Unveiling Models

What's about Model-Driven Approaches

Daniel Perovich

29/08/06

1 Introduction

To model has been an intrinsic activity for men to conceptualize and reason about reality. Models have played an important role in several human activities, mainly in the context of science and also engineering. As well as other engineering disciplines, models have been incorporated in software engineering, and hence models have aided developers on abstracting, designing and documenting software systems. Modeling languages emerged in the 80's, increasing their acceptance and applicability in the 90's. A working software system is the most important artifact of a software development project; such an artifact can be considered as a model expressed in a particular language, namely the language understood by the technological platform intended to execute such model. Several intermediate artifacts, and particularly models, are built in the process of reaching the final working system. Which models must be built, who must do it, and when and how they are to be done, is specified by the software development process guiding the software development project. The diversity of possible processes, and hence kind of models, is enormous, and clearly not yet standardized. This reflects the fact that the software engineering discipline is still incipient, and consequently, it is very rich in improvements and of rapid evolution.

In such a context, the term of model is increasingly recognized as central. However, as other terms in the discipline, a veil of mysteriousness is still wrapping it up. We consider that there are three directions to tackled such concept, and thus, unveiling models. First, it is important to set up the intuition of the term. The concept of model has been revisited by several authors in the recent years. Even though there is no standard definition for it, several aspects are recurrent in those definitions. To revisit the definition in order to get the intuitive meaning of the term is one of the directions to explore. Second, the foundation that supports the term must be stated. Major efforts in this direction has been placed by means of a meta-circular approach based on the object-oriented paradigm; such approach is followed by the four-layer architecture proposal of the Object

Management Group (OMG) [8]. To build the foundation of something, i.e. to provide the semantics for a concept, implies to map this concept to a more basic and better-understood set of concepts. In this sense, the object-oriented paradigm is the target of OMG's approach. To provide a mathematical foundation for models is the second direction to undertake. Third, the concept must be studied from the perspective of its applicability. We have already mentioned that models are actually in use, but no standard procedures are defined to aid their applicability at each step of a software development project. Several authors have proposed different approaches and have clearly defined certain kind of models, but such effort is far from being complete. What is more, there is no clear procedure to characterize models and neither to contextualize software development techniques with respect to them. To aid such understanding in the third aspect to cover.

The goal of this work is to tackle these three directions. First, we revisit the definition of the term model. We propose several aspects from which this term should be analyzed, and as a result, we build a general definition for the term. Second, we develop the foundations for the concept of model and its related terms. Our approach differ from those found in the related literature as we build a mathematical foundations based on graphs. Remarkably, not only this foundation is aligned to the intuitive meaning of the concepts, but also it is easily mappable to OMG's four-layer architecture. Finally, we propose a multidimensional framework that accomplishes two purposes: to characterize models and to contextualize existing software development techniques. Most of the discussion may not result new to the reader as the topics we work on are somehow treated in the bibliography. Nevertheless, to the best of our knowledge, to organize the discussion in terms of a multidimensional space which thoroughly explores six aspects that characterize the term, is not done in the bibliography. The proposed framework helps to clearly define what kind of representation a model should look for, which kind of relationships with other models are possible, and how software development techniques are involved. Then, this work make the following contributions:

- identifies several aspects that provide the intuition on the meaning of the term model and its related terms, based on various definitions proposed by different authors, and proposed a definition that embraces all these aspects;
- builds a simple but accurate foundation which formalizes the terms of model, modeling language and model transformation, using graphs to express their semantics;
- propose a multidimensional framework to characterize the term model and its applicability, which in turn helps to contextualize software development techniques and provides a clear basis for understanding how model-driven approaches can/will evolve.

This work is organized as follows. To begin with, inspired on the definitions provided by several authors, Section §2 analyzes different aspects of the term model in order to set up the intuition on it. Later, Section §3 deepens this intuitive meaning. It proposes a rigorous definition for model, introduces the need for modeling languages, and defines the concept of model transformation based on those formal definitions. Afterwards, Section §4 presents the proposed multidimensional framework to categorize models and to aid the understanding on several software techniques such as software development processes and their disciplines, model-driven approaches, aspect-oriented approaches, etc. Finally, Section §5 concludes this work commenting on important remarks that can be derived from the study/application of this framework. It also suggests research lines that can be further developed.

2 Definition/Intuition

Unlike other terms in software engineering, the concept of *model* is broadly understood (at least intuitively) and accepted in the software community. Either due to a larger trajectory of usage in other disciplines or to the fact that subsequent authors have been refining the definition previously proposed by other authors, nowadays such definitions are being revisited with minor changes.

In this section we identify and analyze several aspects which are involved in the definition of the term model and its related terms. Our goal here is to set up the intuitive understanding of these terms. Afterwards, a definition for all of them is proposed so as to close this section. The aspects involved in the concept of model are the following.

What is being modeled. A *model* is a representation of another thing, being such thing a physical, abstract or hypothetical reality [3]. The model is not the actual thing, it just describes or represents those aspects of the reality that are important from a given perspective and fulfills a given objective. In [2] it is discussed the need for a particular term to name what is being modeled; we call *system* to what is being modeled by a model. As noted by the authors in [2], even though the term system generally refers to a software system in the software engineering discipline, this term is conceived in a broader sense and complies to whatever the target of a model is; e.g. for a business model the system is the business itself.

Rephrasing: a *model* is a representation of a *system*.

Considered a representation of a system. Then, the model is not the system itself, they are separate things. A model is a representation or description of (part of) the system. Whether the system may be physical or not, a model is an abstract thing. A system is generally complex and hence unmanageable directly. A model focuses on the set of characteristics of the system aimed to certain purpose, by omitting those that are irrelevant to it. As established in [10], a model captures the essential aspects of a system and ignores others; which ones are essential is a matter of judgement that depends on the purpose of the model.

Rephrasing: a *model* is a representation of (part of) a *system*, capturing the essential aspects of such system and ignoring others.

Level of meaning. Some authors use the term abstraction to characterize the relationship between a model and the system, i.e. a model is an abstraction of the system. Considering that to abstract means to pay special attention on something to the exclusion of all else, hence a model is an abstraction in the sense that it focuses on essential parts ignoring others considered as detail. From a different viewpoint, the abstraction provided by a model can lay at different levels of meaning, focusing on distinct concerns, and partially or completely describing the system; these are separate dimensions in which a model can be formulated. We explore this idea further in Section §4. However, it is important to keep in mind that a model represents a system at a given level of meaning or understanding, and so compelling a set of particular points on the mentioned dimensions.

Rephrasing: a *model* is a representation of (part of) a *system*, capturing the essential aspect of such system and ignoring others, at a given level of meaning.

Purpose and perspective. As studied in [10], models are used for several purposes: from capturing and thinking about the requirements, design and implementation of a system, to organize, master and explore alternatives of a complex system. The better understood the purpose, the most useful the model (for this given purpose).

In addition, a model is built from a particular perspective. Several stakeholders are involved in a software development project and different aspects of a system may be of interest for them. A model usually serves a given purpose of a particular (set of) stakeholder(s) or covering a particular (set of) concern(s).

Rephrasing: a *model* is a representation of (part of) a *system*, capturing the essential aspects of such system and ignoring others, at a given level of meaning, expressed from a particular perspective and serving an specific purpose.

Its constituent elements. A model is a composed artifact. It consists of a set of *model elements* structurally organized by means of different kind of relationships such as composition, aggregation, specialization, among others. In the Unified Modeling Language (UML) [9], as noted in [10], a model comprises a containment hierarchy of packages in which the top-level package corresponds to the entire system. We consider such a description to be too specific, and so prefer the constituent elements to be of any kind and related by any sort of relationship. A *model element* is a representation of a particular characteristic of the system, being the interconnected set of model elements the representation of the entire system.

Rephrasing: a *model* is composed by a structurally organized set of interconnected *model elements*, each representing a particular characteristic of the *system*.

A medium in which it is expressed. As we mentioned above, a model is an abstract and intangible thing, i.e. it is not physical or manipulable. In order to be used by more than one individual, a model must be somehow denoted so as it can be shared among individuals and be the subject of work for the development team (e.g. studied, updated, etc.).

A model is expressed in terms of a *modeling language*. A modeling language defines the set of constructs that can be used to express models. This set of constructs establishes (i.e. determines and hence limits) the kind of representation of a given *system* that can be made by means of the language. From another point of view, a particular aspect of a system can be represented by those modeling languages that provide the necessary constructs to fully represent this aspect at a specific level of meaning and precision. A modeling language determines both the syntax (either textual or graphical) and the semantics for each kind of construct. The clearer the syntax and the more unambiguous the semantics, the clearer and more unambiguous the model built by means of the language.

Counting with a well-defined modeling language enables the manipulation and storage of models, particularly by computer-assisted tools. In this context, a tool aimed to view and edit models is called a model viewers or editor, while a tool that persists a model is called model repository. Computer-assisted tools (environments) generally involve this kind of tools among others. Besides, and somehow more important, such tools permits to automatically manipulate models generating a given output (models) from a particular input (also models). Such tools, called *model transformations*, are the basis for model-driven approaches.

Rephrasing: a *model* is expressed in terms of a *modeling language* which defines (both the syntax and the semantics of) the set of constructs that determines and limits the kind of representation of a *system* that can be made by models built on this language. A *modeling language* makes a model tangible and operable, and hence, enables the development of computer-assisted tools for viewing, editing, storing and transforming models.

