Computer Languages, Systems & Structures 43 (2015) 139–155

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Model-driven engineering: A survey supported by the unified conceptual model

Alberto Rodrigues da Silva

INESC-ID & Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal

ARTICLE INFO

Article history:
Received 14 November 2014
Received in revised form
19 May 2015
Accepted 6 June 2015
Available online 19 June 2015

Keywords:
Model
Metamodel
Modeling language
Software system
Model-driven engineering
Model-driven approaches

ABSTRACT

During the last decade a new trend of approaches has emerged, which considers models not just documentation artefacts, but also central artefacts in the software engineering field, allowing the creation or automatic execution of software systems starting from those models. These proposals have been classified generically as Model-Driven Engineering (MDE) and share common concepts and terms that need to be abstracted, discussed and understood. This paper presents a survey on MDE based on a unified conceptual model that clearly identifies and relates these essential concepts, namely the concepts of system, model, metamodel, modeling language, transformations, software platform, and software product. In addition, this paper discusses the terminologies relating MDE, MDD, MDA and others. This survey is based on earlier work, however, contrary to those, it intends to give a simple, broader and integrated view of the essential concepts and respective terminology commonly involved in the MDE, answering to key questions such as: What is a model? What is the relation between a model and a metamodel? What are the key facets of a modeling language? How can I use models in the context of a software development process? What are the relations between models and source code artefacts and software platforms? and What are the relations between MDE, MDD, MDA and other MD approaches?

1. Introduction

A model is an abstraction of a system often used to replace the system under study [40,20,37]. In general a model represents a partial and simplified view of a system, so, the creation of multiple models is usually necessary to better represent and understand the system under study. Modeling is a well-known technique adopted by Engineering fields as well as other areas such as Physics, Mathematics, Biology, Economy, Politics and Philosophy [20]. However, in this research we focus on models in the context of Software Engineering and Information Systems fields. That means that our models are thus language-based in nature and tend to describe or prescribe some system as opposed, for example, to models in Mathematics which are understood as interpretation of a theory [11].

Models allow sharing a common vision and knowledge among technical and non-technical stakeholders, facilitating and promoting the communication among them. Furthermore, models make the project planning more effective and efficient while providing a more appropriate view of the system to be developed and allowing the project control to be achieved according to objective criteria [8,85].

E-mail address: alberto.silva@tecnico.ulisboa.pt

http://dx.doi.org/10.1016/j.cl.2015.06.001

In the last decades numerous techniques and modeling languages have been proposed to support the design and the development of complex software systems. Many of these languages were defined in the context of methodological approaches – such as structured, object oriented or unified methodologies/processes, fundamentally with the purpose of facilitating and sharing a common and coherent vision of the system under study and, consequently, of easing the communication among stakeholders [8,84,85,34]. However, during this last decade a new trend of approaches has emerged considering models not just as documentation artefacts, but as central artefacts in the software engineering process. In addition to the benefits referred above, it also allows – through complex techniques such as meta-modeling, model transformation, code generation or model interpretation – the creation or automatic execution of software systems based on those models. These proposals – such as MDA [51,21], Software Factories [24], or recently DSL Engineering [83] – have been classified generically as Model-Driven Engineering (MDE) but also by related names such as model-based engineering (MBE), model-driven development (MDD), model-driven software development (MDSD) [45,3,74,67], or model-based testing (MBT) [80]. Regardless of the adopted term and the particular application, all of them share common concepts and terms that need to be abstracted, discussed and understood.

The objective of this paper is to survey and discuss the essential concepts of MDE and, in particular, to propose a unified conceptual model that clearly identifies and relates those concepts, namely the concepts of system, model, metamodel, modeling language, transformation, software platform and software product. For the sake of simplicity and readability this unified conceptual model is described through a coherent set of UML class diagrams complemented by descriptions and discussions in natural language. The proposed conceptual model is based on earlier work, in particular, on work that relates the conceptualization of models and metamodels [64,20,19,37], modeling languages [46,66], and relations between models, transformations, platforms and software products [74]. However, as further discussed in the Related Work section, unlike these former works, this paper intends to give a simple, broad and integrated view of the key concepts and respective terminology commonly involved in the MDE, answering to common questions like: What is a model? What is the relation between a model and a metamodel? What are the key facets of a modeling language? How can I use models in the context of a software development process? What are the relations between models and source code artefacts and software platforms? and What are the relations between MDE, MDD, MDA and other MD approaches?

This paper is organized in 7 sections. Section 2 introduces the concepts and definitions of system, model and metamodel that are central in MDE. Section 3 defines the concept of a modeling language with its different facets (i.e., abstract syntax, concrete syntax, semantics and pragmatics) and also asserts that a modeling language provides one or more viewpoints and can be classified according the abstraction and perspective dimensions. Section 4 extends the proposed conceptual model by introducing and relating the concepts of software product, software platform, artefacts and model transformations. Section 5 relates and analyses the terms used around the MD approaches. Section 6 compares and discusses our research with the related work. Finally, Section 7 presents the conclusion and identifies issues for future work.

2. Models and metamodels

This section introduces the essential concepts underlying MDE, namely the concepts of system, model, metamodel and their relations.

In the context of MDE, we define "***system*** *as a generic concept for designating a software application, software platform or any other software artefact*". Additionally, as suggested in Fig. 1, a system might be composed of other subsystems and a system may have relations with other systems (e.g., a system may communicate with others).

2.1. Model

A model is an abstraction of a *system under study* (*SUS*, also known as the "Universe of Discourse" or just "system"), which may already exist or is intended to exist in the future.

2.1.1. Model definition

In the absence of a common definition for "model", it is relevant to refer some of its popular attempts, namely the following: (1) model is a set of statements about the system under study [64]; (2) model is an abstraction of a (real or

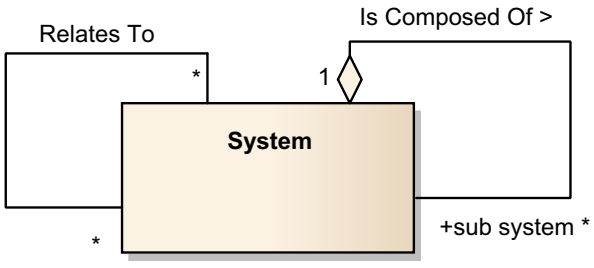

Fig. 1. The System definition.

language-based) system allowing predictions or inferences to be made [37]; (3) model is a reduced representation of some system that highlights the properties of interest from a given viewpoint [65]; and (4) model is a simplification of a system built with an intended goal in mind so a model should be able to answer questions in place of the original system [6].

From these definitions there is a consensus that a *model defines a system under study (SUS) and* vice-versa. However, a *model is itself a system*, with its own identity, complexity, elements, relations, etc. In particular, when we think about a model of a model we have to consider that one of them plays the role of the system under study and, consequently, it is itself a system. To sum up, and as suggested in Fig. 2 we define "***model*** *as a system that helps to define and to give answers of the system under study without the need to consider it directly*".

2.1.2. Criteria and principles

To distinguish a model from any other type of artefact, Stachowiak proposes three criteria for their unique identification [40]: (1) *Mapping criteria*: It must be possible to identify the object or original phenomenon (of the system) that is represented or mapped in the model; (2) *Reduction criteria*: The model must be a simplified version of the original, so not all aspects of the original must be depicted in the model; and (3) *Pragmatism criteria*: The model has to be useful; namely it should be able to replace the original for certain purposes.

The mapping criteria do not require the existence of the original. As in most engineering areas, usually temporally models precede the original and are used extensively to check the completion of the original. The reduction criteria suggests that the model is a simplification of the original, which for instance ignores original features not relevant in the model, or adds features that can enrich the model. Finally, the pragmatism criteria refer to the useful character of models in the sense that they must serve some purpose.

On the other hand, Booch identifies a *set of purposes and benefits* deriving from the existence of models, namely that [7]: models help to visualize a system, as it is or as we want it to be; models allow to specify the structure and the behavior of a system; models give a template that help guide the development process; and models help to document the decisions taken along the project lifecycle.

2.2. Metamodel

Like for the model definition, there is a variety of definitions for metamodel, some of them are unclear or too weak, such as the OMG's definition that simply states that "a metamodel is a model of models" [51]. However, some authors have reflected extensively on these concepts, and give the following definitions: (1) a metamodel is a model that defines the language for expressing a model [51]; (2) a metamodel is a model of a language of models [19]; (3) a metamodel is a specification model for which the systems under study being specified are models in a certain modeling language [64]. However, the deepest analysis of this subject is maybe authored by Kühne that introduced important concerns for this discussion, such as the classification of models as token (or instance) versus type models, ontological versus linguistic instantiations, and the language definition stack [37].

