithic entities but intricate composites. This applies to function (e.g. different tasks in a workflow description, transaction against an online shop) as well as structure (e.g. the parts of a car, the hierarchy in an organisation). Therefore, we can say that the major reason why we need models is to reason about the complexity of the SUS without having to deal with it directly. As a result, a suitable model would have to exhibit the appropriate structure for it to be useful. For this reason, we prefer to say that, for a statement about an SUS (expressed in a given language) to be a real model, it needs to be homomorphic with the SUS that it represents. This means that the structure of the model and the structure of the SUS must coincide to some degree, and the operations that are possible on the SUS are also possible on the model. For example, consider a car and the parts it is made of. This is the subject under study. By looking at it, we can (albeit with a big effort) enumerate all its parts, sub-parts, etc. recursively in a tree-wise fashion. A model that represents a car and its parts should exhibit the same structure and allow for the same operation, i.e. enumerate parts and sub-parts recursively (Figure 2).

![Figure 2. A model must be homomorphic with its SUS so the operations that are possible on the SUS are also possible on the model.
](image)

We have established that models represent SUS. This statement, however, raises two additional questions. First of all, what kind of entities are the model and the SUS? Secondly, what is the nature of this “representation”? In order to answer the first question, we need to take into account that the usual depiction of modelling activities that is found in most
software engineering works suffers from a very peculiar problem: they show the model as being a representation of a fragment of reality (whatever “reality” means) and external to it. A very good example can be found in (Martin and Odell, 1992), Figure 5.1. Although this may seem a simple matter of notation, it reveals the underlying assumption that a model is somehow different to its SUS. However, even the most simplistic thought experiment would show that, if a SUS is part of reality, then a model of it is also part of reality, as vividly explained by (Meyer, 1997), p. 230-231. Therefore, models are not external to reality but components of it. Precisely because of this, we can create models that represent other models, thus creating a model chain, as will be discussed later.

Answering our first question, we can conclude that anything can be a SUS, and that a model, once created, becomes part of reality and, therefore, is a potential SUS for further models. With regard to our second question (what is the nature of the connection between a model and its SUS), two different scenarios are often found, as described by (Seidewitz, 2003). Sometimes a model is created to describe an existing SUS and enable reasoning about the SUS in its absence. For example, a training flight simulator is a model of a real aeroplane that is better suited for training pilots than the SUS it models. Some other times, a model is created to specify a SUS that does not exist. For example, blueprints of a building are created to define how the building will look like and how it will be built. We will call the former backward-looking models since they look backward (as far as temporality is concerned) to the SUS they model, and we will call the latter forward-looking models since they look forward into the future.

Software engineering makes use of both kinds of models. At the beginning of a software development project, models of an application domain are created. These models try to represent an existing reality, and for this reason they are backward-looking models. However, the last stages of a project involve models that specify the final system to be built and help us create it by detailing every single aspect of it. These are forward-looking models. At some point during the project, focus shifts from backward-looking to forward-looking (Figure 3). We can observe this by realising that the main reason for creating backward-looking models is to get rid of unnecessary detail so that the resulting model is simpler than (but, hopefully, homomorphically equivalent to) the real thing; on the other hand, the main reason why we create forward-looking models is to explicitly document as much detail as possible about a SUS that we intend to create. Detail is actively removed when looking back, but actively added when looking forward. We may assume that the nature of these two kinds of details is not the same. This issue, together with the mechanisms by which the back-to-forward shift happens, are certainly interesting topics by themselves, and will be considered as the theme for further research.
At some point during the project, focus shifts from looking back to the application domain into looking forward to the final system to build.

For both types of models (backward- and forward-looking), homomorphism dictates that the structure of the model must match the structure of the SUS at some relevant level of detail; in other words, for each relevant entity in the SUS there must be an entity in the model that plays the same structural roles. What is the nature of the connection between a relevant entity inside the SUS and its “surrogate”, homomorphic entity inside the model? We will use the term “interpretive mappings” to refer to the collection of information that allows finding out these connections when necessary.

Interpretive mappings are not always one-to-one relationships. In fact, the “cardinality” of the mappings depends on the characteristics of the representation process employed to create the model. For example, an architectonic scale model of a building usually contains a simplified, small-scale version of each room of the real building. In this case, each small room in the model is mapped to a real room in the real building; the cardinality is one to one. If the same model contains a figurine to exemplify how people would interact with the building in the real world, this figurine does not correspond to any particular person in the SUS, but to the prototypical idea of a person in the SUS interacting with the building. A number of persons in the SUS can play the structural role that the figurine plays in the model and, therefore, the interpretive mapping between the figurine and the persons it represents is one-to-many. There is a second way in which an interpretive mapping can be one-to-many; consider a label attached to the above mentioned architectonic scale model that reads something like “persons walk through this way”. Rather than incorporating a prototype of potential SUS entities, this model entity declares what kind of entities in the SUS are suitable for interpretive mapping. Summarising, we can identify three different kinds of interpretive mappings:
• **Isotypical** mappings are those that map one model entity to one SUS entity. The model entity straightforwardly represents the SUS entity.