Definition

Summing up these aspects, we can define the term *model* in the following way:

> A *model* is a representation of (part of) a physical, abstract or hypothetical reality, called *system*, capturing its essential aspects and ignoring others, at a given level of meaning, expressed from a particular perspective and serving an specific purpose. A *model* is a composed artifact structurally organized by a set of interconnected elements, called *model elements*, each representing a specific characteristic of the *system*. As a *model* is an abstract thing, a *modeling language* is used to express a *model* and its constituent parts. A *modeling language* comprises the definition of the syntax and semantics of the set of constructs that determines and limits the kind of representation of a *system* that can be done by *models* expressed on the language. A *modeling language* makes a *model* tangible and operable, enabling the development of computer-assisted tools for viewing, editing, storing and transforming models.

3 Foundation

In the previous section we study the different aspects involved in the definition of the term model and we provide a concise definition of it. Even though that section encompasses the broadly-understood conception of what a model is, such definition, expressed in terms of natural language, can only help on aligning the readers' intuition of what a model comprises. In this section we encourage a formal definition of the term. In addition, we complement the formal foundation by also defining the related terms mentioned in the previous section.

In this section we first present a formal definition for model based on graphs. Then, we introduce the notion of modeling language which completes the definition of model. Later, we formally define model transformation. Finally, we conclude this section by commenting on the relationship among our formalization and the well-known specification practice by means of four-layer architecture proposed by the OMG.

3.1 Model

A *model* is particular kind of graph with a specific set of characteristics. Initially, let's consider such graph G as a pair (V_G, E_G), where V_G is a set of vertexes and E_G is a set of edges. Let's now analyze them separately in order to come up to a full definition of the term.

Each vertex $v \in V_G$ has identity, is typed, and has a labeled tuple associated to it. Vertexes have identity, that is, each vertex can be distinguished from any other, even though the associated information (their corresponding labeled tuple) coincide. The typing property of vertexes organizes the set V_G in several (possible overlapping) subsets; the type of the vertex characterizes its meaning. Finally, each vertex has a labeled tuple associated to it; such tuple holds data which is particular to the vertex. Hence, a graph G consists of a set of vertexes V_G, a typing function τ_{V_G} from the set of vertexes V_G to the set of vertex types T_{V_G}, and a function δ_{V_G} from the set V_G to the set of labeled tuples Δ_{V_G} that indicates the associated labeled tuple to each vertex.

The component E_G of a graph is a set of ordered pairs of elements of V_G, named edges. Each edge $e \in E_G$ has identity, i.e. it is possible to distinguish among two edges, even though they hold the very same data and corresponds to the very same pair of vertexes. Notice that this implies that E_G is a multi-set if its elements are simply ordered pairs, as the pair (v, u) can appear more than once. Moreover, E_G can have loops, i.e. there can be edges with the very same vertex in both component of the ordered pair. There exist two functions that for each edge obtains the first and second component of the pair, respectively. As vertexes, edges are typed and has a labeled tuple associated to it. Again, the type categorizes the edges and characterizes their meaning, and the labeled tuple hold the edge's data. Hence, a graph G consists of a set of edges E_G, a typing function τ_{E_G} from the set E_G to the set of edge types T_{E_G}, and a function δ_{E_G} from the set E_G to the set of labeled tuples Δ_{E_G} that indicates the associated labeled tuple to each edge.

It is possible to extend even further the notion of edge. Instead of considering binary edges (as they are ordered pairs), we can define edges as an n-ary tuple of vertexes from V_G; for example, this approach allow us to have ternary relationships among vertexes. Moreover, each component of the n-ary tuple can also be a set of vertexes $V_e \subseteq V_G$ instead of a simple vertex; this extension allow us to establish relationships between set of vertexes. An additional extension is to allow edges to be n-ary tuples of (sets of) elements in $V_G \times E_G$, allowing also to establish relationships among edges. By gaining in flexibility we loose in simplicity. Hence, we will keep the notion of edge simple as generally such complex structures can be simulated by additional vertexes and simple edges.

Then, a model M is defined by the tuple $(V_M, \tau_{V_M}, \delta_{V_M}, E_M, \tau_{E_M}, \delta_{E_M})$. Notice that the set of vertex types T_{V_M} and the set of edges types T_{E_M}, corresponding to the codomain of the functions τ_{V_M} and τ_{E_M} respectively, were explicitly excluded from the definition of a model M. Besides, the structure of the of the associated labeled tuples, i.e. the codomain of the functions δ_{V_M} and δ_{E_M}, namely Δ_{V_M} and Δ_{E_M} respectively, were also excluded. Thus, our definition of model is parameterized in these four aspects. By providing particular sets for each of these parameters we can have a model fully specified.

3.2 Modeling Language

The previous definition of *model* explicitly left out the typing and data definitions for vertexes and edges. From another point of view, the definition of such general aspects were factorized out from the set of models that share the same aspects. Thus, we can initially define a *modeling language* L as the tuple $(T_{L_V}, \Delta_{L_V}, T_{L_E}, \Delta_{L_E})$ of particular codomains required to fully define a model. By this means, a *modeling language* defines both the set of types (constructs) and the set of associated data (construct structure) that can be used to define particular models in the language.

Certain aspects still remain absent from the definition of *modeling language*. We have stated previously that vertex and edge types characterize them. On one hand, by characterize we can simple understand that types define groups in the universe of vertexes and edges. On the other hand, we can expect that types enforces specific restrictions on vertex and edges. This second approach allow us to place further requirements to how vertex and edges can be defined in models. Such requirements involve properties on the following: which type of vertexes can be connected by edges of a given type and which kind of information can be held by the associated tuples (i.e. the internal structure of the tuple) of vertex and edges of given types. These kind of restrictions are structural and further indicates how the constructs defined by the language (the types) can be instantiated and used in order to build a model.

Generally, another set of restrictions are required in order to fully specified the set of valid models. Such restrictions involve properties on the following: how many times can a vertex participate on edges of a given type, general properties on the values held by the associated tuples, and relationships among the presence or absence of edges under given conditions. Many of this kind of restrictions may have been resolved structurally, but generally, not only some of them cannot be expressed in this way, but also they complicates the structure of the models. Hence, we separate this set of restrictions from those that are purely structural. Then, we can state that a *modeling language* L is a tuple $(T_{L_V}, \Delta_{L_V}, T_{L_E}, \Delta_{L_E}, S_L, R_L)$, where the first four components are as before, S_L is the set of structural restrictions that apply on the first four components, and R_L is the set of additional restrictions that must also hold for these components. Both sets S_L and R_L has predicates as elements.

Finally, we can extend the notion of *modeling language* allowing a sub-typing relationship among the vertex types and edges types. By this means it is possible to incrementally specify the properties of vertexes and edges. Although possible and recommended, such extension is not further explored in this work.

Model as instances. Given that a *modeling language* determines the set of constructs (types) and relationships that can be used by *models* expressed on the language, it is said that a *model* is an instance or belongs to the *modeling language*, fact denoted by $M \in L$. To state that $M \in L$ is to state that vertexes and edges of M has types (and tuple types) defined by L and that every predicate in S_L and R_L holds for M. We say that a model M is well-structured when every predicate in S_L holds, and we say that M is well-defined when predicates in either S_L or R_L hold.

Language for languages. A *modeling language* can be seen as a *model* itself where vertexes corresponds to the types and edges corresponds to the set of structural restrictions of the language. Hence, there is a (meta-) language $\mathcal{L}$ that can be used for defining *modeling languages*.

Basis. The basis behind the foundation for models and modeling languages presented so far is inspired in well-known foundations for (programming) language processing tools such as parsers and compilers. In such tools, a textual program (just a sequence of characters) expressed in terms of a concrete syntax is parsed in order to build a parse tree. As parse trees include more detail than needed (due to having been build from the input text), an abstract syntax tree (AST) is later generated from the parse tree. An AST is a finite labeled directed tree conforming an abstract representation of the original text. An AST contains only those vertexes and edges that are obtained from the syntax rules that directly affect the semantics of the language, omitting the others. All possible ASTs for a language are determined by its abstract syntax. A subsequent step is to build an abstract semantic graph (ASG). An ASG is at a higher level of abstraction of an AST [11]. It is built from an AST by a process of enrichment an abstraction. In the former, the AST is extended by appending edges from those vertexes that references other vertexes. In the latter, some detailed constructs present in the AST are removed.

Thus, the way models were defined can be seen as a particular kind of ASG. Moreover, modeling languages corresponds to the concept of abstract syntax, extended to ASGs. Notice that we consider a model definition directly as an ASG, not as any textual or graphical representation which can be later transformed into this ASG.

3.3 Relationship to the meta-modeling approach

Besides the intuition for the proposed foundation in terms of ASGs, it is important to notice also how close is our foundation with respect to the four-layer architecture for model specifications suggested by the OMG. The four-layer architecture involves at the bottom-most level, called M0, the instances of models. The following level, namely M1, consists of the model of the system. M2 consists on meta-data, it is a model for defining models; models in M2 are called meta-models. Finally, the top-most level is called M3, and consists of the basic minimal infrastructure to define meta-models [3]. The OMG proposes several standard meta-models at M2, namely the Unified Modeling Language (UML) and the Common Warehouse MetaModel (CWM) [5], while basing the top-most level by a single, meta-circularly specified standard, Meta Object Facility (MOF) [6].

Our foundation can be easily mapped into such four-layer architecture. *Models* are equivalent to models inhabiting the M1 layer while the *modeling languages* presented inhabits M2, as they provide the constructs for building models. Finally, conceiving the single language $\mathcal{L}$ for defining modeling languages is analogous to having a single top-most meta-meta-model.

