Some Problems of Current Modelling Languages that Obstruct to Obtain Models as Instruments

José Miguel Cañete Valdeón*, Francisco José Galán Morillo, and Miguel Toro

E.T.S. de Ingeniería Informática. Universidad de Sevilla.
Avda. Reina Mercedes, S/N. 41012. Sevilla. Spain.

Abstract. In this paper we reflect on the usefulness of current modelling languages. We defend that objects elaborated with such languages are instruments that pursue one or several typified purposes, which include: (1) to represent knowledge about some subject, whether real or imaginary; (2) to help in understanding and in answering questions about the properties of some subject; and (3) to stimulate the engineer's creativity in solving some problem. We reason that achieving this instrumental role in modelling languages is a necessary condition for a Model-Driven Software Engineering. However, studying several languages of common use in practice, we claim that there are at least four problem categories that obstruct that useful models as instruments can be elaborated with current modelling languages.

Keywords: models, models as instruments, design of modelling languages, Model-Driven Engineering (MDE), Model-Driven Development (MDD).

1 Introduction

Models have been used for years in Software Engineering by the main methodologies and languages. From being secondary products of life cycles, they are currently beginning to be thought of as central elements in leading the software development process. This idea is being explored by initiatives such as the Model Driven Architecture [13]. It has motivated the recognition of the need of a true Model-Driven Engineering, which admits the importance of models not only in the architecture (as product) but also in the process [7].

We believe that a Model-Driven Software Engineering is only feasible if models are understood as *engineering instruments*. From this perspective, models are much more than simple descriptions of subjects: they are entities that can be used to *assist* the software engineer. We began to explore this idea in a previous position paper [1], where we studied the instrumental role played in current Software Engineering by the conceptual objects that are elaborated with the so-called modelling languages, as well as by models in the scientific sense (e.g. those to estimate cost and effort in the development process).

* For comments about this paper, please write to the address: jmcv@us.es

In this paper we reflect on some problems that obstruct current modelling languages of producing models that can be regarded as *useful* engineering instruments. From the results of our previous study, we point three broad categories of purposes that objects created with such languages may pursue as aspiring engineering instruments. This sets a context in which we identify four types of problems that move these models away from their intended purposes. Our objective is that these problems can be avoided in the design of future modelling languages. We have detected the following categories of problems:

1. Existence of an inadequate ontology.
2. Inability of introducing and controlling uncertainty.
3. Lack of a precise constructive guide.
4. Lack of a forecast of reasoning roads.

After having analysed these deficiencies, we claim that there is an even more important problem: there is a shortage, in current Software Engineering, of modelling languages capable of producing models as instruments of real usefulness. We illustrate our results with well-known examples taken from the modelling bibliography, mainly from the Requirements Engineering field: UML use-cases in the Unified Process [6], analysis models in OMT [11], essential models in Syntropy [2], and problem diagrams in Jackson's Problem Frames approach [5]. We have complemented the exposition of some of the problems with additional notations as Cockburn's approach to use cases [3] and UML state diagrams [12].

The rest of the paper is organized as follows. Section 2 introduces three broad categories of purposes for modelling languages, and it exposes the *intended* purposes of the case studies. Sections 3 to 6 elaborate on the four problems listed above, illustrating each one with the appropriate case studies. At the end of each section we expose our partial conclusions. Section 7 presents overall conclusions extracted from our study, relating the obtained results with our reflections about the role of models in a Model-Driven Software Engineering. We close in Section 8 presenting related works.

2 Categories of purposes and case studies

2.1 Models as engineering instruments

Models have a consolidated role in modern Science. From the fields of Physics, Chemistry and Economics, Mary Morgan and Margaret Morrison [10] reasoned that scientific models behave as *instruments of investigation* with three kinds of functionalities: (1) to assist in the construction of new theories and in the exploration of existing ones; (2) to structure and to show measurements about the world, as well as to serve as measurement and prediction tools; and (3) to help in the design of new technologies and mechanisms to intervene in the world.