• **Prototypical** mappings are those that map one model entity to a set of SUS entities given by example, i.e. the model entity exemplifies the kind of SUS entities that can be mapped to it.

• **Metatypical** mappings are those that map one model entity to a set of SUS entities given declaratively, i.e. the model entity is a description of the properties that SUS entities must comply with in order to be mapped to it.

Figure 4 shows examples of the three kinds. Notice that, in principle, there is nothing that prevents the three kinds to coexist within the same model. Also notice that the cardinality on the side of the model is always one, meaning that for a given SUS entity, at most one model entity will be mapped to it.

We must clarify that these three kinds of interpretive mappings are disjoint. This is usually clear, but some scenarios can result in confusion. For example, imagine a model that includes the following sentence “the cats living in 3/21 Walker Street, Lavender Bay”. This sentence fragment is a metatypical representation of a collection of cats, since it declares what kind of object it refers to (cats) and the properties that objects of this type must exhibit in order to comply with the description (i.e. living at 3/21 Walker Street, Lavender Bay). However, our knowledge says that only one cat lives at that address, so we could interpret the statement as an isotypical representation. This, however, is not correct, since the statement is not portraying a cat living at that address but declaring cats living at that address. The fact that only one cat happens to comply with the representation is circumstantial and does not change the kind of interpretive mapping.

2. OMG’s Suite

2.1. **UML Class Diagrams**

We can find these three kinds of mappings in software engineering. Consider a UML class diagram, for example. This diagram depicts a model and, as such, can be thought of as a model of that model. Each box in the class diagram that a UML-aware expert identifies as “a class” is really a representation of a class in the class model. The box on the paper is just a visual artefact that represents a conceptual construct (the class). We must emphasise here that boxes on the paper are *not* classes but *represent* classes. Therefore, each box on the
diagram is isotypically mapped to the corresponding class in the model. Because this is an isotypical mapping, the cardinality is one to one, and we often equate the box on the paper to the class in the model, thus confounding the model and its representation. This is convenient but does not mean that the box and the class are the same thing. Now, each class in the model is a declaration of what kinds of objects can appear in the system being modelled; therefore, each class in the model is metatypically mapped to a number of objects in the running system. At the same time, we could argue that each class in the model is isotypically mapped to a class in the source code. Finally, let’s consider what UML 1.5 (OMG, 2003b) calls object diagrams, i.e. structural diagrams in which objects can appear. A box drawn on a piece of paper that is identified by an expert as “an object” is an ambiguous thing. Some people would interpret it as a representation of a particular object in the running system (an isotypical mapping) while others would interpret it as an example of an object that may occur in the running system (a prototypical mapping). Since the cardinalities of these two mappings, as well as their semantics, are different, we claim that UML 1.5 is ambiguous in relation to object diagrams. UML 2 (OMG, 2003c) removes the concept of Object as a model entity and introduces InstanceSpecification. The term “object” has driven some people to believe that the “objects” in UML 1.5 object diagrams are real objects (and the associated confusion related to objects being in the same layer together with their classes), when they are just a representation of objects. In this sense, the term “instance specification” is much better suited, since it clearly reflects that the model element is just a specification (i.e. an entity in a forward-looking model) of an instance in the SUS. However, the same ambiguity that we found in UML 1.5 remains, since the definition, description and semantics of InstanceSpecification in UML 2 (see OMG, 2003c, p. 62-64) allow for both isotypical (“An instance specification may specify the existence of an entity in a modeled system.”) and prototypical (“An instance specification may provide an illustration or example of a possible entity in a modeled system.”) mappings. Whether this open definition enhances expressiveness or hinders interpretation in practice is something that only time will tell.

2.2. Layering

In general, we can say that OMG standards use metatypical mappings between layers with only one exception: InstanceSpecification in UML 2 (or Object in UML 1.5) can exhibit isotypical and prototypical mappings to user models, as explained in the previous section. In all the remaining cases, OMG’s products not only use metatypical mappings but they choose a very special kind of metatypical mapping, namely, instance-of relationships. As we said in the previous section, a metatypical mapping is one in which the set of SUS entities that can be mapped to a given model entity is specified declaratively, i.e. the model entity is a description of the properties that SUS entities must comply with in order to be mapped to it. A type (in the UML sense) is definitely a declaration of its instances, so an instance-of relationship between a type and its instances can well be seen as a valid way to implement a metatypical mapping. However, other ways exist, and OMG’s line of products ignores them: a subtype-of relationship between a subtype and its supertype also implements a metatypical mapping, since the supertype can be seen as a specification of what properties SUS entities
must exhibit so they can be mapped to it. There is therefore no reason why software engineering modelling structures should not use it.