Another subtle but important remark is that OMG's models are mainly object-oriented. The basic essential component of MOF is the *Class* construct containing a set of properties (in which elements of other classes can be attached), is much more similar to our models or even modeling languages organized as vertexes holding data and edges relating them (both aspects held by properties in MOF). Hence, it is notable that there is a semantical equivalence among our proposal and the widely-accepted OMG's one.

3.4 Model Transformation

The previous section formally defines the concepts of model and modeling language. As we mentioned in Section §2, having a formal representation of models aids the development of tools for manipulating and storing models; such tools can be logical mechanisms or computer programs. Based on the foundation we have already built, we can logically define a model transformation. A computer representation of this foundation would enable such transformations to be programmed and executed. We define a *model transformation* as a function which takes a model as input and produces a model as its output. Such function can really be of any arity and hence, a tuple of models may be its input and a tuple of models may conform its output.

Let's start by defining unary transformations. An unary transformation is a function which transforms a model into a model. Let M be the input model, and let L be the modeling language in which M was build, i.e. $M \in L$. Let L' be the language in which the target model is to be expressed. Hence, a unary model transformation T (that generates only one model) is a function from L to L', denoted by $T : L \to L'$. In other words, T is a member of the set of functions $\{L \to L'\}$.

Recall that a modeling language L states two sets of restrictions, namely S_L and R_L. The transformation T is defined only on those models that are well-defined, i.e. that satisfy both sets of restrictions. Given a particular language L, we can derive the more general language $\widehat{L}$ which is equal to L except that the set of additional restrictions is empty, i.e. $R_{\widehat{L}} = \emptyset$. A well-structured models in $\widehat{L}$ is both well-defined in $\widehat{L}$ and also well-structured in L but may not be well-defined in L. Then, we can define a particular kind of transformation $\widehat{T_L} : \widehat{L} \to \mathcal{B}$ which takes those models in $\widehat{L}$ that are also models in L to `true` and takes all others to `false`; notice that $\mathcal{B}$ is a model for boolean values, i.e. it consists on two (unconnected) vertexes, namely `true` and `false`. In other words, the transformation $\widehat{T_L}$ can be used to state whether a model M satisfies or not the set of additional restrictions R_L of the modeling language L.

Two other specific kind of unary transformations are those that build a model M

from an input string (parser transformation), and those that produce an output string representing the model (pretty-printing transformations). Both kinds of transformations take, and generate respectively, a model expressed in terms of a string of characters. Such particular language is generally called Σ^* and can be represented by a type of vertex for each element in Σ (which resembles an alphabet), and a single type of edge denoting precedence among them.

Binary transformations take two input models M and M' and generate another one. Such transformation T is a function from $L \times L'$ into L'', being these languages the corresponding ones for the models. A binary transformation merges or combines what is being modeled in the input models generating a third one consisting in such combination. This kind of transformations need specific information about the correspondence among elements on the models. Such additional information can be placed in either of the two models (i.e. one model states to which element in the other model corresponds; this can be done by means of names), on the transformation itself (i.e. the transformation determines which elements are corresponding ones), or in a separate model. This latter approach is much more flexible as this third model parameterized the transformation.

There is a specific kind of binary transformations in which the transformations were originally thought of as unary but additional information is required to generate the target model. Approaches as Model-Driven Architecture (MDA) [7], propose to mark the source model prior to apply the transformation. Such marks provide the additional information required to generate the desired target model. Instead of following this approach, a separate model can be considered where those marks are independently expressed. In such scenario, the transformation is binary as it takes the original model M and an additional model representing the required marks for M. Again, this approach provides greater flexibility.

Transformations of greater arity are also possible; we have already considered a ternary one above. The Model-Driven Development approach is based on this general idea of transformations taking several input models and generating several output ones. The need to output more than one model can be seen as outputting an element on the cartesian product of the languages of each of the models. Anyways, such transformations can be split in separate ones, each generating one of these models.

As transformations are functions, they can be composed. By this means, having a transformation $T : L \to L'$ and another transformation $T' : L' \to L''$, the composed transformation $T' \circ T : L \to L''$ is defined for every M in L as $(T' \circ T)(M) = T'(T(M))$. Besides, the inverse transformation T^{-1} can be defined as $T^{-1}(M') = M$ if and only if $T(M) = M'$. It must be notice, however, that even though we can logically define such transformation, there is not always the possibility or a direct procedure to obtain the transformation T^{-1} given a transformation T. A transformation T for which T^{-1} is known (i.e. it can be built) is called bidirectional.

3.5 Transformation language

A model transformation $T : L \rightarrow L'$ takes a model $M \in L$ to generate a model $M' \in L'$. Such transformation must be somehow expressed in order to unambiguously define what the transformation is about. To this end, there must be a specific language that provides the set of constructs required to express any kind of transformations. Such language can be declarative or imperative. The former consists on a set of rules that must hold on the source and target model. The latter consists of a set of instructions that establishes what must be created in the target model by querying the source model. Both approaches, however, must have the ability to query a model, and the latter must also be able to instantiate one.

There are distinct approaches for specifying transformations. One first approach is to use a particular language $L_{T_{LL'}}$ for the transformation which permits to query instances of the language L and to query and create instances of the language L'. Such language provides special constructs for manipulating models expressed exclusively in these particular languages. A second approach is to define a general language L^T which provides constructs for manipulating (querying and instantiating) any language. Such transformation language must be based in terms of the meta-modeling language presented above, namely $\mathcal{L}$. In either case, a transformation can also be seen as a model, as it is an element specified by a given language. Thus, transformations are also subject of transformations.

4 Models' Dimensions

The previous sections tackled the definition of key concepts related to model driven approaches, namely models, modeling languages and model transformations, both intuitively and formally. Several aspects of models were discussed in Section §2, including the notion of system and the fact that a model serves a given purpose, is elaborated from a given perspective and at a given level of abstraction. This intuition, however, was not formalized in Section §3. In this section we provide a framework aimed to contextualize these aspects of models.

Models play an important role in software development projects. They are elaborated during most phases of the process, serving as documentation, deliverables, and as input and output of several activities. A model repository (see §2) is nowadays even more helpful than before as several models are built and used during the development effort, each tackling different aspects of the software system. Moreover, to state and preserve the relationships among all these models is becoming even more important. First, it helps developers to trace model elements, to understand where they come from or where they have influence. Second, these relationships are indispensable to computer-assisted tools in order to manipulate models accurately.

The way to correctly characterize and catalogue models is partially cleared.

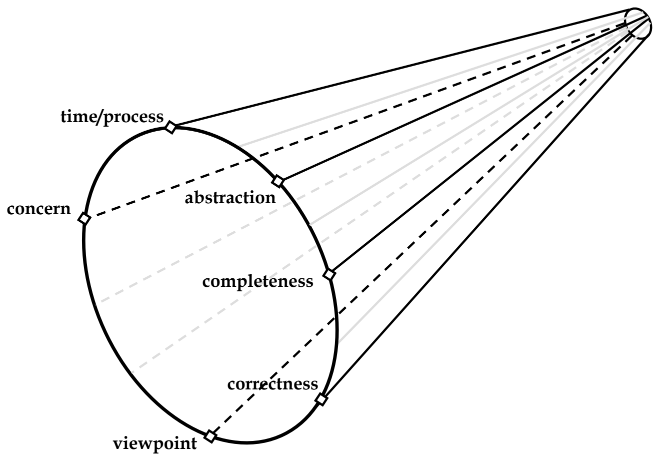

Figure 1: Sketch of the multidimensional framework

Models are part of UML, which provides some variety of dependencies that allow certain degree of tracking of model elements. In this section we deepen the characterization of models. We explore several dimensions in which models take part. Based on these dimensions, development processes, activities, and artifacts, can be clearly located and understood.

The study of this framework is done incrementally, beginning with the *time/process* dimension and incorporating all the others one by one. Together with introducing dimensions, software development techniques are commented and discussed to the light of the framework being built. The multidimensional framework proposed focuses on six major aspects of models and modeling. Even though several additional dimensions may be considered, the coverage of those presented here is enough to identify most major software techniques, including model-driven approaches. Figure 1 sketches the multidimensional framework elaborated in this section[1]. The main dimensions are named (represented by solid lines in the figure) while other dimensions (shaded lines) are present but not considered in what follows.

4.1 Time/Process

The first dimension in which models must be considered is the *time* or *process* dimension, as every software development project generally follows a software development process and spans on certain period of time. This dimension can be continuous if considering it as a time line, or it can be discrete by considering

[1]The reader should be aware of the difficulty of drawing in a plane an element of six dimensions. Although presented as a cone, the far-most circle resembles the origin while the solid lines resembles the dimensions, which must be thought as orthogonal to each other.

an specific set of points in time which are of interest for the project.

Dimension conformation and localizing a model. In the first scenario, the origin of the dimension is the project initial point in time, and the dimension span to the end of the project. Moreover, it can span infinitely so as to cover any future evolution of the system. A model would cover a (possible infinite) time span, beginning at the point in time the model was created, and lasting till the end of the project (or indefinitely). In this scenario, even though a model may exists till the end of the project, it may be really useful for a shorter period of time. Consider a model M_i that was updated during an activity in order to generate a newer version of it. We face two possibilities. On the one hand, we can consider the new version M_{i+1} as the very same model as M_i, preserving the former overwriting the latter. On the other hand, we can consider the new version M_{i+1} a separate model from M_i; the model M_{i+1} is considered as being created while M_i remain untouched. Then both models are preserved. The first case is preferable to small changes in the model while the second case is preferable to large changes. In this case, we must state that M_{i+1} is somehow related to M_i. Then, the model M_i does not necessarily span till the end of the project, instead it may span till a new version is available.