We followed the spirit of the former authors in our previous paper [1], where we analysed the instrumental abilities of models in current Software Engineering. We studied not only "models" as objects created with modelling languages, but

also "models" in the scientific sense. In this paper we focus on those objects in the first sense. We can find three, non-disjoint categories of purposes that such objects may aspire to.

The first and most frequent purpose is to serve as a *repository of knowledge* about some subject, whether real or imaginary. The description language usually presents a graphical-textual form. As any narration, descriptive models can come in a variety of levels of approximation towards the described subject (precision) and of formality. The most commonly described subject is the system under development itself; a extremely precise and formal descriptive model about the system is its program code[1], as it contains all the details needed to be executed by a computer, and it is written in a formal language.

Another category of purposes is the *analytical* one. This includes to help in understanding some subject, and to assist in answering questions about it (whether immediate or complex). To use the model is to reason with it, so the modelling language must be associated with a suitable reasoning framework to be employed by the user (the software engineer). There must also exist an underlying theory about the subject under analysis; such a theory constitutes a basis on which the different reasonings can be made.

The third category of purposes is the *creative* one. Such models are capable of stimulating the user to discover ideas or even to suggest her/him ideas about how a certain problem can be solved. As in the preceding case, there must exist both a suitable reasoning framework and an underlying theory about the problem.

2.2 Case studies

Below we present our selected case studies and we catalogue each language with respect to its *intended* purpose/s. In the next sections we will reason if the intended purposes are achieved in fact.

UML use cases and the Unified Process. The Unified Modeling Language [12] is a collection of notations, in principle with no associated guides of construction. The intended purpose seems to be descriptive; pg. 1-1 of [12] states: "The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modelling and other non-software systems". However, the UML is been used by many methodologists as a basis to develop languages with analytical and creative purposes; in some cases they keep the UML as is, and in others they explicitly modify it. An example is the proposal by Jacobson, Booch and Rumbaugh: the Unified Software Development Process (UP) [6]. Their method contributes constructive guides aimed to elaborate UML models that can be used not only as mere descriptions, but also as instruments for reasoning about the development of a software system, as well as for stimulating the modeller with ideas about the composition of the system. To make things concrete, let

[1] It has been with the recent advent of MDA that code has also been considered a model [13, 7, 9]

us take the UML use-case notation. We will centre on a claimed benefit of such models: the capability to drive the development process (chapter 3 of [6]). During the "Analysis workflow", activity "*analyze a use case*" (pp. 203–207) intends to identify analysis classes from use-cases, with the assistance of previously elaborated "domain models". Therefore this activity is considering use-case models as instruments with a *creative* purpose: they are intended to help the engineer to solve the problem of obtaining a set of analysis classes that are part of the essential architecture.

OMT analysis models. The Object Modeling Technique [11], aims in its first phase, analysis, to elaborate three kinds of models[2]: object, dynamic, and functional. With them, the authors pursue two purposes: (a) (descriptive) to devise a precise, concise, understandable, and correct model of the real world (pg. 148); and (b) (creative) to serve as the skeleton of the design (pg. 227). For this second purpose, the authors provide a set of guidelines for using the three former models, consisting in identifying design classes from those in the analysis models (this is called "object design", chapter 10 of [11]). From this perspective, the three kinds of models elaborated during analysis play the role of helping the engineer in developing the software system: they have a creative purpose.

Syntropy essential models. The Syntropy method [2] aims to elaborate three kinds of models: essential, specification, and implementation. Similarly to OMT models in the analysis phase, essential models pursue two purposes: (a) (descriptive) to understand a situation, real or imaginary (pg. 12); and (b) (creative) to help in devising a system specification, i.e., a specification model (pp. 269–272).