Figure 5. The four-layer OMG approach to model chaining. The top layer, MOF, is a self-model.

Furthermore, not only instance-of and subtype-of relationships are valid implementations of metatypical mappings; “exotic” relationships such as deep instantiation (Atkinson and Kühne, 2001) and powertype instantiation (Gonzalez-Perez and Henderson-Sellers, 2005b; Henderson-Sellers and Gonzalez-Perez, 2005) have been recently defined in the literature. The need for these new, “exotic” relationships is very clear: the concept of representation is transitive, meaning that a model of a model of a SUS is also a model of the same SUS. However, the instance-of relationship, the only one adopted by OMG, not only is not transitive, but is also representation-blind, i.e. it is not aware of the representation process. This means that an instance-of relationship allows us to describe the properties of an instance from the perspective of its type, but it cannot take into account further chained representation processes that may involve the specified SUS entity in a forward-looking model. For example, consider the class Class in (the latest draft of) UML 2 (OMG, 2003c). Class has a name attribute (inherited from NamedElement) as well as an isAbstract attribute. Together with the semantics of Class as described by the UML 2 specification, this is enough to metatypically map Class to any class in a user model (the SUS of UML 2, see Figure 5). However, from the perspective of UML, a class in the user model (such as Book) is just an instance-of Class, and therefore an object (since Class is a class). As an object, it will have a value for the name attribute (“Book” in our example) and a value for the isAbstract attribute (false in our example). This object cannot be the class Book (since it is an object and not a class, to start with), but is a representation of the class Book. However, the purpose of creating a Book class is to instantiate it into book objects in the running system, but since UML can only generate an object that represents the Book class, and not the Book class itself, this cannot be done. Of course, software engineers are well accustomed to do the mental
trick of automatically navigating the isotypical mapping between the object that represents the Book class, generated from UML, and the real Book class (its SUS entity, synthesised on the fly). However, and from a formal modelling perspective, this is only a trick and, in our view, far from acceptable in an engineering discipline. The reasons are twofold: first of all, UML (or any other approach that is limited to instance-of relationships as a means of implementing metatypical mappings) can only generate objects that represent classes, as explained above. The real classes are not derived from UML but from the objects derived from UML, by developers that use their subjective judgement to synthesise them as necessary. For example, when a developer creates the Book class by instantiating the Class class in UML, an object representing the Book class is created. The box on the paper labelled “Book”, strictly speaking, represents an object which can be isotypically mapped to a class that doesn’t yet exist. This class is created by the developer (or, even worse, separately by each developer that reads the diagram) when necessary. For example, when some code needs to be written from the diagram, a real class (for example, a C# class) is created. This class, as we have explained, is not an instance of UML’s Class but an isotypical partner of an instance of UML’s Class. In some scenarios (e.g. mapping classes to database schemata), model transformation technologies such as QVT (Query, View, Transformation) (OMG, 2002b) or MDA (Model-Driven Architecture) (OMG, 2003a) can help with this issue by providing a means of implementing the isotypical mapping between the specific class and object; however, this means that UML is not a self-contained modelling language since it needs assistance from external technologies to achieve formal completeness. The second reason is that modelling tools implemented as software systems (as most are) need to implement a fully formalised modelling infrastructure in order to support the necessary functionality. In our experience, modelling tool implementers know that using instance-of relationships only is not enough to achieve this, and additional workarounds must be added to traverse the “hidden” isotypical mapping that we have described.