The second scenario consists on defining a finite set of points in time which are of interest to the project. Most projects define an execution plan in which major milestones are stated. In addition, the activities involved in a development project work on input artifacts (particularly models) producing new ones or updating them. As projects generally involve configuration managements activities, artifacts are versioned and preserved. Thus, we can choose the set of expected versions, which in turn include versions for milestones, as the set of points in time to be considered in this dimension. In this case, a model simply inhabits a single point in time, particularly the point corresponding to its versioning information. Notice that this approach is more compact than the previous scenario but provides the same information.

Interesting points. It is important to notice that several models within different points in time (versions) can be considered as deliverables and hence be mandatory. Additionally, and yet more important, the final version of the models are those of major interest, particularly the model corresponding to the final working software system; such models are located at the far-most point from the origin of the dimension.

Model relationship. As we consider a model M_i and its subsequent version M_{i+1} as separate models, it is important to provide a mechanism to relate models within this dimension. We call such relationship *trace*, and hence, we can state that M_{i+1} *traces* M_i. The name for the relationship through this dimension is taken from UML. In UML, a trace dependency is a particular kind of an abstraction dependency that indicates a historical development process or other extra-model relationship between two elements that represents the same concept without specific rules for deriving one from the other; such dependency is used as a remainder to the developers during development [10]. Unfortunately, as also stated in [10], a trace dependency can be additionally used to relate model elements at different planes of meaning, such as tracing a design construct in one model to a corresponding requirement construct in another model. It is important to notice that these different levels of meaning do not necessarily imply different points in time, and neither that a lower level of meaning need to be posterior to a higher level of meaning. We propose to limit the *trace* relationship exclusively to the *time/process* dimension. The other intent of UML's trace dependency is covered later when we consider the *abstraction* dimension of the framework.

Remarks. As a final remark, to reduce the time to market can be understood as making this dimension the shortest possible, i.e. making the largest point the closest to the origin of the dimension.

4.2 Viewpoint/Perspective

The *viewpoint/perspective* dimension is a finite discrete dimension that considers the set of all viewpoints or perspectives involved in a software development project.

Dimension conformation. All stakeholders, or more accurate all categories of stakeholders, are represented by a point in this dimension. A stakeholder is an individual with a particular interest in the software development project being carried and/or in the software system being developed. Generally, it is possible to separate two distinct aspects for stakeholders: roles and workers. A role is the definition of a profile of stakeholders. It defines a set of activities in which is needed to participate, a set of artifacts needed to work on, and a set of artifact of him/her direct responsibility. On the other hand, a worker is an individual involved in the software development project. A worker plays one or more roles during the project, and conversely, a role can be played by one or more workers. To the light of this distinction, we find more accurate to define the points of the *viewpoint/perspective* dimension in terms of roles instead of workers. Notice that different workers playing the same role should be interested in the same artifacts (particularly models) and would have the same perspective of the system; a role specifies such perspective.

Localizing a model. A model M can be in one or more points of this dimension, meaning that the particular model is of interest for all those roles marked as points on this dimension. A model rarely covers all points in this dimension. The most broadly targeted models can be the project plan and the software architecture description, artifacts aimed to assist the needs of several stakeholders. Apart from them, most models would cover only a few points in this dimension. It is important to remark, however, that most roles are interested in reaching a working software system, and hence, such final model would inhabit all points in the dimension. It must be noticed that there may exist some roles, such as architecture reviewer or domain expert, that may not be interested in the final system.

Model relationship. We do not identify any important relationship among models through this dimension. Additionally, as far as we know the bibliography does not propose any neither.

Relation with other dimensions. There is no special relationship between this dimension and others dimensions. This dimension is mainly administrative and its goal is to identify (and define) the expected perspective of a model. Generally, new versions of models are of interest of the same category of stakeholders, but this is not necessarily the case. The project manager together with the system architect should collaboratively decide which models are to be built during the project and to whom these models are targeted.

Remarks. As a final remark, it is important to see that this dimension is very important from the management perspective of a software development project, but provides few information to software developers themselves. Developers are more interested in the other dimensions we propose.

4.3 Abstraction

The *abstraction* dimension is also a finite discrete dimension considering all different levels of meaning, understanding or abstraction. It might be considered continuous, but at the price of incorporating more complexity and loosing manageability. Generally, as we present later, a handful of levels of abstraction are enough for a software project. The *abstraction* dimension contains at the origin the bottom-most level of abstraction, meaning by bottom-most level that of the computer infrastructure. At the opposite side of the dimension is the top-most level corresponding to the human abstract understanding of the system. The upper the level, the more understandable for most humans, the lower the level, the more computer-related and technical.

Dimension conformation. This dimension is generally inhabited by the principal development disciplines involved in any software development process: requirements, analysis, design, and implementation. Even though a software development process does not explicitly make such a distinction, these levels of abstraction are implicitly present in the developers conception of the system.

Certainly, we can further separate the levels of abstraction. For example, requirements can be gathered at the business level (higher) or at the system level (lower). Also, the implementation can be subdivided in source code, executable units (those directly deployed), the underlaying platform. Which particular levels are to be used (conceived) in a project strongly depends on the software development process followed and the peculiarities of the system being built. However, it is important to notice that the levels we have already mentioned are widely accepted and used. See for example that these are different disciplines proposed by the Rational Unified Process (RUP) [1]. Additionally, the models proposed by the Model-Driven Architecture (MDA) [7, 4] approach, can easily be mapped to these few levels. The Computation Independent Model (CIM), the Platform Independent Model (PIM), the Platform Specific Model (PSM), and Implementation Specific Model (ISM), directly corresponds to requirements, analysis, design and implementation, respectively.

Localizing a model. Even though we can develop a model whose elements are at a different level of abstraction or meaning, this approach is clearly inconvenient and is strongly unrecommended throughout the bibliography. As we define in Section §2, a model must be a representation of the system as a particular level of meaning, focusing on the essential aspects and ignoring the others. Hence, a model at a given level of abstraction should not include elements conveying neither a higher nor lower level of abstraction. Then, a particular model M is located at exactly one point in this dimension; having a model M inhabiting more than one point is generally due to bad practice.

Interesting points. The final working software system clearly inhabits the lowest level of abstraction as this level is the one a computer can directly execute the system. However, models at such lowest level are improbably directly produced by developers. Instead, models are developed at a higher level than this and several tools are used to convert them into those which are executable by the computer. Generally, the lowest level used by developers is the source code. The implementation at this level is then targeted to a platform to be executed. The basic scenario for this transformation is by means of a compiler which generates the binary sequence understood by the underlying platform. More complex scenarios can be used and are now gaining acceptance. Compilers generates intermediate representations (i.e. models at lower levels of abstraction) which are then executed by a virtual machine, further compiled by just-in-time compilers, etc.; such is the case of Java and the .NET platform.

There are specific kind of models that can be place next to the underlying platform, even lower than source code. Deployment scripts or configuration files are particularly targeted to the platform and specifies the parameters for a smooth execution of the system in such platform. Clearly, such model concretize a deployment model at a higher level of abstraction, if present.

The final working system, located near to the computer level of abstraction, is seldom the unique model developed during a software development project. Several models are built, at higher level of abstractions, which provide improved understanding, communication and reasoning. The higher the level, the best for human understanding, and thus, models at the highest levels are preferable. Consequently, there is a (enormous) gap between the computer expectations of a model and our expectations; such gap is not new, software development processes devote enormous effort to propose how to traverse it.

Model relationship. We can consider a model M' as a more concrete version of a model M, or conversely, a model M as a more abstract version of a model M'. Model M' is at a lower level of abstraction than M, even though if follows a similar purpose and represents the system from the same perspective as M. Uniquely, these models are different in the sense that M' incorporates additional characteristics that approximates it to the underlying platform and may omit, also, some information from M which is more related to human understanding. Then, we define the *concretizes* relationship among models, and the converse relationship *abstracts*, to state such relationship among models. Thus, we can state that M' *concretizes* M, and in other words, that M *abstracts* M'.

UML proposes a dependency kind called abstraction for this very same purpose. However, this dependency targets a broader kind of relationship than ours. Quoting [10], an abstraction dependency is a relationship between two elements at different abstraction levels, at different levels of precision, at different levels of concreteness, or at a different level of optimization. We consider that such relationship can be categorized in different independent kind of relationships. Hence, we consider our *abstracts* relationship as the one in UML under the meaning of different abstraction level or different concreteness levels (a distinction among this two aspects is unclear and is not further commented in the bibliography). Notice that the precision aspect is being tackled later by the *completeness* dimension.

Additionally, UML proposes an specific variation of the abstraction dependency called refinement. This relationship represents a fuller specification of something that has already been specified at a certain level of detail or at a different semantic level [10]. It remains unclear if a different semantic level means a different level of meaning or abstraction, but it can be guessed that it does. Again, such a relationship seems to tackled several aspects of how two elements (and particularly models) can be related; on the one hand detail (precision) and on the other hand semantic (abstraction) level. We propose to use the *abstracts* relationship for relating models through this dimension, and prefer to use the *refines* relationship to state the relationship among models through the *completeness* dimension; as we mentioned above, such dimension is concerned on detail and precision.