Jackson's problem diagrams. Problem diagrams were introduced by Michael Jackson as part of a wider framework [5] aimed to help the engineer to solve the problem of *designing* a correct decomposition for those real-world problems in which a software system ("machine") is required. These models are intended to achieve purposes in the three previously stated categories; some concrete purposes in each category are:

- Representative: the parts of the world where the problem is located, the problem requirements, the machine specification.
- Analytical: to understand a real-world situation, to verify that a proposed machine specification actually satisfies the requirements of a given subproblem.
- Creative: to discover the main involved subproblems, to insert auxiliary subproblems whose introduction may be useful to alleviate other subproblems.

[2] The name of these OMT models is "analysis models". Do not confuse with models that pursue the analytical purpose, formerly introduced.

3 Problem: existence of an inadequate ontology

3.1 Ontologies

One of the ingredients that constitute a modelling language is an ontology, i.e., a universe of concepts and relationships between them. A portion of the ontology is intended to be used by the modeller, so it has an associated notation, whether graphical or textual. The problem is that some modelling languages include an ontology that is not adequate for the pursued purposes.

3.2 Examples

We find a classical example of this in the case of object-oriented languages that intend to *describe* the real world, where the software problem is located, through object-oriented ontologies. The inadequacy of such ontologies for the purpose of describing the world has been noted by authors such as Cook and Daniels [2], and Jackson [5].

If we consider use-case models with the purpose of *describing* the problem domain of a system-to-be, we have another example of an inadequate ontology. It only contains one element to represent all the important aspects in the world: the "actor" concept. This clearly results insufficient when we try to structure and describe the part of world concerned with the system requirements. To this aim, use-case models need to be accompanied in UP with domain models. Domain models in turn contain an excessively simple ontology (based on classes, associations and generalizations), so it needs to be specialized for each application domain (e.g. the "business profile", [12]). In contrast, authors like Wieringa [14] propose a more adequate ontology for the same purpose: physical domains, social domains, conceptual domains, and lexical domains.

The ontology in Jackson's problem diagrams includes, among others, the root concepts of "domain", "shared phenomenon", "interface" and "requirement". The first two are subsequently refined in other concepts. This ontology allows the modeller to acquire the relevant knowledge about a concrete real-world problem. Such knowledge, besides covering the representational purposes stated in the previous section, constitutes, on one hand, the input for a mental pattern-matching process, looking for what Jackson calls "problem frames": patterns of commonly found problems. To this end, the author has defined the frames with the same ontology of problem diagrams. Once identified, a problem frame provides the engineer with additional knowledge about concerns and difficulties of that problem, and even suggests design ideas (e.g. to insert auxiliary subproblems whose introduction may be useful to alleviate other subproblems). On the other hand, problem diagrams are the basis for applying a number of reasoning schemes and strategies that, in turn, allow the application of the theoretical knowledge of each problem frame, thus achieving the analytical and creative purposes stated before. Section 6.2 elaborates on such reasoning schemes and strategies.

3.3 Conclusions

There are some modelling languages whose ontologies are not adequate to their purposes because they are not able to acquire the suitable knowledge to describe what is required or because they do not allow the construction of the reasonings that are needed to apply the underlying theory.

4 Problem: inability of introducing and controlling uncertainty

4.1 The need to introduce and control uncertainty in Software Engineering models

Uncertainty is a natural companion of every engineer when devising an entity. As the development process goes on, uncertainty about the entity is ideally reduced. Therefore, it seems logic that a software engineer can have at her/his disposal a collection of modelling languages that support a variety of uncertainty levels, so that s/he can choose the most comfortable one depending on the stage of the project. Uncertainty may present two forms: (a) to state something about what we are not sure, and (b) to omit certain details we do not know or we are in doubt. But allowing the introduction of uncertainty in a model is not enough: a language should also provide some mechanism to control it. Control means: (1) to specify exactly which model elements introduce uncertainty and how much it consists of; and (2) to have mechanisms for the modeller to raise or to reduce the uncertainty at will.