2.3. Integrating Process and Product Metamodels

In the context of software development methodologies, metamodels are simply forward-looking models that represent any possible methodology that can be created. Like their model-oriented cousins (see first meaning of “metamodel”, above), methodology metamodels can be seen also as languages that can be used to express, in this case, methodologies. In any case, the content of a methodology metamodel depends on what the metamodel authors understand by “methodology”. We will adopt the definition given by (Gonzalez-Perez and Henderson-Sellers, 2005a): a methodology is a specification of the process to follow and the work products to be generated, plus consideration of the people and tools involved, during a software development effort. Since a methodology needs to consider both process and product aspects, and a methodology metamodel must be able to represent any possible relevant methodology, then process and product aspects must be integrated within the metamodel. The philosophy underpinning such integration is a linguistic simile: meaningful messages are built by applying actions to objects or, more specifically, complete sentences are constructed by combining verbs and nouns. Verbs specify the actions that are performed, while nouns denote the authors and receivers of the
said actions. Verbs without nouns can only specify actions, but an action needs a noun on which to be performed; similarly, nouns without verbs can only specify receivers or authors of actions, but not the actions themselves. Translating this into the methodology field, a software development process defines the actions to be performed when developing software, but these actions are meaningless without a detailed definition of the producers that execute the actions and the products that are involved. Similarly, products of software development (such as models or documents) alone are not enough; an indication of the process appropriate to create and employ them is necessary.

The Australian standard metamodel for software development methodologies, AS 4651 (SA, 2004), is intended to be used by a methodologist in creating a specific, tailored methodology. On the process side, AS 4651 offers classes that allow the methodologist to define activities, tasks, techniques and phases, among others. This falls within the same scope as OMG’s SPEM (Software Process Engineering Metamodel) (OMG, 2005). From the product side, AS 4651 offers classes that allow the methodologist to define models, languages and model unit kinds. This falls within the same scope as OMG’s MOF (Meta-Object Facility) (OMG, 2002a). AS 4651 defines associations between process-related classes and product-related classes. Consequently, the appropriate links can be defined between process and product elements in the methodology. AS 4651 is the only standard metamodel, as far as we know, to fully support product/process integration. The capability of AS 4651 to allow the methodologist to define both process and product aspects does not imply that both must be defined every time a methodology is needed; a method engineering approach (Brinkkemper, 1996) guarantees that a significant repository of pre-defined methodology components is maintained and re-used as necessary.

In order to achieve a similar degree of power, the OMG suite of products would need to be integrated at the most abstract level possible. MOF would seem to be an appropriate candidate for such as integration but, unfortunately, MOF mixes together two different concerns: on the one hand, MOF tries to define the infrastructure necessary to define modelling languages (such as UML); on the other hand, it tried to be an auto-model so it can fulfil its role at the top of the hierarchy (Seidewitz, 2003; Smith, 1999). Our claim here is that a true auto-model should be minimal since, if the auto-model is not minimal (i.e. contains entities that are not used by the auto-model itself), then these entities would not have any interpretive mappings to any SUS entities, which violates one of the premises in our theory (see Section 1). From this perspective, every class in MOF should have instances that are part of MOF in order for MOF to be a valid auto-model. Since MOF is a language specification, structural by nature, classes such as Operation and Exception cannot have instances that are part of MOF. Therefore, we must conclude that it contains more classes than it would need to define itself. Of course, this may well be a consequence of trying to offer the necessary infrastructure to represent modelling languages. A possible solution to this paradox would be to reduce the MOF to a minimal auto-model, perhaps along the lines of the CDIF meta-metamodel, and then create a methodology metamodel as an instance of MOF. Such a metamodel would justifiably focus on defining the language elements necessary to specify methodologies, addressing both process and product aspects. In this scenario, UML would be the product side of any methodology, while SPEM would comprise
the process side of the above mentioned metamodel (Figure 6). The recent Australian standard, while outside of the OMG suite of products, already offers such support.

Finally, we note that, currently, UML is a direct instance of MOF, but it should not be; this becomes clear if we realise that UML is one of many possible modelling languages, while SPEM is not one of many possible process specifications but a metamodel to build process specifications. UML and SPEM, therefore, are not at the same conceptual level, having different foci and level of abstraction, so integration cannot be performed between them simply in the way intended by the OMG.

3. Conclusions
This paper presents a technology-agnostic analysis of the concepts of model and modelling in software engineering. It has been shown that a model is always homomorphic with its subject under study (SUS), and that interpretive mappings (either isotypical, prototypical or metatypical) are required in order to trace model entities back to the SUS entities that they
represent. The difference between forward- and backward-looking models has been explained, and the issue of focus shift identified. The need of product and process integration in methodologies has also been addressed in the context of the OMG’s suite of models. Furthermore, UML has been determined to be ambiguous with relation to its Object or InstanceSpecification (depending on the UML version) class, and a “hidden” isotypical mapping has been identified that prevents software developers from deriving real class models from UML. Also, OMG’s process standard (SPEM) and modelling standard (UML) are identified as being incompatible as far as an integral process plus product methodology metamodel is concerned. The Australian Standard AS 4651, however, is considered valid and suggested as an alternative.

Acknowledgements
We wish to thank the Australian Research Council for providing funding towards this project. This is contribution number 05/09 of the Centre for Object Technology Applications and Research.

References