2.2.1. Metamodel definition

Based on the previous referred works we define "***metamodel*** *as a model that defines the structure of a modeling language*". However, from Fig. 3 we still have to understand the following facts: First, through the relationship *ElementOf*, between *Model* and *Modeling Language*, a modeling language is a set of models (or a model is an element of a modeling language). Second, through the relationship *Defines*, between *Metamodel* and *Modeling Language*, a metamodel is a model of a modeling

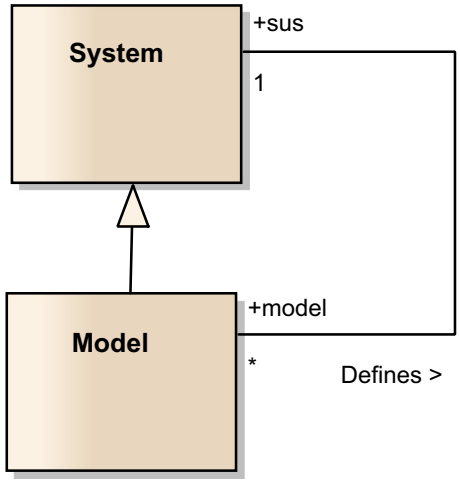

Fig. 2. The Model definition: relationships between model and system.

language structure (or a modeling language is defined by a metamodel). Third, a metamodel is a model of a set of models or is a model of models. Finally, a model conforms with a metamodel (via the *ConformsWith* relation), meaning that the model should satisfy the rules defined at the level of its metamodel as extensively discussed by Kühne and others [37,73,57].

2.2.2. Meta-metamodel, metamodel and model

A well-known and recurring problem of metamodeling is how to set the initial metamodel. If a metamodel is a model of a modeling language, there must be a *meta-metamodel* describing its modeling language, and so on, in higher levels and more abstract meta-metamodels. The common solution to overcome this problem is to use a language that, at a particular level of this hierarchy, describes itself in its own language. There are numerous examples of this solution. In the field of natural languages, the English Language describes itself in English at the level of grammar definition, dictionaries, etc. In the field of programming languages, Lisp is a well-known example of a language that describes itself, providing in particular a Lisp compiler written in Lisp [44].

In the field of modeling languages the solution proposed by OMG, based on a four-layer architecture and directly supported through the *Meta Object Facility* (MOF) [55] is a popular example. At the top of that hierarchy there is the meta-metamodeling layer (designated as M3) that is mainly responsible for providing a language to specify metamodels. MOF is a unique meta-metamodel layer because it is instantiated from its own model, i.e., the MOF is defined in MOF (technically using a restricted set of meta-classes designated by *InfrastructureLibrary*). In the layer below (designated as M2), metamodels are defined by instantiation of the meta-metamodel (i.e., each element of the metamodel is an instance of an element defined in the meta-metamodel). UML [53] or Common Warehouse Metamodel (CWM) [52] are some examples of those metamodels, i.e. examples of MOF instances. In the layer below M2 (designated as M1), the models are defined according to the interest and needs of its users: typically for different application domains and different levels of abstraction, e.g. at the level of business definition, technical requirements, or software design. Note that a user model can contain either model elements (i.e., classes and concrete types e.g., *Video*) or instances of these types (e.g., the instance*:Video*). Finally, the lowest level of the hierarchy (the M0 layer) contains real instances of elements defined in the model that actually exist in the context of a computational environment or even in the real world (e.g., a specific Video in your own laptop).

3. Modeling language

As mentioned in the previous section, a modeling language is defined by a metamodel and is a set of all possible models that are conformant with its respective metamodel. However, to provide a complete definition we should consider other aspects or facets (as suggested in Fig. 4), and consequently we define "***modeling language*** *as a set of all possible models that are conformant with the modeling language's abstract syntax, represented by one or more concrete syntaxes and that satisfy a given semantics. Additionally, the pragmatics (of a modeling language) helps and guides how to use it in the most appropriate way*". Metamodel and notation are common synonyms to, respectively, abstract syntax and concrete syntax [25].

3.1. Abstract syntax

The definition of a modeling language usually starts with the capture and identification of the concepts, abstractions and relations underlying the application domain, which represents domain analysis phase of DS(M)L development [46,32]. This

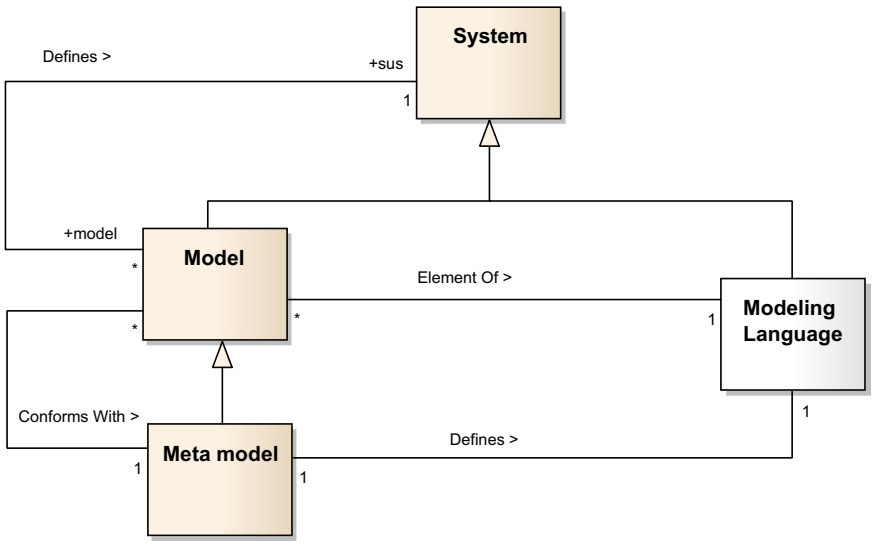

Fig. 3. The metamodel definition: relationships between metamodel and model.

is primarily an exercise of abstraction and conceptualization, and then synthesis of the domain knowledge, that the architect of the modeling language has to know or has to obtain from the direct interaction with the domain experts. The result of this activity produces the *modeling language's abstract syntax*, which corresponds to a metamodel with all the concepts identified at the meta domain level.

The usual techniques to the definition of abstract syntaxes are, in general, *grammars* for natural languages, or *context free grammars* (e.g., specified in compact formalism based on Backus-Naur Form (BNF) notation) for programming languages [57]. However, in the scope of modeling languages the common practice is the use of *metamodeling techniques* [79,46,66]. For example in the context of OMG, the common practice for defining modeling languages is through the use of the UML profile mechanism or directly by using the MOF language (or some of its variations) [66,55].

The abstract syntax defines all the names of the identified concepts, and their respective relations, so it is important that those names would be close to the application domain to be easily understood by its users. On the other hand, its simplicity should be promoted, for example to avoid introducing additional concepts with misunderstood terms that could make it more expressive, but also more difficult to use and to maintain.

As illustrated in Fig. 4, the abstract syntax still includes the *"structural semantics"* (*or static semantics*) that is mainly focused on setting binding rules among its elements. For example, and in a simplified way, structural semantics allows to define that an element of type A can be related to other elements of type B according to this or that constraint; or you can determine that elements of type A can never be related to elements of type C. In general, structural semantics of a modeling language can be described in the different ways: either through a declarative constraints language (e.g., Object Constraint Language (OCL) [56] in the case of UML); through informal natural language specification; or through a blending approach. For example, the structural semantics of UML itself is described extensively in natural language, class diagrams, and some few aspects defined in OCL [7].

The structural semantics is needed because the structure of a language (abstract syntax) is many times context dependent and this cannot be capture by context free metamodels (i.e. grammars) [57]. Additionally, the structural semantics prevents its users from creating models that violate the rules of liaison and the orchestration of its elements, but this can only be achieved if there is proper tool support where such structural semantics are duly captured and validated.

3.2. Concrete syntax

The *concrete syntax of a modeling language refers to its notation*, i.e. the way users will learn and will use it, either by reading or by writing and designing the models. The *notation* is an important aspect of a modeling language because it corresponds to the perspective and user experience that users would have. So, the success of a modeling language will depend on the right balance between simplicity and expressiveness, and in particular the following concerns should be addressed when designing a concrete syntax: writability, readability, learnability and effectiveness [83].

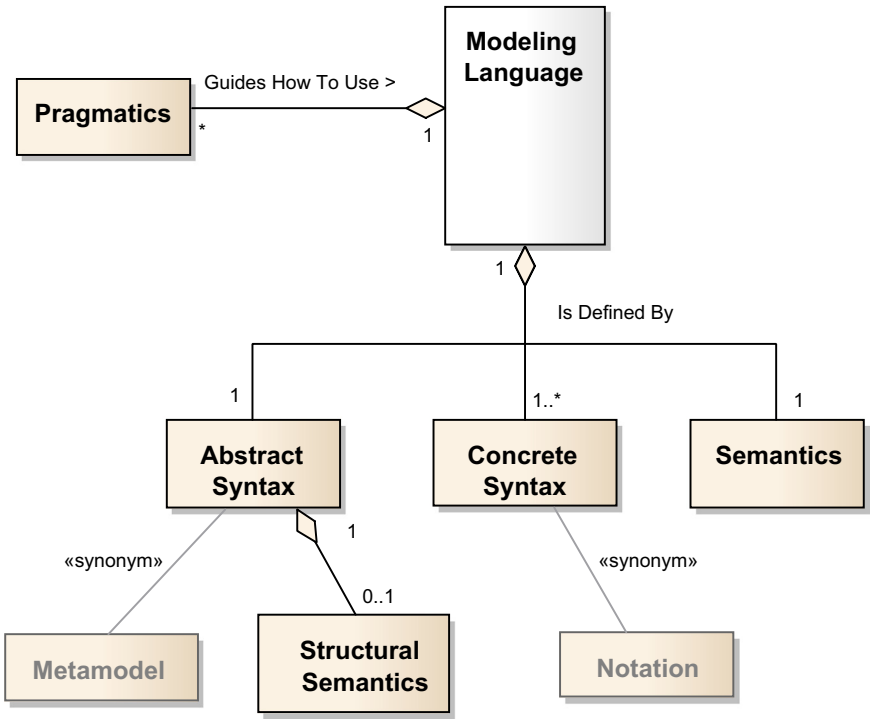

Fig. 4. The Modeling Language definition.