4.2 Examples

Cockburn's approach to use cases. A place where uncertainty is particularly needed is the one constituted by the early phases of a project, when the requirements are beginning to be discovered. In Alistair Cockburn's book about use cases [3], we observe several examples in which the author needs to express that some steps of the use cases were not known by the analyst at the moment of writing (e.g. step 4a1 in "buy stocks over the web" –pg. 4– and step 3a1 in "get paid for a car accident" –pg. 5). As Cockburn's use cases are described in natural language, the author resorts to interrogative sentences in the form "*what do we do here?*". Therefore, the language does not offer any means to control uncertainty.

Unified Modeling Language. The UML is intended to serve as a language not only applicable to the phase of advanced design, but to all the development process. However, we have not found almost any element in UML's ontology that allows the controlled introduction of uncertainty. The reason is that UML is a language primarily conceived to elaborate precise models. The fact is that modellers often use UML with uncertainty semantics, because that is what they

need. However, the language metamodel does not allow this. As an example, consider the model in figure 1(a). Its purpose is to describe the behaviour of a software system consisting of a controller of a lamp. Consider the LampOn state. What we want to express with UML is the following:

LampOn = state : a "click" order has been sent to the lamp, requesting for turning on, and the user has not issued any more "click" events.

This description does not make explicit the detail about the composition of the LampOn state, perhaps because it was a design decision that the software engineer had not still taken at the moment of elaborating the model, or perhaps because s/he just did not want to indicate that information. However, when the model in figure 1(a) is interpreted in UML's metamodel, a metaclass must be chosen for LampOn: SimpleState or CompositeState. There is not a metaclass that allows the modeller to express such uncertainty about the controller. If no more models are provided, LampOn is interpreted as SimpleState. There is no chance of expressing uncertainty.

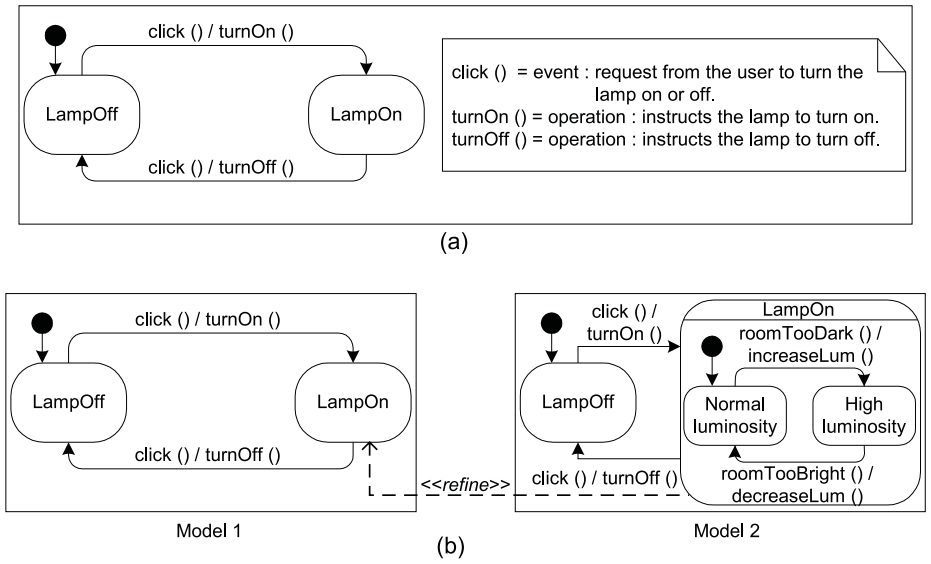

Fig. 1: Modelling the controller of a lamp. According to the UML metamodel, LampOn is a SimpleState in (a), and a CompositeState in (b). However, the modeller wanted to leave such detail unspecified in (a).