Relation with other dimensions. Now we have already introduce various dimensions, it is important to study the connection among them. We have

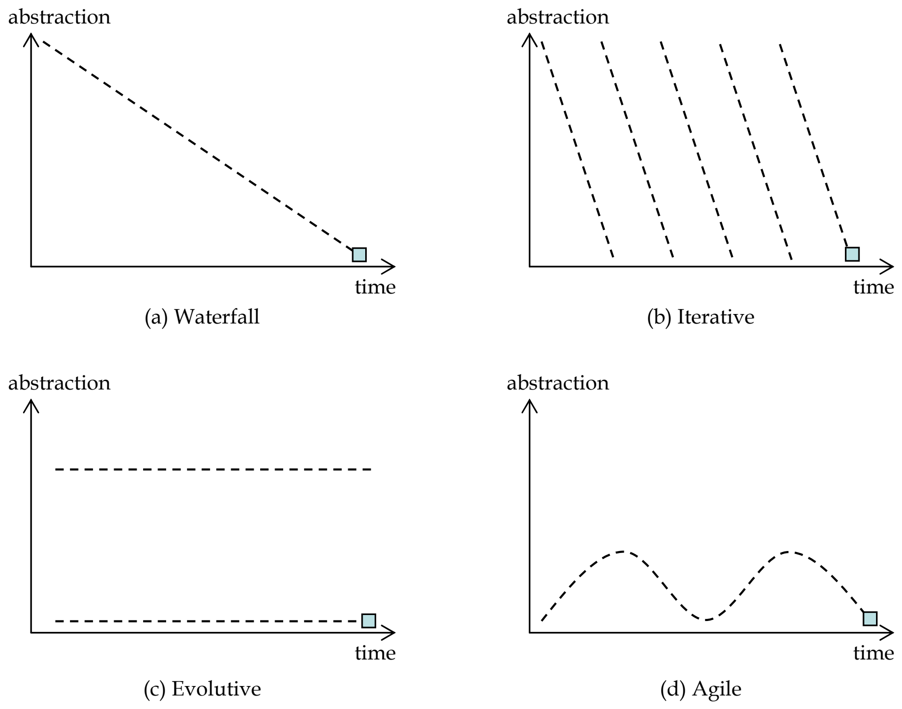

Figure 2: Relation among the *abstraction* and *time/process* dimensions.

not done this before as we state that the *viewpoint/perspective* is independent of other dimensions. However, by relating the *time/process* and *abstraction* dimensions we can provide a better understanding of the usefulness of this multidimensional framework. Using this two dimensions we can graphically show how different software development process models behave in terms of artifact elaborations towards a final working software system. In Figure 2 we depict four of such process models.

Figure 2a presents the waterfall process model which organizes a process in stages to tackle requirement gathering, analysis, design, implementation and testing, where one begins when the previous one is totally completed. Graphically, every non-increasing function can be used to represent which artifacts are created at which point in time. Notice that we use a continuous line in order to make clearer the exposition, however, recall that we consider both dimensions as discrete. The figure shows that the more abstract models are built at the beginning, and that models at a lower level of abstraction are built afterwards. Notice also that the final working system (depicted as a small square) is reached at the end of the project. A models at a low level of abstraction *concretizes* a model in a higher level. In the case of a linear curve (as in the figure), no *traces* relationship exists. However, although we show a linear curve, any non-increasing function is possible, as several versions of models at the same level of abstraction can be built. In such case, *traces* relationships are also present.

In Figure 2b an iterative process is shown. An iterative process is organized in iterations, each of which proceed on every activity performed in a waterfall process model, but tackling only a small part of the whole system. Each iteration generally produces an increment obtaining finally the working software system. In the figure, we show a project consisting only of five iterations reaching the final working product at the end of the fifth one. A model M_{i+1} located at a given level of abstraction at the $(i+1)$-th iteration *traces* a model M_i at the very same level of abstraction but in the i-th iteration; such relationship takes place horizontally. Also, the model M_i *concretizes* an upper model in the *abstraction* dimension in the same point in time; such relationship takes place vertically. Again, although we use a linear curve, any non-increasing function apply to the depiction of an iteration, and different curves are possible for the iterations.

An evolutive process model is shown in Figure 2c. This kind of process involves to parallel carry out specification, development and validation of the process. The three activities are performed altogether producing new versions of the artifacts till reaching the final working system. The figure show models at two different levels of abstraction, the upper level corresponds to an specification model while the lower level to an implementation model. At both levels, subsequent versions of these two models are produced, obtaining the final system as the last version of the lowest level. This example also show that the relationship between the two dimensions is not necessarily a function, it is a relation. However, in processes where no parallel development exists, such relation is properly a function. Notice that this is not necessarily the case for the two previous cases, at least the iterative process model as it can include parallel development, even though we have not depicted this case.

Finally, Figure 2d sketches an agile process model which are mainly devoted to implementation, using round-trip engineering among implementation and design. Models at both levels of abstraction are used, one used to derive a newer version of the other, either automatically or manually. Even though we present it as a sinusoidal curve, it is generally flatter than this, consisting of several versions of models at the implementation level and some picks to the design level at certain points in time.

Remarks. It is important to notice that testing activities are not directly expressed in the *abstraction* dimension. Its impact indirectly influence on the combination of this dimension with the *time/process* one, generating a new version M_{i+1} of a model M_i at the same level of abstraction. We are not able yet to state when a model is more accurate than other; we can only guess it by considering the fact that certain time has passed. Nonetheless, it is possible to have a newer version of a model but without any improvement in terms of correctness. This particular issue is covered by the *correctness* dimension discussed next.

As a final remark, we can say that model-driven approaches are targeting the construction of mechanisms and tools to aid the conversion of models at the highest levels into models at the lowest levels automatically, or at least computer-assisted.

4.4 Correctness

The *correctness* dimension consists of the level (percentage) of correctness of models. Even though the dimension can be both continuous or discrete; considering the percentage of correctness is generally easier to maintain and also to characterize models. The fact of being correct is particularly difficult to establish for a model. A definition of correctness must be stated and procedures to measure it are required, preferably quantitative procedures to qualitative ones. However, this is not an easy task.

The testing activity directly tackles the improvement of the accuracy on models. Several testing techniques have been designed, most of them tackling models at the lowest levels of abstraction. In spite of this, revision techniques are being applied to most kind of models, and hence the testing phase is more able to improve the correctness of models

Within this dimension, it is important to consider the fact that nothing is correct by itself, or, from another perspective, everything is correct by itself. In other words, a sentence like `x := x + 1;` can be correct or incorrect, depending of the expected behavior for the sentence. Additionally, to represent a domain concept as a class or as an association can be correct or incorrect, depending mainly of the domain expert knowledge and on the semantics of the domain concept. Then, the correctness of a particular model element, and consequently its containing model, is relative to the expected intent for the element. So, to state how correct is a model is to establish how well it satisfies the expected intention. The expected intention of a model should be clearly stated, generally independently of the model itself, and to check the correctness of the model involves to verify that the model satisfies its specification. It is important to notice that to build a model specification is also to build a model.

An important problem arise when considering this dimension. Let's consider that we can catalogue models as specification or realization models[2]. Having specification models, we can verify that a realization model is correct if somehow it does realize its corresponding specification model. The problem is, however, how to determine whether an specification model is correct or not. For models at a low level of abstraction, its specification can be elaborated in terms of the specification models at higher levels. However, it is not possible to (systematically) determine whether the specification models at high levels of

[2]Such distinction may be treated as a separate dimension of the framework.

abstractions are correct or not. To be correct would mean to match all stakeholder understanding of the system, a particularly difficult task to concretize in an specification (model). It is important to notice that the problem we identify here is the fact that states the distinction among verification and validation activities in the testing discipline. To verify means to check whether the model satisfies its specification while to validate means to check whether it satisfies the stakeholders intent.

Dimension conformation. We propose to conform this dimension in percentage of correctness, or any other discrete scale defined for the particular project. As a matter of convenience, we match the origin of the dimension to the most correct while the far-most point to the less correct.

Localizing a model. A model can be placed at any point of this dimension, but exactly at one of them. The software development process must define several metrics to quantitatively determine the level of correctness of a model, and hence, to uniquely locate it at the corresponding point in this dimension.

At the light of the previous discussion, specification models may be difficult to determine their correctness, mainly those corresponding to the higher levels of abstraction. The intuition of the responsible of those artifacts can be an approximative metric.

Interesting points. The most interesting point in this dimension is the origin, corresponding to flawless models. However, testing techniques which do not apply formal methods for verification can seldom reach this level. Then, the best models are those which are nearest to the origin, being the final working system (hopefully) one of them.

Model relationship. We can define a relationship to relate models through this dimension. Let's consider a model M_i to be the input of the testing activity and as an output we determine a set of flaws in it. The change management discipline together with the responsible roles of M_i would determine the required changes and then perform them. An improved new version M_{i+1} of M_i would be created which corrects (most of) the detected flaws. Then, we can state that M_{i+1} *corrects* M_i as the former one is more correct than the latter.

It is important to remark that a flaw in a model is due to one of two different causes: an error in the model or an error in the specification. In other words, a model can fail to satisfy its specification because it does not properly realizes its specification or because although it does realize its specification, such specification is not correctly expressed in the already built specification. Suppose we have model M_i and its specification model S_i. When we detect a flaw in M_i then either M_i or S_i are incorrect. To correct a model means either to update it or to update its specification. The first case is the one detailed above. The latter case would produce a new version S_{i+1} that *corrects* S_i. Notice that the same relationship is used for both scenarios.