Different notations can be provided, namely graphical, textual, tabular and forms-based, or combinations of them. In general, graphical notations tend to be more suitable to illustrate relations between concepts, changing values in a spatial or temporal distribution, causal and temporal sequences between events, or data and control flows in process modeling scenarios. However, graphical models are not so scalable as textual or tabular-based models, that means they are not the most appropriate to support large models; also they are poorly applicable to write or visualize logical expressions or complex actions – for such, textual notations tend to be more appropriate [74,83]. Finally, because a modeling language might have multiple concrete syntaxes, it would be possible to combine both for the sake of their users: for example to adopt a textual notation for writing/authoring and a graphical notation for just reading/visualization.

3.3. Semantics

In general the semantics reveals the meaning of syntactically valid expressions (or models) specified in a given language. For natural languages, this means correlating sentences and phrases with the concepts, thoughts and feelings based on our experiences and background. For programming languages, semantics describes the behavior that a computer should follow when executing a program in that language. This specification can describe the relationship between the input and output of a program or can provide a step-by-step explanation on how a program will execute on a real or virtual machine.

For modeling languages, and depending on the concepts and models involved, there are two types of *semantics* [26]: executable and non-executable semantics. *Executable semantics* concerns concepts directly related to programming languages, i.e. related to the order of execution of programs, such as those found in state machines, sequence and activity diagrams. On the other hand, *non-executable semantics* concerns concepts not directly related to software execution (or any other kind of execution), such as those concepts involved on the deployment of software components on hardware nodes (e.g., UML component and deployment diagrams), or specification of user requirements (e.g., through UML use case diagrams).

Most of the existing semantics frameworks, developed originally for programming languages [71], are also used to define the semantics of executable models (such as UML state machines or BPMN business process diagrams). Some of these frameworks are “structural operational semantics” and “translational semantics” [71,29]. The *structural operational semantics (SOS)* describes formally a language's semantics by means of a set of inference rules: the individual computation steps are the interpretation of a given concept of the language that will produce an equivalent symbolic representation in a real or virtual machine. The *translational semantics* is a formal way to describe the language's semantics, where the abstract syntax of a source language is mapped into the abstract syntax of a target language which is supposed to be formally defined (e.g., as finite state machine or abstract state machine). Furthermore, the mappings between the source and the target languages might be supported by model language transformations such as QVT, ATL or DSLTrans (see a further discussion below, in Section 4). Regarding non-executable models their semantics can also be defined according a translational framework.

However, and in many cases the semantics of a modeling language is defined in an informal way (by using natural language specifications) instead of following the referred formal frameworks. In the end, and in those situations, the semantics is just taken into consideration during the software system development, preferably through code generation mechanisms, but also by implementing them through source code directly written in a specific programming language.

Finally and based on this discussion, we may classify the semantics of a modeling language according two-orthogonal dimensions: *executable* vs. *non-executable semantics* and *formal* vs. *informal semantics*. A further analysis and discussion of the challenges and directions in formalizing the semantics of modeling languages is proposed by Byrant and others [10].

3.4. Pragmatics

The *pragmatics of a natural language* has been treated as an area of linguistics and is concerned with the study of its use in communication acts in which the factors regarding the context are decisive, such as social, cultural, psychological, historical or geographical factors. While the semantics is concerned with the meaning of language's constructs, pragmatics is concerned with the meaning and interpretation of the language in a context dependent way, i.e. it depends on its users' knowledge and on the various factors on which the communication occurs.

On the other hand, the *pragmatics of a modeling language* has not drawn particular interest from the community, with a few exceptions such as [43,65]. The pragmatics of modeling languages has been focused on the definition and discussion of aspects related to their use in practical contexts, namely in the definition of its types of users or roles (e.g., domain experts, requirements engineers, software architects and end-users), the activities to be conducted (e.g., drawing/writing, refinement, reading, analysis and validation of models), and various factors (e.g., social, environmental or psychological) that may constraint themselves.

Most of the work on pragmatics is concerning how modeling languages can be used in a more efficient and appropriate way, generally involving a set of principles, recommendations and guidelines for their use. There are languages (such as UML) that do not define explicitly any pragmatics, i.e., they deliberately do not define or give any guidance or practices for how they should be used, while others (e.g., SSADM [84], Booch [8] or Yourdon [85]) include the pragmatics in their own definition, in some cases including a well-defined process (or methodology) with practical recommendations and guidelines. Furthermore, the pragmatics might also refers practical aspects of using modeling languages and MDE on real-world projects, such as scalability of the technologies, information overload of large models, efficiency of automatically generated code, and integration with other development tools.

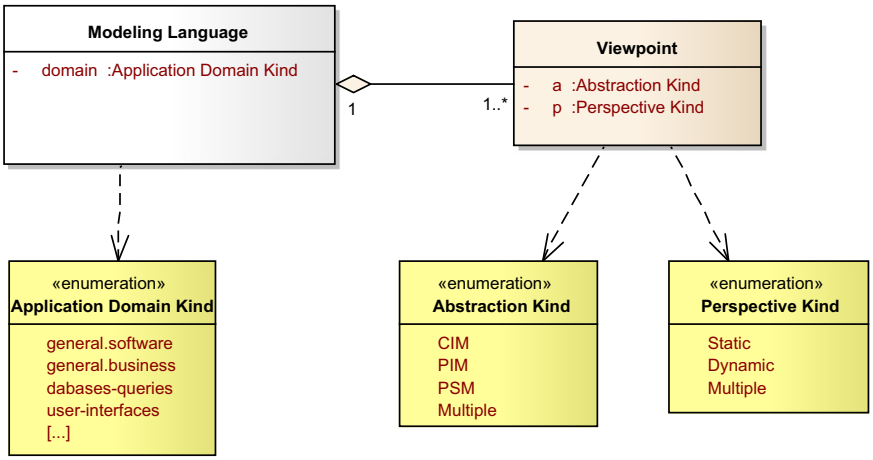

Fig. 5. The classification of a modeling language and its companion viewpoints.

3.5. Classification of a modeling language

In addition to the aspects of a modeling language (introduced above) there is some controversy and debate in the community in what concerns their classification. For example a modeling language might be classified as *general-purpose* (GPML) or *domain-specific modeling language* (DS(M)L) [16,31,41,46,33]. A GPML is characterized by having a greater number of generic constructs, which encourages a wider and widespread use in different fields of application. UML or SysML are popular examples of GPMLs by providing large sets of constructs and notations used for specifying and documenting, respectively, software systems according to the object-oriented paradigm, or any kind of systems as understood by the system engineering discipline. On the other hand, DSLs tend to use few constructs or concepts which are closer to its application domain. Since a DSL is expressed using domain concepts, it is normally easier to read, understand, validate and communicate with, facilitating cooperation between developers and domain experts. Moreover, some argue that DSLs can improve productivity, reliability, maintainability and portability [16,27]. On the other hand, the use of a DSL can raise some problems, such as the cost of learning, implementing and maintaining a new language, as well as the support tools to develop with it [46].

However, others argue that, due to the high-quality and sophistication level of current language workbenches, the tool support is not anymore a main constraint (see further discussion below in Section 5.2). Furthermore, some recent studies have shown that software language engineers do not even have the common practice of evaluating their own languages [22], and conclude that more research is mandatory in the area of software language processes, particularly in what concerns the design, development and evaluation of these languages [22,4,47].

Fig. 5 shows that a modeling language can be classified according its application domain attribute (e.g., based on the values defined in the *ApplicationDomainKind* enumeration) and can be structured by one or more viewpoints.

A viewpoint defines a reusable set of criteria for the construction, selection, and presentation of a portion of a model, addressing particular stakeholder concerns. A *viewpoint* is a general concept that allows applying the *multi-viewpoint modeling principle* or the *separation of concerns principle* that states that a complex system is better defined by multiple views, considering both static and dynamic aspects [7,9]. Therefore, a modeling language provides one or more viewpoints that can be classified according the abstraction and perspective properties. On the *abstraction dimension*, a viewpoint can be classified, for example using the MDA terminology [51], as computational independent model (CIM), platform independent model (PIM), platform specific model (PSM) or multiple (meaning that viewpoint may involve elements defined at multiple abstraction levels). On the *perspective dimension*, a viewpoint can be classified as static, dynamic or multiple (meaning that a viewpoint may involve both static and dynamic elements). A static viewpoint describes a system mainly from its structural perspective with concepts such as classes, objects, nodes, blocks and respective relations. For example, UML class diagrams or component diagrams are static viewpoints. On the other hand, a dynamic viewpoint describes the behavior of a system from a certain perspective with concepts such as tasks, operations, states, events, messages and respective relations. Activity diagrams or state machines in UML or business process diagrams in BPMN are examples of dynamic viewpoints.

3.6. Discussion

Based on the proposed conceptual model, we can better analyse, understand and compare modeling languages. We can also use this conceptual model to help defining our own modeling languages, with their specific facets, principles and guidelines.

For example Table 1 shows the application of this conceptual model to classify two well-known general purpose languages, UML [53] and BPMN [54], and two domain-specific languages, XIS-Mobile ([58]) and DSL3S [72] languages. UML is a GPML for modeling software systems at multiple abstraction levels, and to that purpose UML provides many viewpoints, such as class, object, sequence, use cases, state machine, component diagrams, etc. BPMN may be classified as a GPML for modeling business systems from a dynamic perspective and mainly at a computational independent abstraction level; BPMN provides four viewpoints: process, collaboration choreography and conversation diagrams. XIS-Mobile is a DSML for modeling mobile applications in a cross-platform (or platform independent) way; XIS-Mobile is defined as a UML profile and provides the following viewpoints: domain, business entities, architectural, use cases, navigation space, and interaction space views (or diagrams). Finally, DSL3S is a DSML for spatial simulation in the field of Geographic Information Systems; DSL3S provides the following viewpoints to organized static and platform-independent models: simulation, scenario, animat and animat interactions views.

Furthermore, there are other discussion topics that are worthwhile to mention, namely in what concern the quality of modeling languages. An open topic is the discussion of *what should be these key qualities for analyzing and comparing modeling languages* and their corresponding trade-offs, as for example simplicity, expressiveness, focus, multiple views, or usability.

The *simplicity* of a modeling language can be analyzed at various levels, such as at the level of its abstract syntax (through the definition of its concepts and relations) and its concrete syntax (through the definition of textual or graphical representation). A simple language should have few concepts and its notation should be simple and consistent. Notwithstanding, an expressive modeling language may require the introduction of additional concepts and notations, which would facilitate the work of its users, but, on the other hand, make it less simple. For example, in the context of business process modeling, UML activity diagrams are simpler (but also less expressive and more informal) than Business Process Model and Notation (BPMN) diagrams [54].

Krogstie proposed and discussed extensively this topic of quality of modeling languages and quality of models, in particular with the SEQUAL framework [34]. On the other hand, Barišić proposed a process for evaluating the usability of DSLs [4]. Recently, Morais and Silva also proposed the ARENA framework and used it to compare and evaluate user-interface modeling languages [47].

4. Software products, platforms and transformations

MDE approach claims that the use of modeling languages help to specify models in a certain level of abstraction, and also that those models are used to support the development of software applications [3,74,67]. As suggested in Fig. 6, we define "***software application (or software product)*** *as a system composed of a nontrivial integration of software platforms, artefacts generated through model-to-text transformations, artefacts directly written by developers, and eventually models directly executable in the context of a particular software platform*". (For the sake of simplicity it is not illustrated in Fig. 6, but software applications, platforms, artefacts and models should all be considered "systems" according to the discussion above in Section 2.1.1., see Fig. 1).

Table 1
Classification of modeling languages: UML2, BPMN, XIS-Mobile and DSL3S.

Modeling Language				
Name	**Application Domain**	**Viewpoint**	**Abstraction**	**Perspective**
UML (Unified Modeling Language)	General/Software	Class Diagram	Multiple	Static
		Object Diagram	Multiple	Static
		Sequence Diagram	Multiple	Dynamic
		Use Case Diagram	PIM	Dynamic
		State Machine Diagram	Multiple	Dynamic
		Component Diagram	PSM	Static
		–	–	–
BPMN (Business Process Modeling Notation)	General/Business Processes	Process Diagram	CIM	Dynamic
		Collaboration Diagram	CIM	Dynamic
		Choreography Diagram	CIM	Dynamic
		Conversation Diagram	CIM	Dynamic
XIS-Mobile (DSL for Mobile Apps)	Specific/Mobile Apps	Domain View	PIM	Static
		BusinessEntities View	PIM	Static
		Architectural View	PIM	Static
		UseCases View	PIM	Dynamic
		NavigationSpace View	PIM	Static
		InteractionSpace View	PIM	Static
DSL3S (DSL for Spatial Simulation Scenarios)	Specific/Spatial Apps	Simulation View	PIM	Static
		Scenario View	PIM	Static
		Animat View	PIM	Static
		Animat Interactions View	PIM	Static

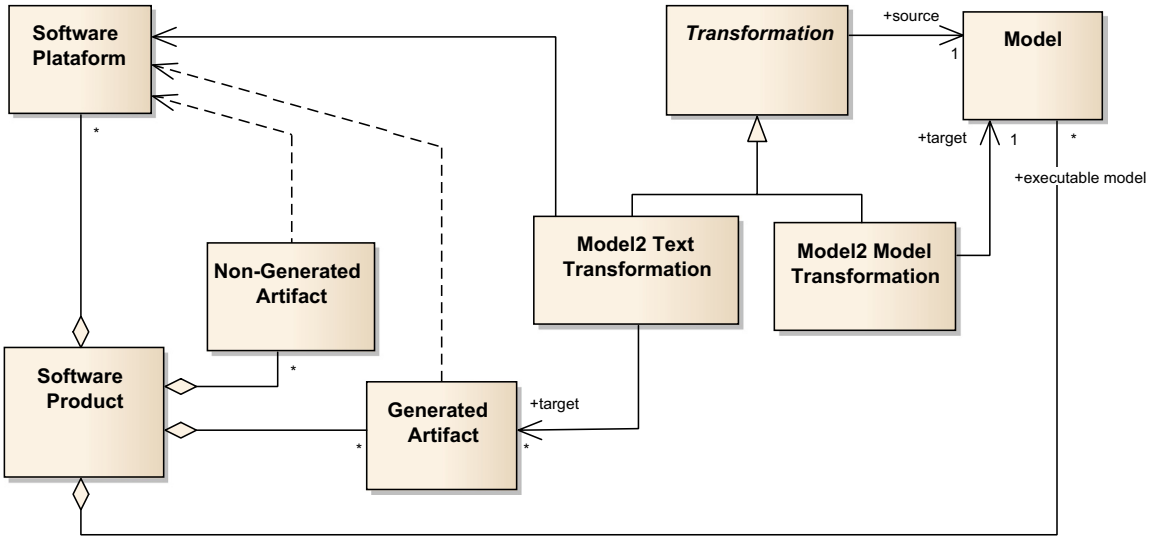

Fig. 6. Software product, platforms, transformations, and models.

First, *software platforms* mean an integrated set of computational elements that enable the development and execution of a class of software products [9,83,24]. Usually these elements provide different functionalities through reuse and extensibility mechanisms and are referred to technologies such as middleware, software libraries, application frameworks, and software components, but also database management systems, web servers, content management and document management systems, workflow management systems, etc. Thus, it is common for a certain class of software product the use and integration of several software platforms, which might have constraints and dependencies among them. For example a mobile application developed for Android depends on the Android platform and on the SQLite database, which depend on the Java Virtual Machine and so on.

Second, *generated and non-generated artefacts* are also elements of the software application. These artefacts might be only relevant during the development time while other artefacts might be relevant at application runtime. Nevertheless, both types of artefacts are tightly dependent on the involved platforms. Many examples of these artefacts might be considered such as: source and binary code files, configuration and deployment scripts, database scripts, and even documentation files, including the models themselves.

Third, two main types of *transformations* tend to be considered in MDE approach. On the one hand, *model-to-text transformations* (M2T) that generate or produce software artefacts – typically source code, XML and other text files –, from models. The most common technique for this class of transformations is known as code generation, and there are multiple solutions and techniques as discussed by Czarnecki and others [14,74]. On the other hand, *model-to-model transformations* (M2M) allow translating models into another set of models, typically closer to the solution domain or that satisfy specific needs for different stakeholders. These transformations are specified through distinct languages, such as the mainstream programming languages, but also by specialized model transformation languages, for different purposes and with different modeling paradigms such as QVT,[1] Acceleo,[2] ATL,[3] VIATRA,[4] DSLTrans[5] as extensively discussed by Syriani and others [76,36,15,23,2].

Fourth, as suggested in Fig. 6, *models* are a central concept of the MDE approach. On one hand, a model can be created directly by users (i.e. model designers) or can be produced automatically from model-to-model transformations and, then, still edited and refined. On the other hand, models can be used to produce generated or non-generated software artefacts, respectively by means of M2T transformations or direct authoring by their users (i.e. software developers). Furthermore, in some particular situations, models can be directly interpreted and executed by specific platforms integrated with the software application [42,13]. Of course, it is important to emphasize that *to be effectively used in the context of the MDE models must be defined in a consistent and rigorous way*. In general, it is required a certain level of quality in order that those models might be properly used in M2M or M2T scenarios. For this purpose there are features that the tools should provide, such as model analysis, validation and simulation as discussed in [81].

[1] QVT. http://www.omg.org/spec/QVT/
[2] Acceleo, http://www.eclipse.org/acceleo/
[3] ATL. http://www.eclipse.org/atl/
[4] http://eclipse.org/viatra/
[5] https://github.com/githubbrunob/DSLTransGIT

5. From abstract to concrete model-driven approaches

This section discusses the final research question of this paper: What are the relations between MDE, MDD, MDA and other MD approaches? This question is still important because these terms are used in the community many times without a clear understanding of their meanings and respective relations.

Fig. 7 shows the proposed conceptual model that introduces these terms and their relations. This model defines a hierarchical structure of MD approaches, each one at a level of abstraction, with the respective generalization and specialization relationships. On the top of this hierarchy it is the most abstract term (MDE) while in the bottom there are some concrete MD approaches that are briefly introduced below. It is not the purpose of this paper to give a complete or exhaustive identification of all middle-level and concrete-level approaches.

Table 2 complements Fig. 7 to help understand and discuss the proposed model. There are a certain number of aspects that were considered to support the analysis and comparison of those MD approaches, namely: abstraction level, software engineering disciplines, models, transformations, meta-modeling languages, application domain, and tool support. The aspect "abstraction level" defines how abstract or concrete is an approach. For simplicity reasons we define just 3 levels: high, medium and low. The aspect "software engineering disciplines" defines the engineering tasks addressed by each approach. For this classification we considered the following disciplines, as defined in Rational Unified Process (RUP) [35]: business

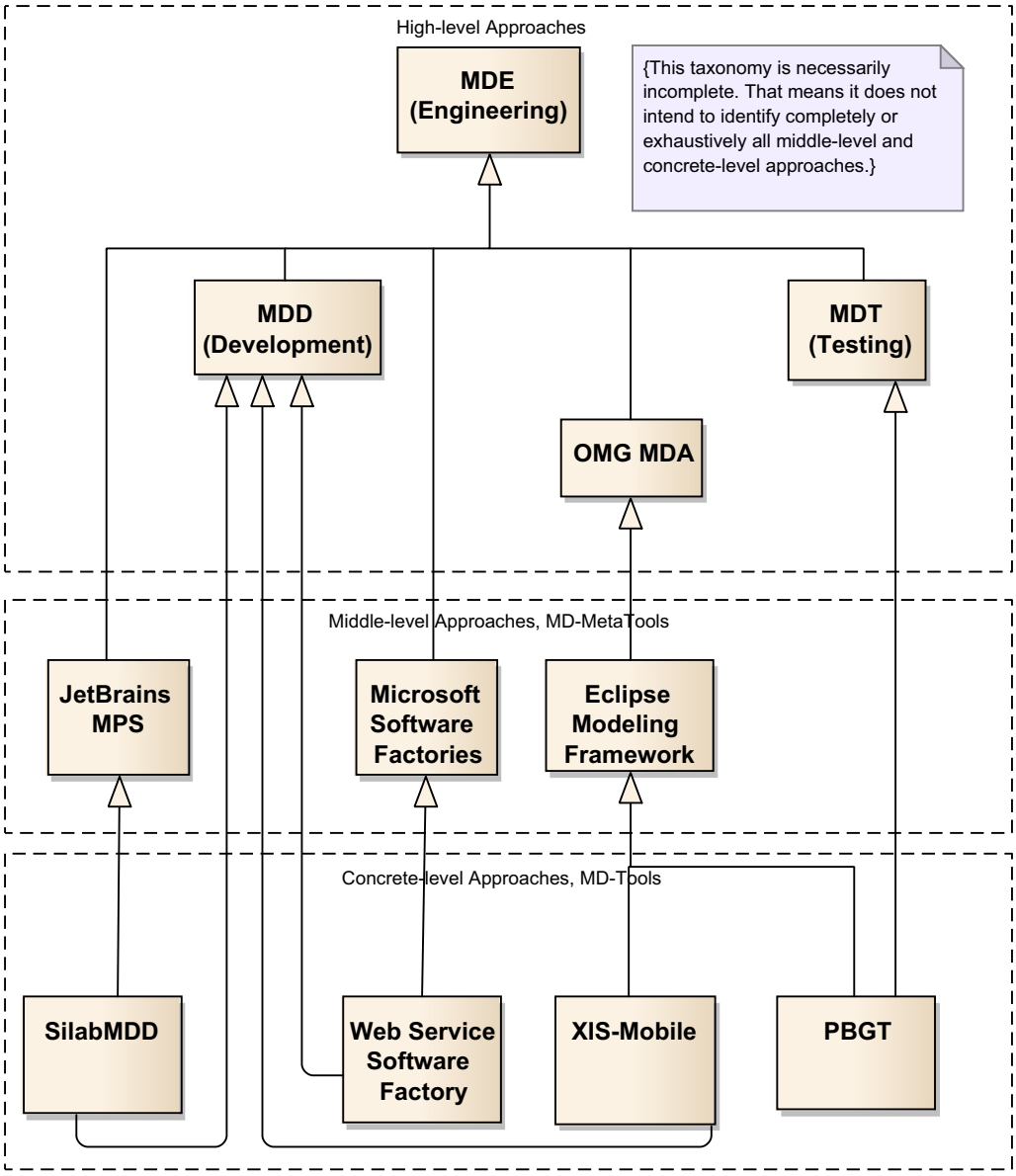

Fig. 7. MDE, related terminology and concrete approaches.

Table 2
Classification of model-driven approaches: from abstract to concrete approaches.

Model-Driven Approaches	Abstraction Level	Software Eng. Disciplines	Models		Transformations		Meta modeling languages	Domain	Tool Support
			Levels	Language	Types	Languages			
MDE	High	Any	ND	ND	ND	ND	ND	Any	ND
MDD		Requirements, Analysis and design, Implementation	ND	ND	ND	ND	ND	Any	ND
MBT		Testing	ND	ND	ND	ND	ND	Any	ND
OMG MDA		Analysis and design, Implementation	CIM, PIM, PSM	UML, UML Profiles	M2M, M2T	QVT	MOF, EMOF, Ecore, UML	Any	Several, e.g., Eclipse Modeling Framework, Enterprise Architect MDG
EMF (Eclipse Modeling Framework)	Medium	Any	Any	UML, UML Profiles	M2M, M2T	Several	Ecore, EMOF	Any	Eclipse and Eclipse Modeling Framework
Microsoft Software Factories		Any	Any	DSLs	ND	.NET languages	UML	Any	Microsoft Visual Studio - Visualization & Modeling SDK
JetBrains MPS		Any	Any	Textual DSLs	M2M, M2T	Java	MPS's Base Language	Any	JetBrains IntelliJ IDEA and MPS
Web Service Software Factory	Concrete	Design, Implementarion	PSM	DSL	M2T	.NET languages	UML	Web Services	Microsoft Visual Studio
XIS-Mobile		Analysis and design, Implementation	PIM	XIS-Mobile (UML Profile)	M2M, M2T	C#, Acceleo	UML	Mobile Apps	Sparx EA, Eclipse Modeling Framework, Accelo
SillaMDD		Requirements, Analysis and design, Implementation	CIM	SilabReq (Textual DSL)	M2T	Java	MPS's Base Language	Requirements of Business Apps	JetBrains IntelliJ IDEA and MPS
PBGT (Pattern Based GUI Testing)		Testing	PIM	PARADIGM	M2T	Java	Ecore	Software Testing	Eclipse Modeling Framework, Selenium

Legend: ND (Not defined or not relevant); Models Levels: CIM (Computing independent Model); PIM (Platform Independent Model); PSM (Platform Specific Model); Transformations Types: M2M (Model to Model), M2T (Model to Text).

modeling, requirements, analysis and design, implementation, testing, deployment; of course this list can be extended with other disciplines such as maintenance and simulation. The aspect "models" defines the abstraction levels (e.g., CIM, PIM, PSM) and the modeling languages supported by each approach, as discussed in Section 3. The aspect "transformations" defines the types (e.g., M2M and M2T) and the languages used by each approach to support the model transformations. The aspect "meta-modeling languages" defines the technologies used to define modeling languages. The aspect "domain" defines the application domains of each approach; usually only concrete approaches are domain-specific. Finally, the aspect "tool support" defines the type of tool supported by each approach.

5.1. High-level approaches

MDE is on the top of abstraction of MD approaches. MDE is a software engineering approach that considers models not just as documentation artefacts but also as first-class citizens, where models might be used throughout all engineering disciplines and in any application domain. In this context, MDE is better classified as a software engineering paradigm and so, it does not have any concrete tool support. At this same level of abstraction are MDD and MBT (as well as other concerns could be considered such as enterprise engineering [17] or ontologies and knowledge representation [68]).

Model-Driven Development (MDD) approach is mainly focused on the requirements, analysis and design, and implementation disciplines [45,3,74,67]. Concrete MDD approaches tend to define modeling languages to specify the System Under Study (SUS) at different levels of abstraction, to provide M2M and M2T transformations in order to improve the productivity and quality of the process and the final software system.

Table 3
Related work analysis – Part 1.

Related Work		Seidewitz, 2003 (2003) [64] What models mean	Stahl et al., (2005) [74], Model-Driven Software Development	Favre et al., (2005) [19], Towards a megamodel to model software evolution through transformations	Kühne, 2006 (2006) [37], Matters of (meta-) modeling
Research Questions	What is a model?	Y	Y	Y	Y
	What is the relation between a model and a metamodel?	Y	Y	Y	Y
	What are the key facets of a modeling language?	N (not explicit)	Y	N (not explicit)	N (not explicit)
	How can I use models in the context of a software development process?	N (not explicit)	Y	N (not explicit)	N (not explicit)
	What are the relations between models and source code artefacts and software platforms?	N (not explicit)	Y	N (not explicit)	N (not explicit)
	What are the relations between MDE, MDD, MDA and other MD approaches?	N	N	N	N

On the other hand, ***Model-Based Testing (MBT)*** [80] is mainly focused on the automation of the testing discipline. Testing models are used to represent the desired behavior of the "System Under Test" (SUT), to represent testing strategies and the testing environment. A testing model describing the SUT is usually an abstract, partial representation of the SUT's desired behavior. Test cases derived from such a model are functional tests on the same level of abstraction as the model, and might then be mapped into executable tests that can communicate directly with the SUT under specific testing tools and frameworks.

Model Driven Architecture (MDA) is the MD approach proposed by OMG, available since early 2000, and focused primarily on the definition of models and their transformations [51,21]. MDA supports the definition of models at different levels of abstraction, namely Computational Independent Models (CIM), Platform Independent Models and Platform Specific Models PSM (PIM). Computational platforms correspond to concrete implementations of application servers, database servers, content management systems, frameworks and software architectures; these platforms can be described themselves through Platform Description Models (PDM). MDA also considers different types of model-to-model transformations, namely: CIM-CIM, CIM-PIM, PIM-PIM, PIM-PSM and PSM-PSM. Additionally, it considers the transformation of PSM models into source code and other types of textual artefacts (PSM-Text). In theory, an application developed under the MDA approach is platform-independent, which allows it to be installed on different computing platforms and properly supports different technologies thanks to these transformations, particularly to PIM-PSM PSM-PSM and PSM-Text transformations.

In spite not being a concrete MD approach, because it did not specify concrete modeling languages and associated tools, MDA still defends the use of several concrete OMG specifications (that is the reason that we put it in Fig. 7 a little below MDD and MBT), namely: (1) MOF as a core component of its meta-modeling architecture; (2) QVT[6] as a set of languages for model query and transformations; and (3) UML Profiles[7] as a simple but practical way to define graphical DSMLs. However, the creation of DSMLs based on UML profile mechanism – i.e., based on the definition of stereotypes, tags and constraints –, is not free of criticism. For example, the definition of a UML stereotype allows adding new properties to the original element, but not to eliminate or inhibit the properties of the original element. In addition, modeling tools that support the UML Profile mechanism usually do not check properly the quality of these models. On the other hand, this mechanism is broadly supported by UML CASE tools and development environments and, consequently, it is used easily to define DSMLs.

5.2. *Middle-level approaches (MD MetaTools)*

In the middle-level of Fig. 7 we identify MD approaches as proposed by their respective companies or communities. These approaches are conducted by technologies and usually supported by complex tools that we refer as "MD MetaTools" and are commonly known as "language workbenches" [18]. Most of these tools provide a collection of features to help users to define DS(M)Ls, with specific editors, model validation, model transformation, etc. Below we briefly introduce Eclipse

[6] QVT (Query/Views/Transformations). http://www.omg.org/spec/QVT/
[7] UML Profiles. http://www.uml.org/

EMF, Microsoft Software Factories, and JetBrains MPS, but many others could be considered as well, namely: MetaEdit+ [79], SDF/Stratego/Spoofax [30], xText [5], Obeo Designer/Sirius [82] or some academic proposals such as MIC (Model Integrated Computing) [78] with the tool GME (Generic Modeling Environment) [1], VMTS [38], MetaSketch [50], or AtomPM ([77]).

A detailed analysis of Microsoft DSL Tools, alongside with equivalent tools (e.g., Enterprise Architect, MetaSketch or MetaEdit+) is discussed in [60] and other comparison of MDE approaches for Web-application development in [61]. Savić et al. report their experience using MPS to implement the SilabReq language, a text language for requirements specification [62]. Additionally, they compare MPS with other alternative tools – namely Spoofax, Obeo Designer, MetaEdit+, XText, Papyrus and EMFText – in what respect the following criteria: support for the abstract and concrete syntax definition, and supported IDEs. Furthermore the annual Language Workbench Challenge (LWC) is another initiative that promotes the comparison and discussion of DSL workbenches. Erdweg et al. present and discuss 10 language workbenches that participated in LWC'2013 [18]. The set of tools analyzed was defined according to the tools that applied to solve an assignment and were subsequently accepted. The assignment was to implement a DSL for questionnaires, which should be rendered as an interactive GUI reactive to user input to present additional questions. Additionally, the DSLs produced did not have the restriction of being graphical.

Eclipse Modeling Project[8] focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations. Eclipse Modeling Project involves an integrated set of extensible tools and frameworks, including EMF (at the core), graphical modeling, textual modeling, and concrete modeling tools particularly supporting OMG specifications, such as UML, OCL, SysML, and BPMN. ***EMF (Eclipse Modeling Framework)*** [75] is the core modeling framework and code generation facility for building tools and applications based on models defined in the Ecore meta metamodel. There are several tools and frameworks developed on top of EMF such as GMF, Sirius, GMF-Tooling, MoDisco, Papyrus, Acceleo, ATL, Epsilon, MMT, Xtext, etc. In general, most of these tools are popular, relatively easy to use and maintain, and they have an open and strong community support.

Microsoft Software Factories [24] is the approach proposed by Microsoft strongly inspired by the "assembly line" metaphor, found in industrial automation areas, which has also been adopted by software engineering through initiatives such as Software Product Lines [12]. A software factory is a structured collection of related software assets used for creating specific types of software, and may include processes, DSLs, templates, integrated development environment, configurations and views. Microsoft Visual Studio offers an integrated suite of tools, called Visualization and Modeling SDK[9] (previously named DSL Tools), which supports the realization of software factories, in particular by providing support to the definition of DSLs, with their respective source code or documentation generators.

Finally, ***JetBrains Meta Programing System (MPS)***[10] is open source and is developed by JetBrains. MPS is is a projectional language workbench, which means that no grammar and parser is involved. Instead, an editor allows changing directly the underlying abstract syntax tree, which is projected in a way that looks like text. MPS supports mixed notations (such as textual, symbolic, tabular, graphical) and a wide range of language composition features based on the BaseLanguage, which is the MPS's meta metamodel. MPS users extend this BaseLanguage to define their own languages: during the process of creating a new language the users directly derive concepts from the BaseLanguage or combine concepts from other existent languages [83].

5.3. Concrete-level approaches (MD tools)

Finally, the bottom of Fig. 7 shows some example of concrete MD approaches, namely XIS-Mobile, WebService Software Factory, SilabMDD, and PBGT. These examples show the applicability of the proposed model in a way that it allows clearly classifying and describing these (and other) MD approaches.

XIS-Mobile[11] is a MDD approach to increase the productivity of developing cross-plaform mobile applications. The XIS-Mobile DSL was defined as a UML profile, with a multi-view organization that supports two design approaches: the dummy and the smart design approach. XIS-Mobile has a supporting framework based on Sparx Systems Enterprise Architect MDG Technology and EMF, which intends to generate source code for multiple platforms from a single PIM model specification, through M2M and M2T transformations. Composed of four major components, this framework suggests developing a mobile application in four steps whenever possible: defining of the required views using the Visual Editor, validating them using the Model Validator, generating the User-Interfaces View models with the Model Generator, and finally generating the application's source code through the Code Generator. This way the developer takes advantage of the MDD benefits, namely increasing his productivity by using a single specification of the system, by avoiding the implementation of boilerplate code and reducing errors ([58,59]).

[8] Eclipse Modeling, http://eclipse.org/modeling/
[9] VSVMSDK (Visual Studio Visualization and Modeling SDK). http://archive.msdn.microsoft.com/vsvmsdk
[10] JetBrains MPS, https://www.jetbrains.com/mps/
[11] XIS-Mobile, https://github.com/MDDLingo/xis-mobile

Table 4
Related work analysis – Part 2.

Related Work		Seidewitz, 2003 (2003) [64] What models mean	Stahl et al., (2005) [74], Model-Driven Software Development	Favre et al., (2005) [19], Towards a megamodel to model software evolution through transformations	Kühne, 2006 (2006) [37], Matters of (meta-) modeling
Definition of	System	Y	N (not explicit)	Y	Y
	Model	Y	Y (Formal Model)	Y	Y
	System to Model relation	N (not explicit)	N (not explicit)	Y (RepresentationOf)	Y (model-of)
	Metamodel	Y	Y	Y	Y
	Model to Metamodel relationships	N (not explicit)	Y (instanceof)	Y (ConformsTo)	Y (instance-of)
	Modeling Language	Y	Y	Y (with the ElementOf association)	Y
	Modeling Language Facets	N	Y	N	N
	Transformation	N	Y	Y (with the IsTransformedIn association)	N
	Software Platform	N	Y	N (not explicit)	N (not explicit)
	Software Product	N	Y	N (not explicit)	N (not explicit)
Approach	Discussed	Textually, with a generic and informal discussion of concepts	Textually and UML class diagrams, deep discussion of concepts	Textually and UML class diagrams, deep discussion of concepts	Textually and mathematically, deep discussion of concepts
	Key concepts discussed	models, metamodels, modeling languages, in general terms	models, metamodels, modeling languages, transformations, platforms, products	models, metamodels, modeling languages, transformations	models, metamodels, modeling languages
	Complementary issues discussed	four-layer metamodeling architecture; meaning of models	domains, domain-specific languages, model2model and model2platform transformation, software system families	transformation models; software evolution	descriptive vs prescriptive models; token models vs type models; classification vs generalization; ontological vs linguistic instatiation

Web Service Software Factory[12] (also known as the Service Factory) is an example of the Microsoft Software Factories and is a concrete MDD approach that provides an integrated collection of resources designed to help to quickly and consistently build Web services that adhere to well-known architecture and design patterns. These resources consist of patterns and architecture topics in the form of written guidance and models with code generation in the form of tools integrated with Visual Studio.

SilabMDD is a MDD approach particularly focused on the Requirements discipline and, consequently, also known as a MDRE (Model-driven Requirement Engineering) approach [39]. SilabMDD includes the SilabReq language which is implemented with JetBrains MPS. SilabReq is a textual DSL that allow users to define and manage requirements based on use-cases specification. Consequently, SilabReq imposes a rigorous definition of use case specification, particularly based on the description of sequences of actions, pre- and post-conditions, and the relations between use cases and elements defined at domain models (still specified textually). The goal of SilabMDD is to provide a complete software development workbench (by extending JetBrains MPS) to be used by requirements engineers, developers, as well as non-technical stakeholders [62,63].

PBGT is a MBT approach that provides generic test strategies (based on user interface (UI) Test Patterns), with multiple configurations for testing different implementations of UI Patterns. PBGT approach is supported by the PBGT tool, in which the UI Test Patterns are defined within a domain specific language, PARADIGM, developed on top of the Eclipse Modeling Framework using the Ecore meta metamodel. PBGT tool is freely available as an Eclipse plugin.[13] It is a fully integrated

[12] Service Factory, http://servicefactory.codeplex.com/
[13] PBGT Tool, http://paginas.fe.up.pt/~apaiva/pbgtwiki/doku.php?id=tools

testing environment that provides functionalities for modeling (either manually or automatically), configuration, automated test case generation, automated test case execution and test coverage analysis. Currently the PBGT tool is able to test both web and mobile (Android based) applications on top of the Selenium framework[14] ([48], 2014).

6. Related work

As mentioned in Section 1, the objective of this paper is to survey and discuss the essential concepts of MDE and supporting this survey by a unified conceptual model that would help to clearly identify and relate those concepts. This research started from earlier work, in particular on the conceptualization of models and metamodels [64,20,19,37], modeling languages [46,66], and relations between models, transformations, platforms and software products [74]. However and differently from those works, this paper intends *to give a simple but broad and integrated view of the key concepts and respective terminology, and answering to common questions* as defined in Table 3. Tables 3 and 4 summarize the comparative analysis of the related work, those that are more related with this paper, namely focused on the following references [64,19,74,37].

The paper "What models mean" [64] introduced the concepts of system, model, metamodel and modeling language and proposed respective definitions. However, this paper did not present any type of conceptual model, and so it is not clear the relations between those concepts. Also this paper did not discuss any other of the MDE concepts like we did in this paper.

The book "Model-Driven Software Development" [74] gives a good and broad overview of the main concepts of MDE, and also tries to explain them through a UML-based model. However, it is not completely clear the definitions of system and model, as well as its relations. However, in spite not being a research paper this is a relevant reference.

The paper "Towards a megamodel to model software evolution through transformations" [19] gave a good inspiration and starting point to propose this survey, in particular with the idea of a "megamodel for MDE". The focus of that paper was just on modeling large-scale software evolution processes. Nevertheless, it also provided some discussion and definitions on concepts such as system, model, metamodel and model transformations. However, contrary to this paper, only system and transformation were defined as first citizen concepts in that megamodel. Additionally, that paper did not define concepts, such as of modeling language, software product, software artefact and software platform.

The paper "Matters of (Meta-) Modeling" [37] provides a formal and extensive definition on the concepts of system, model, metamodel and modeling language. Additionally, it gives a deep discussion on related subjects such as: token models, type models, classification versus generalization, ontological versus linguistic instantiation. That is a significant paper that adopted a formal approach (based on Mathematics expressions) to clearly and unambiguously defined the referred concepts. However, this paper did not refer the other concepts discussed throughout this paper, such as modeling language facets, model transformations, software platform and software products.

Finally, none of these papers try to answer and discuss our last research question concerning the relations between MDE and other MD approaches.

7. Conclusion

MDE is a relatively new engineering approach with some expectations and challenges to be addressed in the next years [49]. As discussed throughout this paper there are some concrete proposals and many more tools and platforms that, in some way would achieve that general vision and relevance.

For an effective implementation of MDE several features should be supported and integrated into appropriate tools, typically classified as IDEs, CASEs and MetaCASEs. Among those we refer the following that can exist in most of these tools, namely to support: (1) the creation of modeling languages, in particular the creation of their abstract and concrete syntaxes and the corresponding structural semantics; (2) multi-user and collaborative environment for models design and management; (3) model validation, with eventual support for model analysis and simulation; (4) model-to-model transformations; (5) model-to-text transformations; and even in some cases (6) models interpretation and execution.

A variety of tools that embody the main ideas of MDE have been developed and improved over this last decade. Some of them correspond to tools developed in an academic environment, as is the case of experiments carried out under GME [1], ProjectIT ([69]), VMTS [38], MetaSketch [50], or AtomPM ([77]). Other tools are commercial, such as the case of Microsoft Visual Studio Visualization and Modeling SDK, Sparx Enterprise Architect,[15] Metacase MetaEdit+,[16] or Obeo Designer.[17] Beyond these, it is worth to highlight some tools and technologies currently developed around the Eclipse Modeling Project and the JetBrains MPS.

The proposed survey on MDE is the result of our research experience in the area throughout this last decade, during which we have designed and developed several modeling languages, tools and real-world applications following the MDE approach [69,70,60,72,58,58,59]. The unified conceptual model proposed in this paper might help others to have a broad vision and a better understanding of MDE and its key concepts and terminology. First, it defines the concepts of system,

[14] Selenium, http://docs.seleniumhq.org/
[15] Sparxs Enterprise Architect MDG Technologies. http://www.sparxsystems.com.au/resources/mdg_tech/
[16] Metacase MetaEdit+. http://www.metacase.com/mep/
[17] Obeo Designer, http://www.obeodesigner.com

model and metamodel. Second, it extensively defines the concept of modeling language, with its multiple aspects, namely abstract syntax, concrete syntax, semantics and pragmatics. Third, it defines, in a cohesive way, the remaining concepts: software product, software platform, software artefact, model-to-model and model-to-text transformations. Finally, the paper also clarifies and discusses common terminology, namely by relating MDE with MDD, MBT, MDA and other MD approaches.

As future work we consider that this survey might be extended by being applied to describe and discuss concrete modeling languages and concrete MDE approaches. Additionally, it can be used as a general and conceptual framework to better analyse MDE practices in industry such as discussed in [28].

Acknowledgments

This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the projects POSC/EIA/57642/2004, CMUP-EPB/TIC/0053/2013, UID/CEC/50021/2013 and DataStorm Research Line of Excellency funding (EXCL/EEI-ESS/0257/2012). Thanks to my colleagues and PhD and MSc students for their strong participation and involvement in this research throughout the last decade. Finally, thanks to the anonymous reviewers for their relevant criticism and suggestions that helped to improve the paper.

References

[1] Agrawal A, Karsai G, Ledeczi A.: An end-to-end domain-driven software development framework. In: Companion of the 18th annual OOPSLA '03 ACM SIGPLAN Conference, ACM; 2003.
[2] Al-Sibahi AS. On the Computational Expressiveness of Model Transformation Languages, Technical Report, IT University of Copenhagen; 2015.
[3] Atkinson C, Kühne T. Model-driven development: a metamodeling foundation. IEEE Software 2003;20(5):36–41.
[4] Barišić A, Amaral V, Goulao M. Usability evaluation of domain-specific languages. In: International Conference on the Quality of Information and Commuications Technology (QUATIC'2012), IEEE Computer Society; 2012.
[5] Bettini L. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd; 2013.
[6] Bézivin J, Gerbé O. Towards a Precise Definition of the OMG/MDA Framework. In: IEEE international conference on automated software engineering; 2001.
[7] Booch G, Rumbaugh J, Jacobson I. The Unified Modeling Language User Guide. Addison Wesley; 1999.
[8] Booch G. Object-Oriented Analysis and Design with Applications. 2nd EditionAddison Wesley; 1994.
[9] Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synthesis Lectures on Software Engineering, Morgan & Claypool, 2012.
[10] Bryant BR, Gray J, Mernik M, Clarke PJ, France RB, Karsai G. Challenges and directions in formalizing the semantics of modeling languages. Comput Sci Inf Syst 2011;8(2):225–53.
[11] Chang CC, Keisler HJ. Model theory. Elsevier; 1990.
[12] Clements P, Northrop LM. Software Product Lines: Practices and Patterns. Addison-Wesley; 2001.
[13] Crane ML, Dingel J.: Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds. ACM; 2008.
[14] Czarnecki K, Eisenecker UW. Generative programming: methods, tools, and applications. Addison-Wesley; 2000.
[15] Czarnecki K, Helsen S.: Classification of model transformation approaches. In: OOPSLA workshop on generative techniques in the context of model driven architecture; 2013.
[16] Deursen A, Klint P, Visser J.: Domain-specific languages: an annotated bibliography. ACM Sigplan Notices; 2000.
[17] Dietz J, Proper E, Tribolet J. (Series Editors): The Enterprise Engineering Series, Springer; 2009–2014.
[18] Erdweg S, van der Storm T, Völter M, Boersma M, Bosman R, Cook WR, et al. The state of the art in language workbenches. Softw Lang Eng 2013: 197–217 Springer.
[19] Favre J-M, Nguyen T. Towards a megamodel to model software evolution through transformations. Electron Notes Theor Comput Sci 2005;127:3.
[20] Favre J-M. Megamodelling and Etymology. Dagstuhl Seminar (Transformation Techniques in Software Engineering); 2005.
[21] Frankel D. Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley; 2003.
[22] Gabriel P, Goulao M, Amaral V.: Do Software Languages Engineers Evaluate their Languages? arXiv:1109.6794; 2011.
[23] Gomes, C., Barroca, B., Amaral, V.: Classification of model transformation tools: pattern matching techniques. In: Model-Driven Engineering Languages and Systems, LNCS 8767, Springer; 2014. p. 619–35.
[24] Greenfield J, Short K, Cook S, Kent S. Software factories: assembling applications with patterns, models, frameworks, and tools. Wiley; 2004.
[25] Harel D, Rumpe B. Modeling languages: syntax, semantics and all that stuff. tech. report MCS00-16, Weizmann Institute of Science; 2000.
[26] Heering J, Mernik M. Domain-specific languages in perspective. CWI; 2007 Technical Report, SEN-E0702.
[27] Hermans F, Pinzger M, Van Deursen A. Domain-specific languages in practice: a user study on the success factors. In: Model driven engineering languages and systems, LNCS 5795, Springer; 2009. p.423–37.
[28] Hutchinson J, Whittle J, Rouncefield M. Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure. Sci Comput Program 2014;89:144–61.
[29] Kamandi A, Habibi J. A survey of syntax and semantics frameworks of modeling languages. In: Proceedings of Computer Science and its Applications (CSA'2009), IEEE Computer Society; 2009.
[30] Kats LC, Visser E. The spoofax language workbench: rules for declarative specification of languages and IDEs. In ACM Sigplan Notices, 45(10), 2010. p. 444–63.
[31] Kelly S, Tolvanen JP. Visual domain-specific modelling: benefits and experiences of using metacase tools. ECOOP Workshop on Model Engineering 2000.
[32] Kelly S, Tolvanen JP. Domain-specific modeling: enabling full code generation. Wiley; 2008.
[33] Kosar T, Oliveira N, Mernik M, Varanda Pereira MJ, Črepinšek M, da Cruz DC, et al. Comparing general-purpose and domain-specific languages: An empirical study. Comput Sci Inf Syst 2010;7(2):247–64.
[34] Krogstie J. Model-Based Development and Evolution of Information Systems – A Quality Approach. Springer; 2012.
[35] Kruchten P. The rational unified process: an introduction. Addison-Wesley Professional; 2004.
[36] Kühne T, Mezei G, Syriani E, Vangheluwe H, Wimmer M. Explicit transformation modeling. Models Softw Eng 2010:240–55.
[37] Kühne T. Matters of (Meta-) modeling. Softw Syst Model 2006;5(4):369–85.

[38] Levendovszky T, Lengyel L, Mezei G, Charaf H. A systematic approach to metamodeling environments and model transformation systems in VMTS. Electron Notes Theor Comput Sci 2005;127(1):65–75.
[39] Loniewski G, Insfran E, Abrahão S.: A systematic review of the use of requirements engineering techniques in model-driven development. In: 13th international conference on model driven engineering languages and systems (MODELS'10), Springerl; 2010.
[40] Ludewig J. Models in Software Engineering – An Introduction. Softw Syst Model 2003;2(1):5–14.
[41] Luoma J, Kelly S, Tolvanen JP. Defining domain-specific modeling languages: collected experiences. OOPSLA Workshop on Domain-Specific Modeling; 2004.
[42] Luz MP, Silva AR. Executing UML Models. In: Proceedings of the 3rd workshop in software model engineering, IEEE Computer Society; 2004.
[43] Martin J, Odell J. Object-oriented methods. Pragmatic considerations.Prentice Hall; 1996.
[44] McCarthy J. History of LISP. In: History of programming languages I. ACM; 1978. p. 173–85.
[45] Mellor S, Balcer M. Executable UML: a foundation for model driven architecture. Addison Wesley; 2003.
[46] Mernik M, Heering J, Sloane A. When and how to develop domain-specific languages. ACM Comput Surv 2005;37(4):316–44.
[47] Morais F, Silva AR.: Assessing the Quality of User-Interface Modeling Languages. In: Proceedings of ICEIS'2015 Conference, SCITEPRESS; 2015.
[48] Moreira RM, Paiva AC. PBGT tool: an integrated modeling and testing environment for pattern-based GUI testing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, ACM.
[49] Mussbacher G, Amyot D, Breu R, Bruel JM, Cheng BH, Collet P, et al. The relevance of model-driven engineering thirty years from now. Model-driven engineering languages and systems (MODELS'2014), Springer; 2014.
[50] Nóbrega L, Nunes N, Coelho H.: The meta sketch editor. In: Proceedings of the sixth international conference on computer-aided design of user interfaces (CADUI'2006), Springer; 2008.
[51] OMG: Object Management Group – MDA (Model Driven Architecture) Guide Version 1.0.1; 2001 Available at ⟨http://www.omg.org/mda/⟩.
[52] OMG: Object Management Group – Common warehouse metamodel (CWM); 2003 Available at: ⟨http://www.omg.org/spec/cwm/⟩.
[53] OMG. United Modeling Language Infrastructure Specification, Version 2.4.1; 2011 Available at ⟨http://www.uml.org/⟩.
[54] OMG. Business Process Model and Notation (BPMN), Version 2.0.2; 2013 Available at: ⟨http://www.omg.org/spec/BPMN/⟩.
[55] OMG. Object Management Group - Meta Object Facility (MOF) Core Specification, v2.4.2.; 2014 Available at: ⟨http://www.omg.org/mof/⟩.
[56] OMG. Object Constraint Language (OCL), v2.4; 2014 Available at: ⟨http://www.omg.org/spec/OCL/⟩.
[57] Paige RF, Kolovos DS, Polack F. A tutorial on metamodelling for grammar researchers. Sci Comput Program 2014;96:396–416.
[58] Ribeiro A, Silva AR. XIS-Mobile: a DSL for Mobile Applications. In: ACM Symposium on Applied Computing (SAC), ACM; 2014a.
[59] Ribeiro A, Silva AR. Evaluation of XIS-Mobile, a domain specific language for mobile application development. J\ Softw Eng Appl 2014;7(11):906–19.
[60] Saraiva J, Silva AR. Evaluation of MDE tools from a metamodeling perspective. J Database Manag 2008;19(4):50–75.
[61] Saraiva J, Silva AR. A reference model for the analysis and comparison of MDE approaches for web-application development. J Softw Eng Appl 2010;3(5):419–25.
[62] Savić D, Silva AR, Siniša V, Lazarević S, Antović I, Stanojević V, et al.: Preliminary experience using JetBrains MPS to implement a requirements specification language. In: Proceedings of the Eighth international conference on the quality of information and communications technology. IEEE Computer Society; 2014.
[63] Siniša V, Lazarević S, Stanojević, Antović I, Milić V, Silva M, et al. Model driven approach. In: Proceedings of the 5th international conference on information society and technology (ICIST 2015), Society for Information Systems and Computer Networks; 2015.
[64] Seidewitz E. What models mean. IEEE Softw 2003;20(5):26–32.
[65] Selic B. The pragmatics of model-driven development. IEEE Softw 2003;20(5):19–25.
[66] Selic B. A systematic approach to domain-specific language design using UML. In: Proceedings of the international symposium on object and component-oriented real-time distributed computing (ISORC); 2007.
[67] Selic B. Personal reflections on automation, programming culture, and model-based software engineering. Autom Softw Eng 2008;15(3–4):379–91.
[68] Selic BV, Gaševic D, Djuric D, Bézivin J, Devedžic V.: Model driven engineering and ontology development, Springer; 2009.
[69] Silva AR, Saraiva J, Ferreira D, Silva R, Videira C. Integration of RE and MDE Paradigms: the projectIT approach and tools. IET Softw J 2007;1(6):217–314.
[70] Silva AR, Saraiva J, Silva R, Martins C. XIS-UML Profile for eXtreme Modeling Interactive Systems. MOMPES, IEEE Computer Society; 2007b.
[71] Slonneger K, Kurtz, Barry L. Formal syntax and semantics of programming languages. Addison-Wesley; 1995.
[72] Sousa L, Silva AR. Preliminary design and implementation of DSL3S – a domain specific language for spatial simulation scenarios. In: Proceedings of the International symposium on cellular automata modeling for urban and spatial systems (CAMUSS); 2012.
[73] Sprinkle J, Rumpe B, Vangheluwe, H, Karsai G. 3Metamodelling – state of the art and research challenges. In: Model-based eng embedded real-time syst, Springer, 2010. p. 57–76.
[74] Stahl T, Volter M. Model-driven software development. Wiley; 2005.
[75] Steinberg D, Budinsky F, Paternostro M, Merks E, Eclipse EMF. Modeling framework. 2nd editionAddison-Wesley; 2009.
[76] Syriani E, Gray J, Vangheluwe H. Modeling a model transformation language. Domain Eng 2013:211–37.
[77] Syriani E, Vangheluwe H, Mannadiar R, Hansen C, Van Mierlo S, Ergin H. AToMPM: a web-based Modeling Environment. Demos/Posters/StudentResearch @ MoDELS; 2013b.
[78] Sztipanovits J, Karsai G. Model-integrated computing. IEEE Comput 1997;30(4):110–1.
[79] Tolvanen J-P, Rossi M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. OOPSLA'2003, ACM; 2003.
[80] Utting M, Legeard B. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers; 2007.
[81] Vangheluwe H, De Lara J, Mosterman, PJ. An introduction to multi-paradigm modelling and simulation. In: Proceedings of the AIS'2002 conference; 2002.
[82] Viyovic V, Maksimovic M, Perisic B. Sirius: a rapid development of DSM graphical editor. In: International conference on intelligent engineering systems, IEEE; 2014.
[83] Voelter M, Benz S, Dietrich C, Engelmann B, Helander M, Kats LC, et al.: DSL engineering: designing, implementing and using domain-specific languages, dslbook.org; 2013.
[84] Weaver P, Lambrou N, Walkley M. Practical SSADM Version 4+. 2nd editionPrentice Hall; 1998.
[85] Yourdon E. Modern structured analysis. Prentice Hall; 1999.