Figure 1(b) shows two models. State LampOn in Model 2 *refines* state LampOn in Model 1 through a refine dependency. Although LampOn in Model 1 has exactly the same iconic form than in figure 1(a), it is interpreted as belonging to the CompositeState metaclass, due to the dependency.

The ≪refine≫ dependency is a promising resource to work with different levels of uncertainty. Page 2-18 in [12] states that "(a refine dependency) specifies refinement relationship between model elements at different semantic levels, such as analysis and design". This resource presents two problems. The first one is that a theory must be established for defining which pairs of elements (or pairs of groups of elements) can be legally related through this dependency. The second one is that such related elements should allow "uncertainty" semantics. For example, suppose that when we introduced the operation turnOn() in fig. 1(a), the modeller did not specify its parameters (because they were unnecessary for the model purposes or because the modeller had not still decided them). Assume the operation is detailed in another model as turnOn (p: Integer). UML interprets turnOn() as: "operation with zero parameters". However, if we connect both operations through a refine dependency, the intended semantics for turnOn() should be: "operation that does not specify the number and type of its parameters, but they are detailed at the other side of the dependency".

The other promising resource to work with different levels of uncertainty is the generalization (Generalization metaclass). We can specify some details about a class and later (or in another model) add more details through a subclass. In UML, the new details must be additive wrt. the former ones, i.e., they cannot refine existing elements, with the exception of methods ([12], p. 2-70). UML does not support state machine generalization through the Generalization metaclass. Instead, the language just suggests an example of how the ≪refine≫ dependency could be used to define two kinds of state machine generalization: subtyping and strict inheritance (pp. 2-166 to 2-168), admitting that "these techniques are all based on practical experience" and "this topic is still the subject of research" (p. 2-166).

We must warn that what is not a resource to vary the uncertainty level is the presentation option that allows to show more or less details about model elements; e.g. suppress the type of an attribute in a class ([12], p. 3-43) or suppress the argument list and return type of an operation (p. 3-46). These are just presentation issues, but the modeller is forced to decide the type of an attribute when defining it, as well as the parameters of an operation (as stated by the metamodel, p. 2-13). What we are looking for are resources that allow the modeller to be intentionally imprecise about some topics being modelled, while still constituting valid models.

UML state diagrams contain some constructors with non-deterministic semantics. Such constructors constitute a "lightweight" mechanism to introduce some uncertainty in a model. Some examples are [12]: (a) the indeterminacy about what transition is going to be fired in a "choice" pseudostate when the guard conditions are true (p. 2-146); (b) the indeterminacy in the selection of the next event that is going to be processed by a state machine when there are several candidates (p. 2-161); the uncertainty in the selection of one between several transitions that are ready to be fired (p. 2-161).

4.3 Conclusions

The ability of describing with uncertainty is basic in an Engineering. We have found a shortage of this characteristic in commonly used modelling languages. The introduction of uncertainty in a model should be accompanied by control mechanisms provided by the language.

5 Problem: lack of a precise constructive guide

This problem is present in many languages that were designed with the intent of producing analytical or creative purposes. These languages provide a guide for using models as instruments, but in many cases it is just assumed that the models are *somehow* elaborated, or at most some very general guides are given for that. Although the use guides may be correct, these languages lack from a precise guide that explains how to construct these models, taking into account the pursued purposes.

5.1 Examples

Recall the UML use-case models in the Unified Process (section 2.2). As we reasoned before, they pursue a creative purpose: to help the engineer to solve the problem of obtaining a set of analysis classes that are part of the essential architecture. UP gives some pieces of advice to identify such classes, in the form of guidelines (pp. 204–205), *provided that* the engineer has adequate use-case and domain models. Such guidelines constitute the proposed use guide of the use-case model as an instrument with a creative purpose. But to this purpose to be successful, there should exist a *constructive guide* that *precisely* explains how to elaborate use-case models and domain models; only then, they could behave as the creative instruments that the methodologists had conceived, and the guidelines for identifying analysis classes would be useful. The Unified Process only provides with loose guides to elaborate use-case and domain models. The authors explicitly admit that (pg. 204): "The use cases described in requirements are not always detailed enough for identifying analysis classes [...] Thus to identify analysis classes you may have to refine the descriptions of the use cases with respect to the inside of the system".