Relation with other dimensions. As we mentioned above, models at any level of abstraction can be incorrect and hence the target of verification or validation. These activities produce change requests which are then performed to build new versions of the models. Hence, once a model M_i is generated, the testing discipline determines at which level of correctness is M_i, while performing the required changes (if exist), a new version M_{i+1} is built and tested again (by means of regression tests). Then, there is no particular relationship among this dimension and the *abstraction* dimension, a model at any level can be erroneous and so subject of change.

It is interesting, however, to remark the relationship among the *correctness* and the *time/process* dimensions. Once the changes outputted by a testing activity are performed a new version of an existing model is developed which corrects the previous one. Hence, after testing and performing changes, a model M_i is turned into a model M_{i+1} which is closer to the origin than M_i in this dimension. In addition, M_{i+1} also *traces* M_i and hence is a bit farther from the origin with respect to the *time/process* dimension.

4.5 Completeness

The *completeness* dimension consists of the level (percentage) of completeness or coverage of models. This dimension can also be continuous or discrete, and again, the latter case makes it more manageable and understandable. Similar to the *correctness* dimension, it is difficult to establish how complete is the representation provided by a model with respect to the model's purpose and perspective. We can consider the completeness of a model M in terms of the percentage of its specification that is covered or solved. For specification models, however, this procedure is not possible. A specification model can be measured in terms of completeness looking at how much detail of the represented system has been considered in the model. For example, a domain model delineating just concepts and their associations is less detailed than a domain model which mentions the attributes of the concepts as well. Moreover, a domain model that indicates the type of the attributes and also state constraints on the overall structure is far more detailed.

It is important to notice that to be more detailed does not imply to be more concrete in terms of the *abstraction* dimension; the example presented above makes clear such distinction. We discuss this issue later.

Dimension conformation. We propose to conform this dimension in percentage of completeness, or, as before, any other discrete scale defined for the particular project. Again, as a matter of convenience, models near the origin are more complete while those far from the origin are less complete.

Localizing a model. A model can be placed at any point of this dimension, but exactly in one of them. It is difficult however to define uniform metrics to check how much complete is a model. Such metrics should depend on the kind of model, conceiving by kind of model the level of abstraction and the concern(s) of the model (concerns are discussed latter).

There are no standard procedures to determine the level of completeness of a model, mainly for specification models. It is the responsibility of the software development process to indicate which kind of models should be built, and at which level of completeness they should be defined, depending on the phase of the project. As we notice later, iterative & incremental process models focus iterations on producing an increment (i.e. on gaining completeness) of the models built so far.

Interesting points. The most interesting point in this dimension is the origin. Models at this point are complete and hence, totally fulfil their purpose and cover all aspects from their perspective, at the particular level of abstraction. Models at this level are precise. Conversely, the farthest we get from the origin, the fewest precision or the most incomplete the model.

Model relationship. A we mentioned above, UML defines the refinement dependency to state that an element represents a fuller specification of other element that has already been specified at a certain level of detail or at a different semantic level.

The *abstracts* relationship was defined above for relating models among different levels of abstraction (i.e. different semantic level). We propose the *refines* relationship to state the model relationship in which a model specifies more detail than other, i.e. that is more complete. Thus, given a model M_i at a certain percentage of completeness or precision, and given a model M_{i+1} consisting of a more detailed version of M_i, we can say that M_{i+1} *refines* M_i.

It is important to notice that in our framework, being more concrete (i.e. less abstract) means to be expressed at a level of abstraction that is more close to the computer. However, to be more precise means to be expressed with further details but at the same level of abstraction. Of course, a model at a lower level of abstraction can include more detail that a model at a higher level of abstraction. Suppose we have an analysis model A_i in which we partially specify the functionality of the system, and then we build (derive) a design model D_i which solves the same functionality but also covers exceptional cases not considered by the analysis model. First, the design model D_i includes certain platform aspects which have been abstracted away by A_i when solving the same functionality; this is the main difference among these two levels of abstraction. Hence, we can state that A_i *abstracts* D_i, or equivalently, that D_i *concretizes* A_i. Second, as D_i is more complete that A_i as it also solves certain functionality which was not included in A_i (i.e. D_i is more precise as it has more coverage than A_i), we can also state that D_i *refines* A_i. For sake of completeness, the developer team should have built another model A'_i which covers as much functionality as D_i but at the same level of abstraction than A_i. In this case, we have that A'_i *refines* A_i and that D_i *concretizes* A'_i, but in this case, the two later models are at the same level of completeness. Even though this is both possible and a clearer scenario, to omit A'_i is a general practice.

This scenario is somehow new to our exposition as we are now stating a relationship among models in more than one dimension at the same time. However, we have already done this before as generally when we state the *traces* relationship among a model M_{i+1} and M_i, another relationship is also present due to the fact that M_{i+1} is somehow (i.e. in the sense of a dimension) an improved version of M_i.

Relation with other dimensions. We can relate the *completeness* and *time/process* dimensions similarly to how we have related the latter to the *correctness* dimension. When a model M_{i+1} is built as a refinement of a model M_i, both the *refines* and *traces* relationships take place at the same time. The newer version is created posteriorly to the old one, and the former is a refinement of the latter.

There is no particular relationship with respect to the *abstraction* dimension. However, in order to clarify the difference among both dimensions, let's revisit some of the examples presented earlier when we introduced the *abstraction* dimension. The first example we consider is the waterfall process model; Figure 3 depicts this example. The figure presents three dimensions at the same time, namely *time/process*, *abstraction* and *correctness*. The project being depicted involves four levels of abstraction. As it follows the waterfall process, it first fully develop the upper-most level, say Requirements. The process of developing this model involves five stages in the example, each model being a refinement of the previous one. Once a complete model at the upper-most level of abstraction is obtained, the project advances to the next phase: constructing the corresponding model at the immediate lower level of abstraction. Subsequent phases take place till the final working system is reached when the model at the lowest level of abstraction is completed; such model is colored differently. The different shaded boxes represents models, where darker color means a posterior stage in the development process. The picture shows that as time passes, the model corresponding to the stage of the waterfall process is repeatedly refined till a complete version is reached. At this time, the following stage is tackled in the very same way. It can also be noticed that, at each plane of completeness, the models residing in such plane are organized in a linear curve. As we have mentioned earlier, any other non-increasing function is also possible. It is needed, however, that the oldest model (left-most) of a particular level of

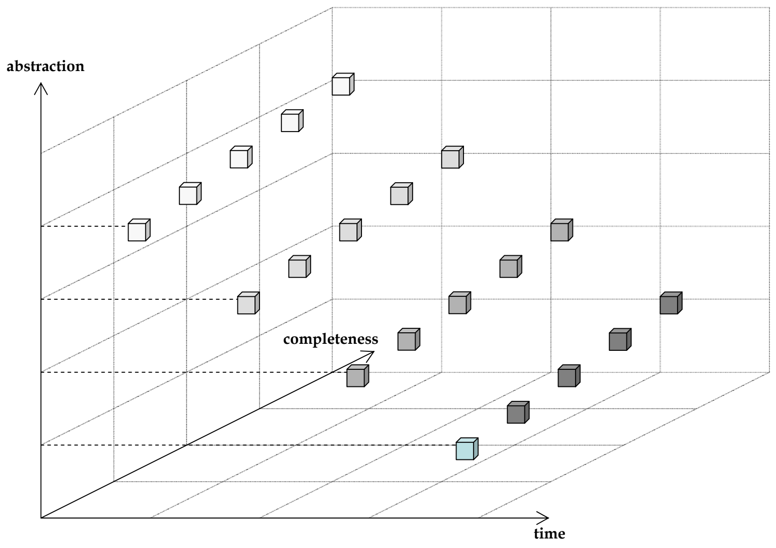

Figure 3: Involved models when following a waterfall process.

abstraction is posterior to the youngest model (right-most) of any upper level of abstraction; such condition is implied by the waterfall process.

The second example to be revisited is the iterative & incremental process model, depicted in Figure 4. This figure considers the same three dimensions as the previous example. The example presents a project consisting of four iterations, and models are organized in terms of five levels of abstraction. The first iteration, in the lightest gray in the figure, builds one initial model at each level of abstraction, one after the other (as before, for simplicity we are considering no parallelism when building models). All models are partial as they are created in the first iteration. The second iteration produces an increment, and hence all models are quite more complete. Subsequent iterations involve a corresponding increment till the final working system is reached; it is colored differently to other models. To improve the understanding of the figure, models (which are drawn as small boxes) are colored in different tones of gray. Each tone corresponds to a given iteration. Models built in the $(i + 1)$-th iteration (shown darker than models at the i-th iteration) are shifted right in the *time/process* dimension as they are built after i-th models, and they are shifted front as they are more complete than i-th models. The slashed lines are only intended to help with the perspective of the picture.

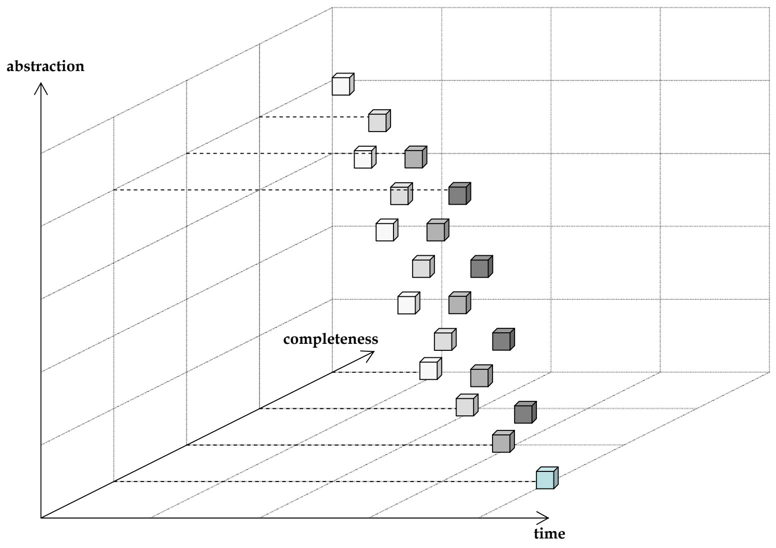

Figure 4: Involved models when following an iterative & incremental process.

4.6 Concern

The *concern* dimension focuses on one of the most tackled problems in software engineering: the separation of concerns. The software engineering discipline has developed several techniques in order to solve how to separate different concerns involved in software development. Object-orientation, by means of encapsulation and information hiding was originally introduced as a promising approach to finally obtain the desired mechanism. Nowadays, additional approaches, mainly complementary to the object-oriented approach have arisen. Aspect-oriented programming, or more generally, aspect-oriented development, and multi-dimensional separation of concerns (MDSOC) techniques, have positioned recently as effective approaches to solve several problems found when using object-oriented development. Apart from them, or conversely in the same direction, model-driven development is positioning as an alternative approach to successfully deal with separation of concerns.

A concern is any piece, fragment or aspect of interest of focus in a system, typically being associated to a system expected feature or behavior. To separate concerns is the process of partition a system into distinct features that overlaps in functionality as little as possible. Each paradigm aids the separation of concerns, mainly by mean of modularity and encapsulation. To organize a software system into separate models can also be considered a paradigm to separate concerns.

The previous dimensions already considered do not tackled the separation of concerns directly. The one that may be most related (or confused) with separation of concerns is *abstraction*. However, the *abstraction* dimension is not about the separation of the system in terms of features or categories of features, it is about the level of meaning at which a particular feature or set of features are represented, human- or computer-oriented. The *concern* dimension involves the separation of these features.

Dimension conformation. There is no broad consensus about which concerns or aspects are important or interesting for all systems, neither for a single system. Software development processes rarely incorporate activities for defining system concerns explicitly. Which concerns are of interest can usually be obtained from requirements; the separation among functional and non-functional requirements provides an initial clear separation. However, considering simply these two concerns is not enough for well-separating concerns. A possible approach to identify the concerns of interest of a system is a more refined categorization of requirements.

Functional requirements can be further categorized. When use-case techniques are employed, each use-case can be consider a concern (or aspect or feature of the system). A finer categorization may be obtained by considering each scenario of each use-case while a coarse-grain one may be attained by using groups (or packages) of use-cases. When use-cases are not used for functional requirement specification, or for those functionalities specified following a different technique, categories or groups of system functionalities can be considered as a concern. Then, a set of well-defined "points" of functionality are considered as concerns, and hence inhabit the *concern* dimension.

Non-functional requirements can also be further categorized. The F**URPS+** categorization identifies and describes several non-functional requirements present in software systems; notice that the "F" refers to functionality and was already considered above. Non-functional requirements refer to or state the desired characteristics that a software system must have or present; such characteristics include availability, efficiency, maintainability, security, persistency, transactionality, reliability, portability, distribution, etc. They can be stated (modeled) qualitatively (at different level of precision) or quantitatively (which are preferable although harder to achieve). Those non-functional requirements that can be isolated and (quantitatively) stated in an independent way, which are desired characteristics of the system being built, should be included in the *concern* dimension.

To sum up, the *concern* dimension is conformed by points corresponding to each category of functional requirements (scenario, use-case, group of use-cases, group of functional system features) and by points corresponding to each desired non-functional requirement that can be quantitatively stated (or modeled).

Localizing a model. A model can be located at any point in this dimension, and clearly it can cover more than one point. A model can cope some or all points corresponding to functional requirements and one or all points from non-functional ones. Also, a model can cope points from both subsets. For example, an analysis model representing all the expected functionality of the software system covers all points in this dimension which refers to functional requirements. A analysis model which represents the realization of a particular use-case would inhabit only one point in this dimension. An implementation model which realizes the functionality of one use-case but also deals with persistency and transactionality inhabits several points, corresponding to functional and non-functional requirements.

Interesting points. All points are of interest in this dimension as concerns are inspired in categories of requirements and all the requirements are expected features of the system. However, it must be clear that not every model inhabits all points. As we mentioned above, a model is a means for modularity and encapsulation (at least at a high level) and hence there are models that are particularly focused on a single point of this dimension. Notice that if each use-case is modeled separately, each of these models refers to a single point. Besides, a model representing the security needs also refers to a single point. It is important to notice also that the final working system should cover all points, and hence, this special model must extend throughout the whole dimension.

Model relationship. At first sight, there appear to be no special relationship between models along this dimension. Two separate models covering different subsets of points, either overlapping or disjoint, need not be specially related as they cover isolated aspects or concerns of the system. Nevertheless, this is not really the case. A model covering one concern may refer to model elements that are defined in other model which covers other concerns. For example, a security model which defines the permissions for each user profile may be somehow related to the actors involved in a separate use-case model. Additionally, a model to quantitatively state the expected performance of the system may specify time restrictions for certain use-cases. As a different example, a model specifying how transactionality is provided may be related to a separate model specifying the persistency mechanism. How the model elements in a model refer to model elements in other (independent) model depends on the particular modeling languages in use; referencing elements by name and/or placing stub elements in the source model are the most common (suggested) practice. Such a relationship among model elements inspires (or permits to derive) a relationship among models.

We propose the *refers to* relationship among models to indicate that the source model includes references to the target model. When a model M inhabiting a particular subset of concerns and containing model elements that allude to model elements in a model M' covering a (possible overlapping) subset of concerns, we say that M *refers to* M'. Even though not required, it should be the case that both M and M' are at the same level of abstraction.

It is important to notice that the relationship through this dimension is not accompanied with a new version of a previous model, as it is the case for the dimensions like *correctness* and *completeness*. The fact that a model M *refers to* a model M' does not imply that one model is a subsequent version of the other. Thus, the *refers to* relationship does not occur at the same time that the *trace* relationship we define through the *time/process* dimension. Notice that the behavior of this relationship is similar to the behavior of the *abstracts* and *concretizes* relationships through the *abstraction* dimension.

Relation with other dimensions. The *concern* dimension can be related to some dimensions which have already been introduced; let's discuss some remarks on this direction.

The *concern* dimension is conformed by points representing functional and non-functional requirements of the system being built. As there is no standard means for modeling system requirements, we cannot establish a direct correspondence to models at high levels of abstraction of the *abstraction* dimension. Generally, use-cases are captured altogether, but apart from non-functional requirements. Hence separate models may be supposed to exist but several concerns may be represented in the same model. On the contrary, to separately specify each of the concerns in a different model conforms a modular approach which favors the reuse. In such scenario, we can think the system requirements to be represented for many models, at least one per point in the *concern* dimension, all of them at the highest level in the *abstraction* dimension. From these models, more concrete models are built towards the final working system at the lowest level. To this end, architectural and design decisions are made in order to build more concrete models that satisfy the specification of those abstract set of models. As we follow this process, the concrete models (i.e. models at lower levels of abstraction) tackle several concerns at once. Notice that generally, to satisfy a non-functional requirement is not only to build a specific software component to this purpose, even though such a component may be needed, a lot of model elements are somehow involved in satisfying the non-functional requirements; these are known as cross-cutting concerns. For example, consider an analysis model that solves all the system functionality, and a security model which specifies the security requirements of the system. A design model (at a lower level of abstraction than the analysis model) can be constructed to provide all the functionality depicted in the analysis model, and also to solve the security requirements by incorporating the needed constructs for this purpose. Finally, at least the bottom-most model, from the *abstraction* dimension perspective, must satisfy all requirements and hence inhabits all concerns.

Then, we can state that, in general, starting from separate models for each concern, all of them at the highest level of abstraction, to build more concrete models from them produce models that covers more than one concern, particularly, the union of the concerns of the original models. To clarify the idea, suppose we have two models A and A' where A' *refers to* A, both at the analysis level of abstraction, and covering different subsets of concerns. Then, we derive a model D at the design level by transforming (or weaving) A and A'. Thus, not only model D *concretizes* A and A', but also it covers the concerns in which both models are involved. So, it is important to remark that the higher the level of abstraction, the smaller the coverage, and conversely, the lower the level the greater the coverage. Notice that following this approach, to "merge"abstract models for building concrete ones involves to widen the coverage, and then, at the last step, the final system covers the whole *concern* dimension.

It is important to notice the difference among being more complete and covering more concerns. The *completeness* dimension is about details and precision, it refers to how deep understanding is represented by the model. The *concern* dimension, however, is about coverage of features, it refers to how much a model embraces all desired features. A model M can be very precise (i.e. it is near the origin in the *completeness* dimension) but only cover one single concern (i.e. a single point in the *concern* dimension). In opposition, a model M can be built with very little detail, just an sketch of a lot of features of the system. Such model, even though covering several (if not all) points of the *concern* dimension, is located very far from the origin of the *completeness* dimension (i.e. it is very incomplete).

It is important to remark here that starting with several models at a high level of abstraction, where each one covers some concerns but incompletely, and needing to "derive" models at a lower level of abstraction which covers those concerns but more completely, is a common scenario in software development projects. Clearly, such derivation implies that the developers take decisions that are beyond concretizing (making more computer-aware). When such decisions are significant to the overall understanding of the system, they mean a hole in the system specification. Here, designers or even implementers take architectural decisions due to incompleteness of the specification, yielding a worse overall structure and quality of the final system.

5 Conclusions

This work was aimed to unveil the concept of *model* and relating terms as *model element*, *modeling language*, and *model transformation*. To throw light upon a broadly-accepted and widely-known concept is not an easy task. This end was pursued by tackling it in three separate directions. • First, we set up the intuitive meaning. We enumerate the aspects we consider relevant for defining the term and then we provide a definition for them. Such definition is inspired in revisited definitions throughout the bibliography. • Second, we look beyond this intuition. We establish the foundation for a formal definition of these related terms. Our formal basis was built in terms of graphs. Additionally, we look forward to somehow validate our foundation. Then, we explore its relationship with well-known techniques for language processing and to already proposed formalizations. In the former case, we present that our formal basis is equivalent to what is known as Abstract Semantics Graph (ASG). In the latter case, we notice that our definition can be mapped to the four-layer architecture and the Meta Object Facility (MOF) initiative. • Third, we attend the applicability of the intuition and this formal basis. The latter permits a deeper understanding of the term, but does not help on how it can be applied in real projects. To assist in this direction, we develop a multidimensional framework that permits the characterization of models by clearly defining six dimensions in which a model can be understood and how different models can be related among this dimensions. In addition, such framework allows us to contextualize several software engineering techniques such us process models, model-driven approaches, object- and aspect-oriented techniques, among others.

The three directions were tackled independently in this work, stating their relationship when needed. Our goal was achieved in the sense that all directions were embraced, and interesting results have emerged while this work was being done. The following fine-grain conclusions can then be remarked.

- *Defintion/Intuition*

  Several authors have recently revisited the definition of the term model and its related terms. These definitions are tending to a unified version as most of them comment, mention or focus on similar aspects. We analyze thoroughly the aspects we consider relevant and, as a result of this analysis, we build a definition for the term.

  Such definition is close to those presented in the bibliography. Our intention was not to innovate, we look forward to do some housekeeping work.

- *Foundation*

  To build the foundation of something implies to map the intuition we have of it into a well-understood and more basic and formal concept. The use of graphs, extended with types and data, provides such required framework.

  In addition to it, the fact that our approach is equivalent to different known techniques, such as Abstract Semantics Graphs and to the four-layer architecture of the OMG, helps on validating our foundation. Moreover, this fact shows that actual approaches are equivalent and also that they follow our intuition.

- *Multidimensional Framework*

  · The multidimensional framework positions as a well-suited tool for understanding the applicability of models and to contextualize software development techniques. The first fact is upheld by noticing that we provide six main aspects involved in the conception of a model, namely the *time* or *process*, the stakeholder *viewpoint* or *perspective*, the level of *abstraction*, *correctness* and *completeness*, and the different *concerns* of interest of a software system. These aspects are somehow treated in the bibliography, but to the best of our knowledge, no thorough exposition has been done. The second fact is noticed as we were able to state how different process models can be depicted in the framework, and we have considered how different paradigms as object- and aspect-orientation and multidimensional separation of concerns are involved or influence the treated dimensions. Although we have also mentioned model-driven approaches, final conclusions on this sense are given next. In what follows, we state further conclusions on the framework.

· *More dimensions.* The proposed framework can be extended by incorporating addiotional dimensions to it. Even though the six dimensions already exposed provide a fine-grain classification of models and permits to identify how different models are related, to add more dimensions may provide even further understanding, unfortunately at the price of more complexity. One of the most clear candidate is a dimension which separate among static and dynamic models. This dimension can be binary and would allow us to state the intention of a model, either static (structural) aspects or dynamic (behavioral) aspects, or both.

Another dimension that was suggested previously refer to specification vs. realization models. This dimension is quite controversial as it seems to be very much tied to the level of abstraction (specifications at high levels and realizations at low levels). However, this is not necessarily the case as we might build an specification model at the implementation level. For example, it can be argued that a testing model consisting of unitary testing routines is an specification model at the implementation level as it aids measuring the correctness of the implementation of the system. Together with this one, several issues should be attended when trying to append such a dimension to the framework.

· *Coping the framework.* Providing that the framework is a six-dimensional artifact, and hence, models are to be catalogued from these six dimensions at the same time, it is a difficult framework to cope with. First, it may be not easy for a developer to clearly identify where a model is located; recall that some dimensions are related to aspects of models we generally deal with qualitatively more than quantitatively. Second, it is a very hard task to depict things graphically, not only when what needs to be depicted involves more than three dimensions, but also simply considering three of them. Recall Figure 4 where we depicted the iterative & incremental process model in the context of three dimensions; in such case it is difficult to cope with the perspective of the figure. To overcome this problems, a computer-tool for aiding on manipulating and visualizing models is required. Such a tool, however, may be difficult to conceive due to the issues stated above.

Nevertheless, the framework throw light upon existing aspects of models and hence, required features that should be somehow handled by any tool working on models. Without the framework, several aspects are merged (or confused) and then less benefits are obtained from the modeling activity.

⊸ *Model-Driven Approaches*

· *Languages.* The formal foundation for models we have built helps on clarify certain facts and decisions that are present in the scope of model-driven approaches.

Model Driven Engineering is about building models in domain specific languages, and repeatedly transforming them in order to obtain a final working system. We have state two important aspects in this work: first, a model needs to be expressed in a language and such language determines and limits what can be expressed by its models, and second, several aspects (dimensions) must be considered when building a model. Then, the need for specific languages that permit to build models in the required points (in the context of this framework) is obvious. Then, this explains why domain specific languages (DSLs) have reemerged in the context of model-driven approaches. Besides, detecting that the foundation built on graphs can be somehow mapped to the four-layer architecture, mainly based on the object-oriented paradigm, elucidate why the initiatives similar to the one of the OMG are gaining acceptance and are gathering most major research on the subject. Additionally, the way transformations can be understood, and to identify the existence of a language to express them all, somehow explain why several authors have proposed extensions of the OMG's Object Constraint Language (OCL) to express them. OCL is a declarative language which allow to express properties on models expressed following the object-oriented paradigm. As models are being stated in languages following this paradigm, the OCL has positioned as a well-suited option.

· *Future directions.* The framework we propose can help on identifying the future directions that model-driven approaches might transit in the following years.

To begin with, it is important to gather together what we have state about the model corresponding to the final working software system. Such model is placed at the last point in the *time/process* dimension, satisfies most *viewpoints*, is expressed at the lowest level of *abstraction*, is totally *complete* and *correct*, and embraces all *concerns*. Such a model cannot be built from scratch, and thus, software development process models try to specify how to reach such a model.

Then, it is also important to identify which models would be desired to build. Such models satisfy several *viewpoints*, but are at the highest levels of *abstraction*, are *complete* and *correct*, and embraces all *concerns*. Then, the main difference with the final system is the *abstraction* gap. In addition, it should be notice that an important effort must be done so as to avoid building lots of models at the higher level of *abstraction*, instead to reuse models should be the desired scenario, mainly those involved in the *concerns* related to non-functional requirements; DSLs can help in this direction as a DSL can provide abstract enough construct whose semantics provide such reuse (of knowledge).

Then, our personal opinion is that model-driven approaches will transit the following lines. First, effort should be directed towards identifying the set of required models at the highest levels of abstraction which are enough for obtaining the final working system by means of automatic transformations. Second, an together with the previous one, several domain specific languages should be defined which encapsulate the knowledge in different concerns and allow developers to express models accurately. Third, more patterns and techniques should be developed in order to simplify the codification of model transformations; these patterns and techniques should provide the realization of those constructs available at the domain specific languages in use, but resolving them at lower levels of abstraction (either design or implementation).

Further work. Several lines of research are suggested from this work, mainly in the two latter directions we explore. • First, even though we present a simple and accurate formalization, it is important to take such formalization a step further. In order to do so, apart from revisiting it so as to improve its precision, it is desired to identify general properties of models that can be stated and derived from the formalization. Besides, it is of major interest to explore how an isomorphism to the four-layer architecture can be stated. Such task would also help on improving the developed foundation. • Second, many ideas related to the framework deserve to be explored. Among others, we can enumerate the followings: to identify an (carefully) incorporate additional dimensions to the framework, to plan and prototype a computer-assisted tool to aid managing, manipulating, visualizing and editing models in the context of the framework, identifying the set of models which suffices to reach a final working system by means of transformations, and studying patterns and techniques which enable automatic transformations from models at high levels of abstraction into models at lower levels.

References

[1] IBM. Rational unified process. `http://www-306.ibm.com/software/awdtools/rup/`, August 2006.

[2] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 2003.

[3] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, Boston, 2004.

[4] OMG. Mda guide 1.0.1. Technical Report omg/2003-06-01, OMG, June 2003.

[5] OMG. Common warehouse metamodel. `http://www.omg.org/cwm`, August 2006.

[6] OMG. Meta object facility. `http://www.omg.org/mof`, August 2006.

[7] OMG. Model driven architecture. `http://www.omg.org/mda`, August 2006.

[8] OMG. Object management group. `http://www.omg.org/`, August 2006.

[9] OMG. Unified modeling language. `http://www.omg.org/uml`, August 2006.

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology. Addison-Wesley, second edition, 2005.

[11] Wikipedia. The free encyclopedia. `http://en.wikipedia.org/`, August 2006.