Regarding OMT analysis models, there is not a *realistic* constructive guide for elaborating such models. The proposed guide depends on the existence of a preexisting "problem statement": a document where all the requirements are clear and the problem is correctly located and bounded in the real world. The constructive guide proposed by the authors simply limits to identify model elements from such problem statement: object classes, scenarios, state diagrams, etc (chapter 8 of [11]).

Essential models in Syntropy are constructed in a similar, loose way. Authors give a more precise constructive guide for essential models than the one provided by OMT, but it also results insufficient to determine where the problem is located and what are the important world aspects to capture into elements of specification models.

5.2 Conclusions

There are many modelling languages which lack from precise constructive guides. Sometimes, this is due to they pursue too ambitious purposes, so almost every guide would be insufficient, like in the cases of OMT's analysis models and Syntropy's essential models, which are general-purpose methodologies.

6 Problem: lack of a forecast of reasoning roads

6.1 Reasoning roads

One of the elements that should accompany a modelling language is a map of the possible reasoning routes that the potential users (modellers) can follow when elaborating models or when using them (as instruments that they are). In the first case, the language designer must provide precise enough construction guides such that the modeller always obtains valid models, ready to be used. If the models are intended to help in reasoning, the language designer must provide a reasoning framework that guides the reasoning process of the modeller when using the model. If the intention is to design a modelling language for stimulating the modeller's creativity about solving a problem, the language designer must foresee the creative mental roads that may arise in the mind of the modeller when using the model.

6.2 Examples

Regrettably, there is not forecast of reasoning roads in the modelling languages we have studied, with the exception of Jackson's problem diagrams. In the problem frames framework [5] we can find a number reasoning schemes and strategies to be used on problem diagrams as a basis.

For verifying that a subproblem fits a problem frame, Jackson implicitly suggests a "reduction to the absurd" strategy: suppose that it does not fit, try some other candidate frames, and consider if the domain descriptions that such other frames require to collect make sense in the subproblem situation. Examples of this can be found in pp. 141–142 of [5].

For verifying that a proposed machine specification actually satisfies the requirements of a given subproblem, the author provides a reasoning scheme with each problem frame. Each scheme ("frame concern") is a correction argument that tries to prove the so-called "system engineering argument"[3] particularized for the problem frame (chapter 5 of [5]).

Other reasoning strategies in the problem frames approach are based on the particular concerns of each problem frame. For example, chapter 10 of [5] describes how the finding of a mismatch between the requirement phenomena

[3] According to Wieringa [14], the system engineering argument states that the machine specification (S) together with the assumptions about the environment (A) entail an emergent behaviour that must match the stated requirements (E): $S \wedge A \models E$.

and the specification phenomena in an "information display frame" may lead to the introduction of auxiliary subproblems in the decomposition that alleviate the subsequent design of the machine.

6.3 Conclusions

The engineer reasons with the model since s/he begins to elaborate it. Conclusions about the modelled subject can be extracted even from not complete models. Reasoning roads must be forecast as much as possible; schemes and strategies must be supplied to help to drive reasoning. The psychology between engineer and model is an important aspect to be taken into account.

7 Overall conclusions

The overall conclusion that is extracted from our results is that there exists a strong shortage of true modelling languages in the categories "analytical" and "creative". Most languages that intend to produce models as instruments in these two categories have failed in the intent, and their usefulness stays limited to serve as descriptive languages. The four problem categories introduced in this paper contribute to this situation.

We have remarked the necessity of real constructive guides in modelling languages. Devising an ontology for helping in reasoning, or for stimulating the emerging of ideas, is a necessary ingredient for such purposes, but it is not enough. A construction guide must be also included with the language, such that it can realistically lead to the intended models. Otherwise, the language is of little use. The lack of a precise constructive guide has revealed to be a serious problem in current modelling languages. Such a guide must predict the reasoning roads in the mind of the modeller and drive them to elaborate the right models. Forecast of reasoning roads and suggestion of schemes and strategies are also necessary when devising the use of the models by the engineer.

Modelling languages that exhibit a descriptive purpose use to tend to *precise* narration. This is the case of the UML. However, we have reasoned that the controlled introduction of uncertainty is necessary, and not only during requirements elicitation, but also during analysis and design. For example, an early design can contain an association between two classes, without committing any navigational or multiplicity issues. This cannot be currently expressed with UML.

In our opinion, Model Driven Engineering needs more analytical and creative modelling languages. However, descriptive ones are predominant, so current proportions should be inverted, or at least balanced. Besides, the role of models as instruments should be consolidated in Software Engineering. This entails some logical steps. First, it should be made precise the purposes of models as engineering instruments. Our proposal in [1] can be further refined. Second, it must be investigated what elements models must be constituted of, in order to they behave as instruments capable of achieving analytical and creative purposes. We have pointed out some of these elements in this work. And third, engineering

practices should be devised for designing modelling languages capable of producing models in the former purpose categories. We believe that Model Engineering should accompany Model Driven Engineering. These three points constitute our current fields of work.

8 Related works

The nature and functioning of models in scientific practice has traditionally been an important research topic in Philosophy of Science [4]. Mary Morgan and Margaret Morrison have recently made a deep study from the fields of Economics and Physics [10]. Their work has inspired us in the ideas as well as in the method.

Regarding to Software Engineering, there is an increasing interest about the role of models, probably due to the introduction of the Model Driven Architecture (MDA). Stuart Kent goes further, extending the role of models not only to architecture, but also to the process, in what he calls a "Model Driven Engineering" [7]. From another perspective, Jochen Ludewig makes interesting reflections about models in Software Engineering [8].

References

1. J.M. Cañete, F.J. Galán, M. Toro (2004). Conciencia de Modelos como Instrumentos en Ingeniería de Software. Una Aproximación desde las Ciencias Naturales y Sociales. Proceedings of the MIFISIS'2004 workshop (Valladolid, Spain).
2. S. Cook, J. Daniels (1994). *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall.
3. A. Cockburn (2001). *Writing Effective Use Cases*. Addison-Wesley.
4. R. Giere (1997). *Understanding Scientific Reasoning*. Fourth Edition. Harcourt Brace College Publishers.
5. M. Jackson (2001). *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley.
6. I. Jacobson, G. Booch, J. Rumbaugh (1999). *The Unified Software Development Process*. Addison-Wesley.
7. S. Kent. Model Driven Engineering (2002). Proc. of Intergrated Formal Methods (IFM 2002). LNCS 2335, pp. 286-298.
8. J. Ludewig (2003). Models in software engineering - an introduction. *Software and Systems Modelling*, no. 2, pp. 5–14.
9. S. J. Mellor, A, N. Clark, T. Futagami. (2003) Model-Driven Development. IEEE Software, pp. 14-18. September-October 2003.
10. M. Morgan and M. Morrison (1999). Models as Mediating Instruments. In *Models as Mediators. Perspectives on Natural and Social Science*. Cambridge University.
11. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice-Hall.
12. Object Management Group (2001). *Unified Modeling Language Specification version 1.5*. OMG document number formal/03-03-01.
13. Object Management Group Architecture Board ORMSC (2001). *Model driven architecture (MDA)*. OMG document number ormsc/2001-07-01.
14. R. Wieringa (2003). *Design Methods for Reactive Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann.