Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 307391, 17 pages
doi:10.1155/2010/307391

Research Article

A Quality Model for Conceptual Models of MDD Environments

Beatriz Marín,[1] Giovanni Giachetti,[1] Oscar Pastor,[1] and Alain Abran[2]

[1] *Centro de Investigación en Métodos de Producción de Software, Universidad Politécnica Valencia, Camino de Vera s/n, 46022 Valencia, Spain*
[2] *Department of Software Engineering & Information Technology, École de Technologie Supérieure, Université du Québec, 1100 Notre-Dame Ouest, Montréal QC, Canada H3C 1K3*

Correspondence should be addressed to Beatriz Marín, bmarin@pros.upv.es

Received 15 February 2010; Accepted 9 June 2010

Academic Editor: Giovanni Cantone

In Model-Driven Development (MDD) processes, models are key artifacts that are used as input for code generation. Therefore, it is very important to evaluate the quality of these input models in order to obtain high-quality software products. The detection of defects is a promising technique to evaluate software quality, which is emerging as a suitable alternative for MDD processes. The detection of defects in conceptual models is usually manually performed. However, since current MDD standards and technologies allow both the specification of metamodels to represent conceptual models and the implementation of model transformations to automate the generation of final software products, it is possible to automate defect detection from the defined conceptual models. This paper presents a quality model that not only encapsulates defect types that are related to conceptual models but also takes advantage of current standards in order to automate defect detection in MDD environments.

1. Introduction

Historically, software production methods and tools have a unique goal: *to produce high-quality software*. Since the goal of Model-Driven Development (MDD) methods is no different, MDD methods have emerged to take advantage of the benefits of using of models [1] to produce high-quality software. Model-Driven Technologies [2] attempt to separate business logic from platform technology in order to allow automatic generation of software through well-defined model transformations. In a software production process based on MDD technology, the conceptual models are key inputs in the process of code generation. Thus, the conceptual models must provide a holistic view of all the components of the final application (including the structure of the system, the behavior, the interaction between the users and the system, and the interaction among the components of the system) in order to be able to automatically generate the final application. Therefore, the evaluation of the quality of conceptual models is essential since this directly affects the quality of the generated software products.

To evaluate the quality of conceptual models, many proposals have been developed following different perspectives [3]. There are proposals that are based on theory [4], experience [5], the observation of defects in the conceptual models in order to induce quality characteristics [6], the evaluation of the quality characteristics defined in the ISO 9126 standard [7] in conceptual models by means of measures [8], a synthesis approach [9], and so forth. Taking into account the advantages and disadvantages of each development perspective for quality frameworks [3], *defect detection* is considered as a suitable approach because it provides a high level of empirical validity provided by the variety of conceptual models that are observed. However, it is interesting to note that this approach is not broadly used in the software engineering discipline, even though defect detection is the most common quality evaluation approach used by other disciplines such as health care [10].

To develop an effective quality assurance technique, it is necessary to know what kind of defects may occur in conceptual models related to MDD approaches. Currently, there are some approaches that detect defects in conceptual models (such as [11, 12]), which are mostly focused on

the detection of defects that come from either the data perspective (data models) or the process perspective (process models). However, defect detection has not been clearly accomplished from the interaction perspective (interaction models) even though all of these perspectives (data, process, and interaction modeling) are essential to specify a correct conceptual schema used in an MDD context [13].

In this article, we present an approach that allows the automatic verification of the conceptual models used in MDD environments with respect to defect types from the data, process, and interaction perspectives. We present a set of defect types related to data, process, and interaction models as well as a quality model that formalizes the elements involved in the identification of the different defect types proposed. This quality model, which is defined using current metamodeling standards, is the main contribution of this article. It is oriented to represent the abstract syntax of the constructs involved in the conceptual specification of software applications generated in MDD environments. From this quality model, defect detection can be automated by means of OCL [14] rules that verify the properties of the conceptual constructs according to the set of defects types defined.

The remainder of the paper is organized as follows. Section 2 presents the related work, including a list with the defect types found in the literature. Section 3 presents a set of conceptual constructs of the conceptual model of an MDD approach. Section 4 presents the quality model (metamodel) in detail. Section 5 presents our conclusions and suggestions for further work.

2. Related Work

In the literature, there is no consensus for the definition of the quality of conceptual models. There are several proposals that use different terminologies to refer to the same concepts. There are also many proposals that do not even define what they mean by quality of conceptual models. In order to achieve consensus about the definition of quality and to improve the conceptual models, we have adopted the definition proposed by Moody [3]. This definition is based on the definition of quality of a product or service in the ISO 9000 standard [15]. Therefore, we define the quality of a conceptual model as "*the total of features and characteristics of a conceptual model that bear on its ability to satisfy stated or implied needs*".

In order to design a quality model, the types of defects that the conceptual models used in MDD environments must be known. Defect detection refers to identifying anomalies in software products in order to correct them to be able to obtain better software products. The IEEE 1044 [16] presents a standard classification for software anomalies, which defines an anomaly as *any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, and so forth, or from someone's perceptions or experiences*. This definition is so broad that different persons can identify different anomalies in the same software artifact, and anomalies that one person identifies may not be perceived as anomalies for another person. Therefore, many researchers have had to redefine the concepts of error, defect, failure, fault, and so forth, while other researchers have used these concepts indistinctly [17]. To avoid the proliferation of concepts related to the software anomalies, in this article, we analyze the proposals of defect detection in conceptual models by adopting the terminology defined by Meyer in [18].

(i) *Error.* It is a wrong decision made during the development of a conceptual model.

(ii) *Defect.* It is a property of a conceptual model that may cause the model to depart from its intended behavior.

(iii) *Fault.* It is an event of a software system that departs from its intended behavior during one of its executions.

Taking into account that the cost of fault correction increases exponentially over the development life cycle [3], it is of paramount importance to discover faults as early as possible: this means detecting errors or defects before the implementation of the software system.

Travassos et al. [19] use reading techniques to perform software inspections in high-level, object-oriented designs. They use UML diagrams that are focused on data structure and behaviour. These authors advocate that the design artifacts (a set of well-related diagrams) should be read in order to determine whether they are consistent with each other and whether they adequately capture the requirements. Design defects occur when these conditions are not met. These authors use a defect taxonomy that is borrowed from requirements defects [20], which classifies the defects as Omission, Incorrect Fact, Inconsistency, Ambiguity, and Extraneous Information. However, they do not present the types of defects that were found in their study.

Laitenberger et al. [21] present a controlled experiment to compare the checklist-based reading (CBR) technique with the perspective-based reading (PBR) technique for defect detection in UML models. The authors define three inspection scenarios in the UML models in order to detect defects from different viewpoints (designers, testers, and programmers). These authors do not explicitly identify the types of defects found in UML models. However, they present a set of concepts that must be checked in the UML models from different viewpoints, and it is possible to infer the types of defects from these concepts.

Conradi et al. [22] present a controlled experiment that was designed to perform a comparison between an old reading technique used by the Ericsson company and an Object-Oriented Reading Technique (OORT) for detecting defects in UML models. The authors present a summary of the defects detected using both inspection techniques for the same project. The findings of the controlled experiment are that one group of subjects detected 25 defects using the old technique (without any overlaps of the defects detected) while the other group of subjects detected 39 defects using the OORT technique (with 8 overlaps in the defects detected). However, the authors did not present the types of defects found in the models inspected in the experiment.

Gomma and Wijesekera [23] present an approach for the identification and correction of inconsistency and incompleteness across UML views. It is applied in the COMET method [24], which uses the UML notation. The authors present 7 defect types related to the consistency between models: 1 defect type for the consistency between use-case diagrams and sequence diagrams, 4 defect types for the consistency between class diagrams and state transition diagrams, and 2 defect types for the consistency between sequence diagrams and state transition diagrams.

Kuzniarz [25] presents a set of inconsistencies found in student designs produced in a sample didactic development process. This proposal corresponds to a case study that was developed to explore the nature of inconsistency in UML designs. The authors present 8 defect types based on subjective but common-sense judgment.

Berenbach [26] presents a set of heuristics for analysis and design models that prevents the introduction of defects in the models. This allows semantically correct models to be developed. In addition, Berenbach presents the *Design Advisor* tool created by Siemens to facilitate the inspection of large models. This tool implements the heuristics proposed by Berenbach for evaluating the goodness of the analysis and design models. For the analysis models, Berenbach presents 10 heuristics for model organization, 5 heuristics for use case definition, 3 heuristics related to the use-case relationships, and 14 heuristics related to business object modeling. For the design models, he presents 2 heuristics for the class model.

Lange and Chaudron [27] identify the incompleteness of UML diagrams as a potential problem in the subsequent stages of a model-oriented software development. These authors refer to the completeness of a model by means of the aggregation of three characteristics: (1) the well-formedness of each diagram that comprises the model, (2) the consistency between the diagrams that comprise the model, and (3) the completeness among the diagrams that comprise the model. Note that the authors use the completeness concept to define the completeness of a model. Since this is equivalent to reusing the same concept (completeness) for its own definition, they do not really describe what is understood by completeness. These authors identify eleven types of defects of UML models: 5 types of defects related to the well-formedness of the diagrams, 3 types related to the consistency among the diagrams, and 3 other types related to the completeness among the diagrams.

Leung and Bolloju [28] present a study that aims to understand the defects frequently committed by novice analysts in the development of UML Class models. These authors use Lindland et al.'s quality framework [4] to evaluate the quality of the class diagrams. They distinguish five classifications that allow the evaluation of the syntactic quality, semantic quality, and pragmatic quality. These five classifications are *syntactic* (for the syntactic quality), *validity* and *completeness* (for the semantic quality), and *the expected is missing* and *the unexpected is present* (for the pragmatic quality). The authors obtain 103 different types of defects in 14 projects. However, the authors only detail 21 types of defects for one class diagram.

Bellur and Vallieswaran [12] perform an impact analysis of UML design models. This analysis evaluates the consistency of the design and the impact of a design change over the code. In order to evaluate the consistency of the design models, these authors propose evaluating the well-formedness of UML diagrams. The proposal of Bellur and Vallieswaran [12] extends the proposal of Lange and Chaudron [27] focusing on the quality of UML conceptual models as well as on the code. These authors identify 4 types of defects for use-case diagrams, 2 types of defects for sequence diagrams, 5 types of defects for the specification of the method sequences, 3 types of defects for the class diagram, 8 types of defects for the state transition diagrams, 2 types of defects for the component diagram, and 2 types of defects for the deployment diagram.

Summarizing, the above proposals present defect types that are related to the consistency (consistency is defined in the IEEE 610 standard as the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component) [29] among diagrams and defect types that are related to the correctness (correctness is defined in the IEEE 610 standard as the degree to which a system or component is free from faults in its specification, design, and implementation) [29] of a particular diagram. Table 1 presents the defect types of the different proposals analyzed. The first column, "Quality Characteristic", divides the defect types into two groups, the consistency characteristic and the correctness characteristic. The second column, "Authors", presents the authors who have proposed these defect types and the corresponding year of the proposals. The third column, "Model", presents the conceptual models (or diagrams) where the defect types have been found. The last column, "Defect Types", presents the defect types.

In the systematic revision of the state of the art, we noticed that all the proposals for defect detection in conceptual models are focused on UML models. However, it is well known that UML diagrams [30] do not have enough semantic precision to allow the specification of a complete and unambiguous software application [31–33], which is clearly observed in the semantic extension points that are defined in the UML specification [30]. For this reason, many methodologies have selected a subset of UML diagrams and conceptual constructs and have aggregated the needed semantic expressiveness in order to completely specify the final applications in the conceptual model, making the implementation of MDD technology a reality.

Since MDD proposals select a set of conceptual constructs and aggregate others to specify the conceptual models, it is important to note that a great number of conceptual constructs increase the complexity of the specification of the models and may cause the introduction of more defects into the conceptual models. For this reason, the conceptual constructs of an MDD proposal must be carefully selected so that the number of constructs that allow the complete specification of software applications at the conceptual abstraction level is as low as possible. In the following section, we present a minimal set of the conceptual constructs for an MDD environment.

Table 1: Defect types of conceptual models in the state of the art.

Quality characteristic	Authors	Model	Defect types
Consistency among diagrams	Laitenberger et al. [21]	UML Class	(i) A class in the design class diagram is not a class in the system class diagram (with the same name) (ii) The number, types, and names of the attributes of a class in the design class diagram are not the same in the class of the system class diagram (iii) The number, names, and arguments of the methods of a class in the design class diagram are not the same in the class of the system class diagram (iv) The associations with their cardinality and arity in the design class diagram are not the same in the system class diagram (v) The constraints between classes of the design class diagram are not the same constraints for these classes in the system class diagram
		UML Collaboration	(i) An object that does not correspond to a class of the class diagram (ii) The collaboration diagram has messages that do not correspond to the system operation (iii) The messages do not have the same number and type of arguments as the operations of the system described in the Operation model
		Operation model of the Fusion method	(i) Operations (read, change, send, and result) that do not have the corresponding message in the collaboration diagram
	Gomma and Wijesekera [23]	UML Use-case	(i) A use case that does not correspond to at least one scenario described by an interaction diagram
		COMET state transition	(i) Each Statechart that does not correspond to a state that is dependent on the control class in a class diagram (ii) The values of the current states, events, actions, and activities that appear on a Statechart that are not declared as attribute values of the respective state, event, action, and activity attributes for the state that is dependent on the control class (iii) An event on a Statechart that does not correspond to a method of the state of the control class in the class diagram (iv) Variables used to define conditions in any Statechart that are not attributes of the state that is dependent on the control class in the corresponding class diagram (v) Each event on a Statechart that corresponds to an incoming message on the state that is dependent on the control object, which is not represented in an interaction diagram (which executes the Statechart) (vi) Each action on a Statechart that corresponds to an outgoing message on the state that is dependent on the control object, which is not represented in an interaction diagram (which executes the Statechart)
	Kuzniarz [25]	UML Use-case	(i) The actor that is defined in the use case diagram is not the same actor that takes part in the interaction that is defined in the corresponding sequence diagram (ii) Not all the steps that are defined in the use case description correspond to messages in the system sequence diagram (iii) There are extension points for the extension of use cases that are missing (not represented) in the diagram of the controller use case
		UML Sequence	(i) An iteration symbol that is related to an iterative task is missing in the sequence diagram

Table 1: Continued.

Quality characteristic	Authors	Model	Defect types
			(ii) There are links used in sequence diagrams that are not associations in the class diagram (iii) There are sequences of messages in the sequence diagram that are not acceptable for the controller use case (iv) There are elements used in pre- and postconditions of the contracts that are not defined in the class model (vi) There are sequences of messages in the sequence diagram that are not an acceptable subsequence for the state machines that take part in the sequence diagram
	Berenbach [26]	UML Use-case	(i) Actors in use-case diagrams that are not specified in the context diagram (ii) Use case without an interaction diagram that shows the possible scenarios
		UML Class	(i) An interface in the class diagram that is not used to communicate with a concrete use case (ii) A class that is not instantiated in any process of the system (sequence and collaboration diagrams) (iii) Methods of the interface class that are not represented in the process of the system (sequence and collaboration diagrams) (iv) Classes in the class diagram that are not specified in the use-case diagram (v) Interfaces in the class diagram that are not specified in the use-case diagram
	Lange and Chaudron [27]	UML Sequence	(i) Messages between unrelated classes
		UML Class	(i) Classes that are not called in the sequence diagram (ii) Interfaces that are not called in the sequence diagram (iii) Methods that are not called in the sequence diagram
	Lange and Chaudron [11]	UML Use-case	(i) Use cases without sequence diagrams
		UML Sequence	(i) Objects of the sequence diagram that are not related to a class in the class diagram
	Bellur and Vallieswaran [12]	UML Use-case	(i) A use case that does not reference a use-case sequence diagram
		UML Sequence	(i) A variable of a general class used in the sequence diagram that is null or is not a valid class in the class diagram (ii) A method referenced in the sequence diagram that is null or is not a valid method in the method sequence charts (iii) An object that is not the sender or the receiver in any interaction (iv) An object that does not reference a valid class and state diagram (v) A message that is not an instance of one class method for some class defined in the system
		UML State Transition	(i) A state diagram that is not related to one and only one class (ii) A state that is not described by one or more attributes of the class (iii) State change events that do not correspond to messages in the method sequence diagrams
		UML Component	(i) An intercomponent relationship that does not have 2 terminating end classes which are valid classes in the class diagram (ii) A component in the component diagram that is not mapped to a physical system described in the deployment diagram

Table 1: Continued.

Quality characteristic	Authors	Model	Defect types
		UML Deployment	(i) A deployment diagram that is not related to one or more component diagrams
Correctness	Laitenberger et al. [21]	UML Class	(i) The types of the attributes of a class are not specified (ii) The methods of a class are not specified (iii) Parameters that do not have a type associated
	Berenbach [26]	UML Use-case	(i) Multiple entry point for the system in the use-case diagram (ii) Diagrams without a description and status (iii) Concrete use cases without a definition (iv) Abstract use cases that are not realized by a concrete use-case (v) Extends use-case relationship that is specified between use cases that are not concrete (vi) A concrete use case that includes an abstract use case
		UML Business	(i) Services of business objects that do not have defined pre- and postconditions
		UML Class	(i) An interface class with private methods
	Lange and Chaudron [27]	UML Sequence	(i) Objects without a name (ii) Abstract classes in sequence diagrams (iii) Messages without a name (iv) Messages without a method
		UML Class	(i) Classes without methods (ii) Interfaces without methods (iii) Classes with public attributes
	Leung and Bolloju [28]	UML Class	(i) Missing association label or cardinality detail (ii) Improper label for a class, an association, an attribute, or an operation (iii) Improper notation for an association, an aggregation, or a generalization (iv) The nonimplicit operations that are present in sequence diagram are not included (v) Implicit operation is listed (vi) Wrong association cardinality (reversed or wrong range) (vii) Wrong location of an attribute or an operation (viii) Wrong association grouping (ix) Missing class, attribute, operation, or association (x) Incomplete class description (xi) Operation that cannot be realized (using attributes and relationships) (xii) Does not use domain-specific terminology (xiii) Poor layout of the class diagram (xiv) Insufficient distinction among sub-classes (xv) Operation naming is improper or ambiguous (xvi) Associations are replicated at sub-classes (xvii) Manual operation is represented as association (xviii) Excessive use of generalization, PK concept, FK concept, or emphasis on statistical information (xix) Redundant attributes (xx) Redundant associations (xxi) Implementation detail is present in the diagram

Table 1: Continued.

Quality characteristic	Authors	Model	Defect types
	Lange and Chaudron [11]	UML Sequence	(i) Message in wrong direction in the sequence diagram
		UML Class	(i) Multiple definitions of classes with equal names
	Bellur and Vallieswaran [12]	UML Use-case	(i) An actor that does not use one or more use cases (ii) A use case that is not used by one or more actors (iii) A use case that does not belong to system
		UML Sequence	(i) A message that does not have a sender and a receiver object (ii) A message that does not conform to the signature of the method corresponding to the message
		UML Class	(i) A class diagram without classes (ii) An association without a source and target class (iii) A class that does not have at least one attribute or method
		UML State Transition	(i) A state diagram without one start state and one end state (ii) A state that has overlap of attribute values describing the state (iii) A state that is not reachable from the start state (iv) A state that cannot reach the end state
		UML Component	(i) A component diagram without components

3. Conceptual Constructs of an MDD Proposal

Model-Driven Development environments generally use Domain-Specific Modeling Languages (DSMLs), which define the set of conceptual constructs in order to represent the semantics of a particular domain in a precise way [34]. For example, the DSMLs for the Management Information Systems (MIS) domain share a well-known set of conceptual constructs. In order to allow the complete specification of MIS applications, the DSMLs must add specific conceptual constructs to the conceptual models.

The MDD methods use models at different levels of abstraction to automate their transformations to generate software products. Model-Driven Architecture (MDA) is a standard proposed by OMG [35] that defends MDD principles and proposes a technological architecture to construct MDD methods. This architecture divides the models into the following categories: Computation-Independent Models (CIMs), Platform-Independent Models (PIMs), and Platform-Specific Models (PSMs). CIMs are requirement models (e.g., use-case diagrams, i* models, etc.), which by nature do not allow the complete specification of final software applications. In contrast, PIMs and PSMs allow the complete specification of final applications in an abstract way, but the PSMs use constructs that are specific to the technological platform in which the final applications will be generated (e.g., java, c#, etc.). Therefore, in this article, we focus on the PIM models (which we refer to as conceptual models of MDD proposals) since they can be used independently of the platform.

To provide details of the conceptual constructs of MDD proposals for the MIS domain, we have selected a specific MDD environment as reference: the OO-Method approach [36, 37]. This approach is an object-oriented method that puts MDD technology into practice [36] by separating the business logic from the platform technology to allow the automatic generation of final applications by means of well-defined model transformations [37]. The OO-Method approach provides the semantic formalization that is needed to define complete conceptual models, which allows the specification of all the functionality of the final application at the conceptual level. The OO-Method approach has been implemented in an industrial tool [26] that automatically generates fully working applications. These applications can be either desktop or web MIS applications and can be generated in several technologies (java, C#, visual basic, etc.).

The conceptual model of an MDD proposal must be able to specify the structure, the behavior, and the interaction of the components of an MIS in an abstract way. Therefore, we distinguish three kinds of models that together provide the complete specification of software systems: a structural model, a behavior model, and an interaction model.

3.1. The Structural Model. The structural model describes the static part of the system and is generally represented by means of a class model. A class describes a set of objects that share the same specifications of characteristics, constraints, and semantics. A class can have attributes, integrity constraints, services, and relationships with other classes.

The attributes of a class represent characteristics of this class. The attributes of a class can also be derived attributes, which obtain their value from the values of other attributes or constants. The integrity constraints are expressions of a semantic condition that must be preserved in every valid state of an object.

Table 2: Defect types of conceptual models found using OOmCFP.

Defect types found using OOmCFP
Defect: An object model without a specification of an agent class
Defect: An OO-Method Conceptual Model without a definition of the presentation model
Defect: A presentation model without the specification of one or more interaction units
Defect: An object model without the specifications of one or more classes
Defect: A class without a name
Defect: Classes with a repeated name
Defect: A class without the definition of one or more attributes
Defect: A class with attributes with repeated names
Defect: An instance interaction unit without a display pattern
Defect: A population interaction unit without a display pattern
Defect: A display pattern without attributes
Defect: Derived attributes without a derivation formula
Defect: A filer without a filter formula
Defect: An event of a class of the object diagram without valuations (excluding creation or destruction events)
Defect: A class without a creation event
Defect: Transactions without a specification of a sequence of services (service formula)
Defect: Operations without a specification of a sequence of services (service formula)
Defect: A service without arguments
Defect: A service with arguments with repeated names
Defect: A precondition without the specification of the precondition formula
Defect: A precondition without an error message
Defect: An integrity constraint without the specification of the integrity formula
Defect: An integrity constraint without an error message

The services of a class are basic components that are associated with the specification of the behavior of a class. The services can be events, transactions, or operations. The events are indivisible atomic services that can assign a value to an attribute. The transactions are a sequence of events or other transactions that have two ways to end the execution: either all involved services are correctly executed or none of the services are executed. The operations are a sequence of events, transactions, or other operations, which are executed sequentially independently of whether or not the involved services have been executed correctly. The services can have preconditions that limit their execution because the preconditions are conditions that must be true for the execution of a service.

The relationships between classes in the structural model can be the following: agent, association, aggregation, composition, and specialization. Agents are relationships that indicate the classes that can access specific attributes or services of other classes of the model. Agents are specifically defined in the reference MDD approach.

3.2. The Behavior Model. The behavior model describes the dynamic part of a system. These dynamics include the behavior of each class and the interaction among the objects of the system. For the specification of the behavior of a system, we select the functional model of the reference MDD approach, which defines the behavior of the services defined inside the classes of the structural model.

The functional model specifies a formula with the sequence of events, transactions, or operations that must be executed when a service is used. This formula must be specified by means of well-formed, first-order logic formulae that are defined using the OASIS language [38].

The functional model specifies the effects that the execution of an event has over the value of the attributes of the class that owns the event. To do this, the functional model uses valuations to assign values to the corresponding attributes. The effect of a valuation is also specified using formulae within the syntax of the OASIS language. The change that a valuation produces in the value of an attribute is classified into three different categories: *state*, *cardinal*, and *situation*. The *state* category implies that the change of the value of an attribute depends only on the effect specified in the valuation for the event, and it does not depend on the value in the previous state. The *cardinal* category increases, decreases, or initializes the numeric-type attributes. The *situation* category implies that the valuation effect is applied only if the value of the attribute is equal to a predefined value specified as the current value of the attribute.

Since services can have preconditions, the conditions and the error messages of the preconditions are also specified using OASIS formulae. The integrity constraints of a class are also specified using OASIS formulae.

3.3. The Interaction Model. The interaction model describes the presentation (static aspects of a user interface like widgets, layout, contents, etc.) and the dialogs (dynamic

Table 3: 23 Defect types of conceptual models found using OOmCFP and OCL rules.

Defect types found using OOmCFP	OCL rules
Defect: An object model without a specification of an agent class	**context** Agent **inv**: **body** self.allInstances->size()>0
Defect: An OO-Method Conceptual Model without a definition of the presentation model	**context** ConceptualModel **inv**: **body** self.presentation->size()>0
Defect: A presentation model without the specification of one or more interaction units	**context** PresentationModel **inv**: **body** self.interactionUnit->size()>0
Defect: An object model without the specifications of one or more classes	**context** StructuralModel **inv**: **body** self.ownedClass->size()>0
Defect: A class without a name	**context** Class **inv**: **body** Class.allInstances()->select(c | c.name.isEmpty())->isEmpty()
Defect: Classes with a repeated name	**context** Class **inv**: **body** self.allInstances()->forAll(c1, c2 | c1 < > c2 **implies** c1.name < > c2.name)
Defect: A class without the definition of one or more attributes	**context** Class **inv**: **body** self.features->select(t | t.oclIsKindOf(TypedProperty))- >collect(t | t.oclAsType (TypedProperty))->size()>0
Defect: A class with attributes with repeated names	**context** Class **inv**: **body** self.features->select(t | t.oclIsKindOf(TypedProperty))- >collect(t | t.oclAsType (TypedProperty))->forAll(a1, a2 | a1 < > a2 **implies a**1.name < > a2.name)
Defect: An instance interaction unit without a display pattern	**context** InstanceIU **inv**: **body** self.displaySet->size()>0
Defect: A population interaction unit without a display pattern	**context** PopulationIU **inv**: **body** self.displaySet->size()>0
Defect: A display pattern without attributes	**context** DisplaySet **inv**: **body** self.relatedAttribute->size()>0
Defect: Derived attributes without a derivation formula	**context** DerivedAttribute **inv**: **body** self.derValue.effect->select(f | f.value.isEmpty())->isEmpty()
Defect: A filter without a filter formula	**context** Filter **inv**: **body** self.filterFormula->select(f | f.value.isEmpty())->isEmpty()
Defect: An event of a class of the object diagram without valuations (excluding creation or destruction events)	**context** Event **inv**: **body** self.allInstances->select(e | (e.kind < > ServiceKind::creation and e.kind < > ServiceKind::destruction) implies e.valuation.size()> 0)
Defect: A class without a creation event	**context** Class **inv**: **body** self.features->select(s | s.oclIsKindOf(Service))- >collect(s | s.oclAsType (Service))-> select(s | s.kind = ServiceKind::creation)->notEmpty()
Defect: Transactions without a specification of a sequence of services (service formula)	**context** Transaction **inv**: **body** self.effect->select(f | f.value.isEmpty())->isEmpty()
Defect: Operations without a specification of a sequence of services (service formula)	**context** Operarion **inv**: **body** self.effect->select(f | f.value.isEmpty())->isEmpty()
Defect: A service without arguments	**context** Service **inv**: **body** self.argument->size()>0
Defect: A service with arguments with repeated names	**context** Service **inv**: **body** self.argument->forAll(a1, a2 | a1 < > a2 **implies** a1.name < > a2.name)
Defect: A precondition without the specification of the precondition formula	**context** Service **inv**: **body** self.precondition.effect->select(f | f.value.isEmpty())->isEmpty()
Defect: A precondition without an error message	**context** Service **inv**: **body** self.precondition->select (c | c.errorMsg.isEmpty())->isEmpty()

Table 3: Continued.

Defect types found using OOmCFP	OCL rules
Defect: An integrity constraint without the specification of the integrity formula	**context** Constraint **inv**: **body** self.effect->select(f | f.value.isEmpty())->isEmpty()
Defect: An integrity constraint without an error message	**context** Constraint **inv**: **body** self.allInstances->select (c | c.errorMsg.isEmpty())->isEmpty()

Table 4: 5 Defect types of conceptual models found in the literature and OCL rules.

Defects Types found in the literature	OCL Rules
Defect: An attribute of a class without the specification of the type	**context** Class **inv**: **body** self.features->select(t | t.oclIsKindOf(TypedProperty))- >collect(t | t.oclAsType (TypedProperty))->select(a | a.type.isEmpty())->isEmpty()
Defect: An argument of a service without the specification of the type	**context** Service **inv**: **body** self.features->select(s | s.oclIsKindOf(Service))->collect(s | s.oclAsType (Service)).argument.type->size<1->isEmpty()
Defect: Associations replicated at sub-classes	**Classifier**::parents(): Set(Classifier); parents = generalization.general **Classifier**::allParents(): Set(Classifier); allParents = self.parents()->union(self.parents()->collect(p | p.allParents()) **context** AssociationEnd **inv**: **body** self.allInstances->forAll(r1, r2 | r1.name = r2.name **and** r1.owningClass.allParents()->select(c | c.name = r2.name)->isEmpty()
Defect: Associations with a repeated name	**Context** Relationship **inv**: **body** self.allInstances()->forAll(r1, r2 | r1 < > r2 **implies** r1.name < > r2.name)
Defect: An association without a source and target class	**context** Association **inv**: **body** self.role->select(e1,e2 | e1.role.kind = EndKind::source **and** e2.role.kind = EndKind::target)

aspects of a user interface like controls, dynamic page change, etc.) of the system [39]. In order to specify the interaction between the users of an application and the system, the OO-Method MDD approach specifies views. A view corresponds to a set of interfaces, which are the communication point between agents and classes of the structural model. When the views of a system have been defined, the interaction model of each view must be specified.

The interaction model allows the specification of the graphical user interface of an application in an abstract way [40]. To do this, the interaction model has a set of abstract presentation patterns that are organized hierarchically in three levels: access structure, interaction units, and auxiliary patterns. The first level allows the specification of the system access structure. In this level, the set of entry options that each user of the application will have available is specified by means of a *Hierarchy Action Tree* (HAT).

Based on the menu-like view provided by the first level, the second level allows the specification of the interaction units of the system. The interaction units are groups of functionality that allow the users of the application to interact with the system. Thus, the interaction units of the interaction model represent entry-points for the application. These units can be the following.

(i) A *Service Interaction Unit* (SIU). This interaction unit represents the interaction between a user of the application and the execution of a system service. In other words, the SIUs allow the users of the application to enter the values for the arguments of a service and to execute the service. They also provide the users with the feedback of the results of the execution of the service.

(ii) A *Population Interaction Unit* (PIU). This interaction unit represents the interaction with the system that deals with the presentation of a set of instances of a class. In a PIU, an instance can be selected, and the corresponding set of actions and/or navigations for the selected instance are offered to the user.

(iii) An *Instance Interaction Unit* (IIU). This interaction unit represents the interaction with an object of the system. In an IIU, the corresponding set of actions and/or navigations for the instance are offered to the user.

(iv) A *Master Detail Interaction Unit* (MDIU). This interaction unit represents the interaction with the system through a composite interaction unit. An MDIU corresponds to the joining of a master interaction unit (which can be an IIU or a PIU) with a detail interaction unit (which can be a set of IIUs, PIUs, or SIUs).

The third level of the interaction model allows the specification of the auxiliary patterns that characterize lower level details about the behavior of the interaction units. These auxiliary patterns are the following.

(i) The *entry* pattern is used to indicate that the user can enter values for the arguments of the SIUs.

(ii) The *defined selection* pattern is used to specify a list of specific values to be selected by the user.

(iii) The *arguments dependency* pattern is used to define dependencies among the values of the arguments of a service. To do this, Event-Condition-Action (ECA) rules are defined for each argument of the service. The ECA rules have the following semantics: when an interface event occurs in an argument of a service (i.e., the user enters a value), an action is performed if a given condition is satisfied.

(iv) The *display set* pattern is used to specify which attributes of a class or its related classes will be shown to the user in a PIU or an IIU.

(v) The *order criteria* pattern allows the objects of a PIU to be ordered. This pattern consists of the ascendant/descendant order over the values of the attributes of the objects presented in the PIU.

(vi) The *action* pattern allows the execution of services by joining and activating the corresponding SIUs by means of actions.

(vii) The *navigation* pattern allows the navigation from an interaction unit to another interaction unit.

(viii) The *filter pattern* allows a restricted search of objects for a population interaction unit. A filter can have data-valued variables and object-valued variables. These variables can have a defined default value, an associated PIU to select the value of the object-valued variables, and precharge capabilities for the values of the object-valued arguments.

Each auxiliary pattern has its own scope that states the context in which it can be applied. With these conceptual constructs we can completely specify applications that correspond to the MIS domain in an abstract way. To formalize the concepts and the relationships among them, we present a generic metamodel for MDD proposals.

4. A Quality Model for MDD Environments

The quality model proposed in this article is specified by means of the modeling facilities that current metamodeling standards provide, specifically, the EMOF specification [41]. The EMOF was selected for its metamodeling language, which is supported by open-source tools such as the Eclipse EMF [42] (for metamodeling purposes) or Eclipse GMF [43] (for the generation of model editors). EMOF is also used by technologies such as ATL [44, 45] or QVT [46] for the implementation of model-to-model transformations. In the following subsections, we present the metamodel, the procedure proposed for defect detection, and a list of defect types found in MDD-oriented conceptual models.

4.1. A Metamodel for Defect Detection in MDD-Oriented Conceptual Models. In general terms, a metamodel is the artifact used to specify the abstract syntax of a modeling language: the structural definition of the involved conceptual constructs with their properties, the definition of relationships among the different constructs, and the definition of a set of rules to control the interaction among the different constructs specified [47]. In EMOF, a metamodel is represented by means of a class diagram, where each class of the diagram corresponds to a construct of the modeling language involved. We use the OCL specification [14] for the definition of the controlling rules of the metamodel since it is part of the OMG standards and can work with EMOF metamodels. Also, OCL rules provide a computable language for rule specification, which allows the defined rules to be automatically evaluated by existent tools such as Eclipse OCL tools [48].

With EMOF, the quality metamodel can specify the constructs involved in the different types of defects as well as the properties that must be present in the different conceptual constructs for the detection of defects. The OCL language used for the metamodeling rules can be used to define specific rules to automate defect detection. The final quality model is comprised of two main elements: (1) a metamodel for the description of the conceptual constructs that are used in MDD environments (which includes all the properties involved in defect detection) and (2) a set of OCL rules that allows the automatic detection of defects according to the list of defects presented in this article.

Figure 1 presents our quality metamodel. As this figure shows, a generic *ConceptualModel* of an MDD approach is comprised of a structural model (class *StructuralModel*), a behaviour model (class *BehaviourModel*), and an interaction model (*PresentationModel*). The structural model has a set of classes (class *Class*). Each class has several features (class *ClassFeature*), which can be services (class *Service*) or properties (class *Property*). In turn, the properties can be typed properties (class *TypedProperty*) or association ends (class *AssociationEnd*). The typed properties correspond to the attributes of a class, which must have a data type (class *DataType*) specified (attribute *Kind*). The typed properties can be derived (class *DerivedAttribute*) or not derived (class *Attribute*). The services can be events (class *Event*), transactions (class *Transaction*), or operations (class *Operation*). The events have valuations (class *Valuation*) to change the value of the attributes of a class. Each service has a set of arguments (class *Argument*) with their corresponding types (class *Type*), and it can also have a set of preconditions (association *precondition*). There are relationships between the classes of the model (represented by the class *RelationShip*), which can be associations (class *Association*), generalizations (class *Generalization*), and agents (class *Agent*). The agent definition is oriented to state the visibility and execution permissions over the classes of the defined model (association *agent*). The associations can be aggregations, compositions, or normal associations (attribute *aggregation* of the class *AssociationEnd*). Each class has a set of integrity constraints (association *integrityConstraint*). The classes, class features,

arguments, and relationships must have a name (class *NamedElement*).

The derived attributes, services, preconditions, and integrity constraints require the specification of the functionality that they perform. This functionality is specified by means of the behaviour model. The behaviour model has elements (class *BehaviourElement*) that can be conditional elements (*ConditionalBehaviourElement*) or constraint elements (*Constraint*). The conditional elements correspond to formulae (class *Formula*) with a condition (association *condition*) and an effect (association *effect*). The constraint elements correspond to formulae (class *Formula*) with an error message (attribute *errorMsg*). The formulae are defined (attribute *value*) by means of a particular language called OASIS, which is similar to the OCL language. Thus, the valuations and the specification of the derived attributes (class *ValueSpecification*) correspond to conditional behaviour elements, and the transactions, and operations correspond to behaviour elements. The preconditions and the integrity constraints correspond to constraint behaviour elements.

The interaction model has a set of interaction units (class *InteractionUnit*) and a set of auxiliary patterns (class *AuxPattern*) that allow the specification of the graphical user interface at an abstract level. The interaction units can be instances (*InstanceIU*), set of instances (*PopulationIU*), services (*ServiceIU*), and composite units (*MasterDetailIU*). The master-detail interaction units correspond to composite interaction units (class *DependentIU*), which are comprised of a master part (class *IndependentIU*) and a set of detail interaction units (class *DependentIU*). In the master part, only instances or populations can be used. In the detail part, instances, populations and other master detail interaction units can be used. Since, the instance interaction units and the population interaction units can be used independently of other interaction units; we classify them in the class *IndependentIU*. However, these interaction units can also be used inside the detail part of master detail interaction units, so we classify them in the class *DependentIU*. The independent interaction units have display sets (class *DiaplaySet*) to present the data. Each display pattern has a set of attributes (association *relatedattribute*) that are specified in the structural model, from which the data will be recovered to show the users of the application. The independent interaction units can have actions (class *ActionSet*) to present the set of services (throw a *ServiceIU*) that can be executed by the users over the instances shown in the interaction units. In addition, the independent interaction units can have navigations (class *NavigationSet*) to present the interaction units that can be accessed. The population interaction units can also have filters (class *Filter*) to search for information in a set of instances, which must be specified with the corresponding formula (class *Formula*). The service interaction units have entry (class *EntryPattern*) and selection patterns (class *SelectionPattern*), which have associated formulae composed of a condition (association *condition*) and an effect (association *effect*).

Since the quality metamodel has been specified using the standards of metamodelling, this metamodel eliminates redundancy of the elements defined and can be implemented using open-source modeling tools.

4.2. The OOmCFP Procedure to Identify Defects. Before the specification of the OCL rules, the types of defects that the conceptual models used in MDD environments must be known. To do this, we use a Functional Size Measurement (FSM) procedure designed for the OO-Method MDD approach, which is called OOmCFP [49]. This FSM procedure was developed to measure the functional size of the applications generated in an MDD environment from the conceptual models. The OOm CFP measurement procedure was defined in accordance with the COSMIC measurement manual version 3.0 [50]. Thus, a mapping between the concepts used in COSMIC and the concepts used in the conceptual model of the MDD approach has been defined [51].

The OOmCFP procedure is structured using three phases: the strategy phase, the mapping phase, and the measurement phase. The strategy phase addresses the four key parameters of software functional size measurement that must be considered before actually starting to measure: purpose of the measurement, scope of the measurement, identification of functional users, and level of granularity that should be measured. The mapping phase presents the rules to identify the functional processes, data groups, and data attributes in the software specification (i.e., in the conceptual model) depending on the parameters defined in the strategy phase. The measurement phase presents the rules to identify and measure the data movements that occur between the functional users and the functional processes.

OOmCFP starts with the definition of the strategy to perform the measurement. The *purpose* of the measurement in OOmCFP is defined as measuring the accurate functional size of the OO-Method applications generated in an MDD environment from the involved conceptual models. The *scope* of the measurement defines a set of functional user requirements that will be included in a measurement exercise. For OOmCFP, the scope of the measurement in OOmCFP is the OO-Method conceptual model, which is comprised of four models (Object, Dynamic, Functional, and Presentation), which allow a fully working software application to be generated.

Once the scope of the measurement has been determined, it is important to identify the *layers*, the *pieces of software*, and the *peer components* that make up the applications. Since the OO-Method software applications are generated according to a three-tier software architecture, we distinguish three *layers*: a client layer, which contains the graphical user interface; a server layer, which contains the business logic of the application; and a database layer, which contains the persistence of the applications (see Figure 2). In each layer of an OO-Method application, there is a *piece of software* that can interchange data with the pieces of software of the other layers. Thus, we distinguish, respectively, three pieces of software in an OO-Method application: the client piece of software, the server piece of software, and the database piece of software (see Figure 2).

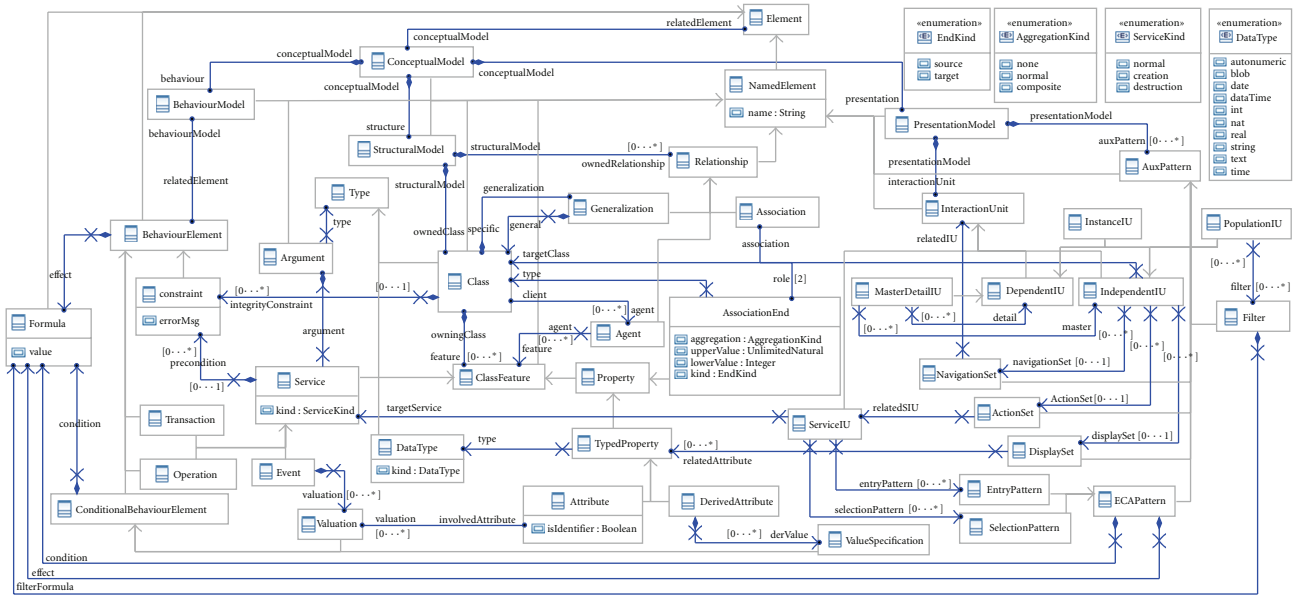

Figure 1: A metamodel for defect detection in MDD-oriented conceptual models.

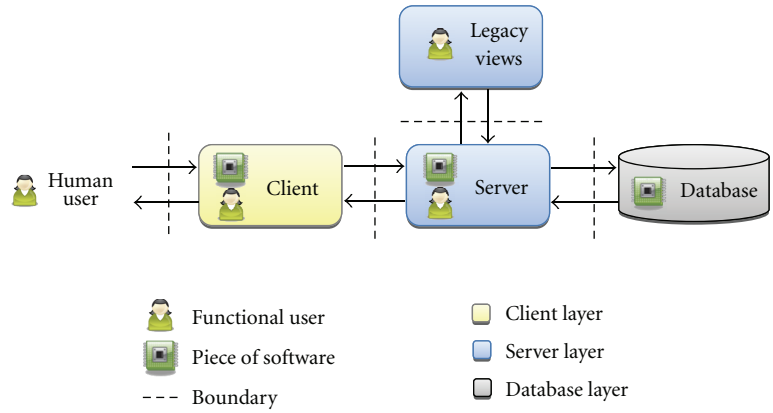

Figure 2: Layers, pieces of software, and functional users of the OO-Method applications.

Since the *functional users* are the types of users that send (or receive) data to (from) the functional processes of a piece of software, the functional users of the OO-Method applications are the human users, the client component of the software, the server component of the software, and the legacy views (see Figure 2). We called these users as "human functional user", "client functional user", "server functional user", and "legacy functional user", respectively.

Once the strategy is defined, OOmCFP starts a mapping phase. A *functional process* corresponds to a set of Functional User Requirements comprising a unique, cohesive, and independently executable set of data movements. A functional process starts with an entry data movement carried out by a functional user given that an event (*triggering event*) has happened. A functional process ends when all the data movements needed to generate the answer to this event have been executed. In the context of OOmCFP, the "human functional user" carries out the *triggering events* that occur in the real world. This functional user starts the *functional processes* that occur in the client layer of the application. In this layer, the functional processes are represented by the interaction units of the conceptual model that can be directly accessed by the "human functional user". The "client functional user" activates *triggering events* that occur in the interaction units of the presentation model. The "client functional user" starts *functional processes*, which are the actions that carry out the server layer of the software in response to the triggering events that occur in the client layer of the software. The "server functional user" carries out the *triggering events* that occur in the server layer of the software. The "server functional user" starts *functional processes*, which are the actions that the database layer carries out in response to the triggering events of the server layer, and the actions that the client layer carries out in response to triggering events of the server layer of the software. The "legacy functional user" activates *triggering events* that occur in the legacy system. The "legacy functional user" starts *functional processes*, which are the actions that the server layer

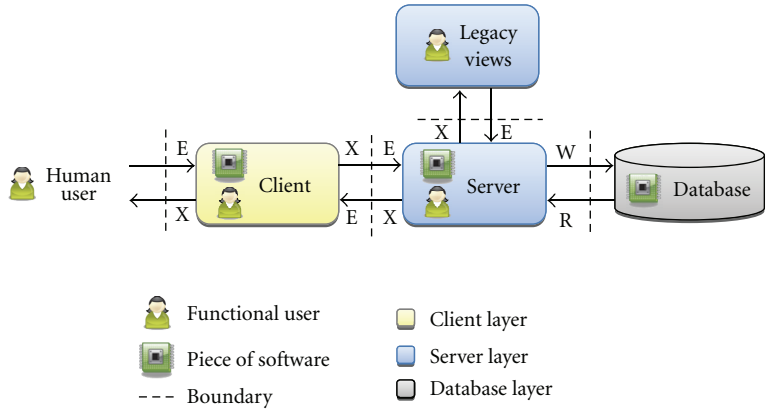

Figure 3: Data movements between the users and layers of an OO-Method application.

of the software carries out to interact with the legacy system. The *data groups* correspond to the classes of the structural model that participate in the functional processes. The *data attributes* correspond to the attributes of the classes identified as data groups.

In the measuring phase, the data movements correspond to the movements of data groups between the users and the functional processes. Each functional process has two or more data movements. Each data movement moves a single data group. A data movement can be an *Entry* (E), an *Exit* (X), a *Read* (R), or a *Write* (W) data movement. This proposal has 65 rules to identify the data movements that can occur in the OO-Method applications (see Figure 3). Each rule is structured with a concept of the COSMIC measurement method, a concept of the OO-Method approach, and the cardinalities that associate these concepts. These mapping rules detect the data movements (E, X, R, and W) of all the functionality needed for the correct operation of the generated application, which must be built by the developer of the application. This proposal has three measurement rules to obtain the functional size of each functional process of the application, each piece of software of the application, and the whole application. A complete description of OOmCFP can be found in its measurement guide (http://oomethod.dsic.upv.es/labs/images/OOmCFP/guide.pdf).

Since the OOmCFP procedure has been designed to obtain accurate measures of the applications that are generated from the OO-Method conceptual model [52] and has been automated to provide measurement results in a few minutes using minimal resources [53], we use it to verify the quality of conceptual models in three case studies. The OOmCFP measurement procedure assumes that the conceptual model is of high quality; that is, the OOmCFP procedure assumes that the conceptual model is consistent, correct, and complete. This is obviously an unreal assumption because conceptual models often have defects. Since a measurement procedure analyzes all the conceptual constructs that are related to the functionality of a system, we consider that a measurement procedure is a valuable tool for finding defects in conceptual models.

4.3. Defect Types. In order to determine the defect types of MDD conceptual models, the proposed metamodel and procedure were applied to three case studies with conceptual models of different functional sizes: a Publishing application (a small model), a Photography Agency application (a medium model), and an Expense Report application (a large model). We identified 39 defects and grouped them into 24 defect types (see [54]). For details, Table 2 shows the set of defect types that we identified using the OOmCFP procedure.

These defect types correspond to those related to structural models and those related to interaction models. This is one interesting contribution of our measurement procedure since, to our knowledge, there are no reported findings of defect types related to interaction models in the published literature. In order to formalize defect detection in the metamodel presented in Figure 1, we defined OCL rules to prevent the occurrence of the identified defects in the conceptual models. Table 3 shows the defect types and the OCL rules of our approach.

In order to identify the maximum number of defects using the quality model, we aggregated the defect types already found in the literature (see Table 1) with the corresponding OCL rules (see Table 4). We selected the defect types related to the class model, which is a diagram commonly used by several MDD proposals. We ruled out the defect types of the literature that were also identified using the OOmCFP measurement procedure.

In the three case studies mentioned above, the conceptual models did not achieve the characteristics of consistency and correctness due to the defect types presented in our approach. The OCL rules presented in Tables 3 and 4 can be implemented for the model compilers of MDD proposals in order to automatically verify the conceptual models with regard to these characteristics.

5. Conclusions

In this paper, we have presented a quality model to evaluate the conceptual models used in MDD environments

to generate final applications through well-defined model transformations. The quality model is comprised of the following: (1) a metamodel that contains a minimal set of conceptual constructs, their properties, and their relationships, which allows the complete specification of applications in the conceptual model of an MDD environment; and (2) a set of rules for the detection of defects in the model, which have been specified using OCL constraints [14].

The design of the metamodel has been systematically performed using an MDD approach as reference. This approach, which is known as OO-Method, has been successfully applied in the software industry. However, it is important to note that even though some elements of the metamodel are specific to the OO-Method MDD approach, equivalent modeling constructs can be found in other object-oriented MDD methods. Thus, this metamodel can be used as a reference to improve existent MDD approaches or as a starting point for the specification of new MDD-oriented modeling languages. Moreover, the main modeling constructs that compose the metamodel are the same constructs present in the UML specification. For this reason, the quality model presented here can be applied to other MDD methods that use UML-like models.

We take advantage of modeling, metamodeling, and transformation techniques to avoid having to manually identify defects in the conceptual models, which is an error-prone activity. Thus, the quality model (metamodel + OCL rules) has been designed for easy application to other MDD proposals. This is feasible because the EMOF standard is used to define the metamodel, which is supported by existent open-source tools [41, 48] and is also used by other MDD proposals for the specification of their modeling languages. Therefore, we can firmly state that the quality model proposed here contributes substantially to improving the MDD processes and the quality of software products generated in this context.

For future works, we plan to apply the quality model into different MDD approaches by using an integration process that automatically generates metamodeling extensions [55–57]. By using the integration proposal, we plan to show how the proposed quality model allows the automatic verification of the list of defect types found in MDD proposals. We also plan to develop empirical studies to evaluate the quality of conceptual models for different MDD proposals. The findings from this evaluation will be used to build a knowledge base for further improvements in the evaluation of the quality of conceptual models related to MDD approaches.

Acknowledgement

This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and GVA ORCA PROMETEO/2009/015.

References

[1] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, 2003.

[2] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: model-driven development,," *IEEE Software*, vol. 20, no. 5, pp. 14–18, 2003.

[3] D. L. Moody, "Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions," *Data & Knowledge Engineering*, vol. 55, no. 3, pp. 243–276, 2005.

[4] O. I. Lindland, G. Sindre, and A. Solvberg, "Understanding quality in conceptual modeling," *IEEE Software*, vol. 11, no. 2, pp. 42–49, 1994.

[5] T. H. Davenport and L. Prusak, *Working Knowledge: How Organisations Manage What They Know*, Business School Press, Boston, Mass, USA, 1998.

[6] W. L. Neuman, *Social Research Methods: Qualitative and Quantitative Approaches* , Needham Heights, Mass, USA, Allyn & Bacon, 4th edition, 2000.

[7] ISO/IEC, ISO/IEC 9126-1, Software Engineering—Product Quality—Part 1: Quality model,2001 .

[8] M. Genero, M. Piattini, and C. Calero, "A survey of metrics for UML class diagrams," *Journal of Object Technology*, vol. 4, no. 9, pp. 59–92, 2005.

[9] S. S.-S. Cherfi, J. Akoka, and I. Comyn-Wattiau, "Conceptual modeling quality—from EER to UML schemas evaluation," in *Proceedings of the 21st International Conference on Conceptual Modeling (ER '02)*, S. Spaccapietra, S. T. March, and Y. Kambayashi, Eds., vol. 2503 of *Lecture Notes in Computer Science*, Springer, Tampere, Finland, October 2002.

[10] R. M. Wilson, W. B. Runciman, R. W. Gibberd, B. T. Harrison, J. D. Newby, and J. D. Hamilton, "The quality in Australian health care study," *The Medical Journal of Australia*, vol. 163, no. 9, pp. 458–471, 1995.

[11] C. Lange and M. Chaudron, "Defects in industrial UML models—a multiple case study," in *Proceedings of the 2nd Workshop on Quality in Modeling of MODELS (QiM '07)*, pp. 50–79, Nashville, Tenn, USA, 2007.

[12] U. Bellur and V. Vallieswaran, "On OO design consistency in iterative development," in *Proceedings of the 3rd International Conference on Information Technology: New Generations (ITNG '06)*, pp. 46–51, IEEE, April 2006.

[13] J. Vanderdonckt, "Model-driven engineering of user interfaces: promises, successes, and failures," in *Proceedings of the 5th Annual Romanian Conference on Human-Computer Interaction (ROCHI '08)*, S. Buraga and I. Juvina, Eds., pp. 1–10, Matrix ROM, Iasi, Romania, 2008.

[14] OMG, Object Constraint Language 2.0 Specification, 2006.

[15] ISO, ISO Standard 9000-2000: Quality Management Systems: Fundamentals and Vocabulary, 2000.

[16] IEEE, IEEE 1044 Standard Classification for Software Anomalies, 1993.

[17] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[18] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, New York, NY, USA, 2nd edition, 2000.

[19] G. H. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object oriented designs: using reading techniques to increase software quality," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, & Applications (OOPSLA '99)*, pp. 47–56, Denver, Colo, USA, October 1999.

[20] V. R. Basili, S. Green, O. Laitenberger et al., "The empirical investigation of perspective-based reading," *Empirical Software Engineering Journal*, vol. 1, no. 2, pp. 133–164, 1996.

[21] O. Laitenberger, C. Atkinson, M. Schlich, and K. E. Emam, "Experimental comparison of reading techniques for defect detection in UML design documents," *Journal of Systems & Software*, vol. 53, no. 2, pp. 183–204, 2000.

[22] R. Conradi, P. Mohagheghi, T. Arif, L. C. Hegde, G. A. Bunde, and A. Pedersen, "Object-oriented reading techniques for inspection of UML models—an industrial experiment," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '03)*, vol. 2749 of *Lecture Notes in Computer Science*, pp. 483–501, Springer, July 2003.

[23] H. Gomaa and D. Wijesekera, "Consistency in multiple-view UML models: a case study," in *Proceedings of the Workshop on Consistency Problems in UML-based Software Development II*, pp. 1–8, IEEE, San Francisco, Calif, USA, 2003.

[24] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, Reading, Mass, USA, 2000.

[25] L. Kuzniarz, "Inconsistencies in student designs," in *Proceedings of the Workshop on Consistency Problems in UML-Based Software Development II*, pp. 9–17, IEEE, San Francisco,Calif, USA, 2003.

[26] B. Berenbach, "The evaluation of large, complex UML analysis and design models," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 232–241, IEEE Computer Society, May 2004.

[27] C. Lange and M. Chaudron, "An empirical assessment of completeness in UML designs," in *Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE '04)*, pp. 111–121, IEEE, 2004.

[28] F. Leung and N. Bolloju, "Analyzing the quality of domain models developed by novice systems analysts," in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pp. 1–7, January 2005.

[29] IEEE, IEEE 610 Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries, 1990.

[30] OMG, UML 2.1.2 Superstructure Specification, 2007.

[31] K. Berkenkötter, "Reliable UML models and profiles," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 203–220, 2008.

[32] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *IEEE Computer*, vol. 39, no. 2, pp. 59–66, 2006.

[33] A. L. Opdahl and B. Henderson-Sellers, "A unified modelling language without referential redundancy," *Data & Knowledge Engineering*, vol. 55, no. 3, pp. 277–300, 2005.

[34] G. Giachetti, B. Marín, and O. Pastor, "Perfiles UML y desarrollo dirigido por modelos: desafíos y soluciones para utilizar UML como lenguaje de modelado específico de dominio," in *Proceedings of the V Taller Sobre Desarrollo de Software Dirigido por Modelos (DSDM '09)*, Gijón, Spain, 2008.

[35] OMG, MDA Guide Version 1.0.1, 2003.

[36] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano, "The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming," *Information Systems*, vol. 26, no. 7, pp. 507–534, 2001.

[37] O. Pastor and J. C. Molina, *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*, Springer, New York, NY, USA, 2007.

[38] O. Pastor, F. Hayes, and S. Bear, "OASIS: an object-oriented specification Language," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '92)*, pp. 348–363, Manchester, UK, 1992.

[39] J. Vanderdonckt, "A MDA-compliant environment for developing user interfaces of information systems," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAISE '05)*, O. Pastor and J. Falcão e Cunha, Eds., vol. 3520, pp. 16–31, Springer, Porto, Portugal, 2005.

[40] P. Molina, *Especificación de Interfaz de Usuario: De los Requisitos a la Generación Automática*, Universidad Politécnica de Valencia, Valencia, España, 2003.

[41] Eclipse Modeling Project, February 2010, http://www.eclipse.org/modeling/.

[42] Eclipse Modeling Framework Project, February 2010, http://www.eclipse.org/modeling/emf/.

[43] Eclipse Graphical Modeling Framework Project, February 2010, http://www.eclipse.org/gmf/.

[44] Eclipse ATL Project, February 2010, http://www.eclipse.org/m2m/atl/.

[45] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proceedings of the Satellite Events at the MoDELS 2005 Conference*, vol. 3844 of *Lecture Notes in Computer Science*, pp. 128–138, Springer, 2006.

[46] OMG, QVT 1.0 Specification, 2008.

[47] B. Selic, "A systematic approach to domain-specific language design using UML," in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '07)*, pp. 2–9, May 2007.

[48] Eclipse Model Development Tools, February 2010, http://www.eclipse.org/modeling/mdt/.

[49] B. Marín, N. Condori-Fernández, O. Pastor, and A. Abran, "Measuring the functional size of conceptual models in an MDA environment," in *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAISE '08)*, Z. Bellahsene, C. Woo, E. Hunt, X. Franch, and R. Coletta, Eds., pp. 33–36, Montpellier, France, 2008.

[50] A. Abran, J. Desharnais, A. Lesterhuis, et al., The COSMIC Functional Size Measurement Method—Version 3.0, 2007.

[51] B. Marín, N. Condori-Fernández, and O. Pastor, "Design of a functional size measurement procedure for a model-driven software development method," in *Proceedings of the 3rd Workshop on Quality in Modeling of MODELS (QiM '08)*, J.-L. Sourrouille, M. Staron, L. Kuzniarz, P. Mohagheghi, and L. Pareto, Eds., pp. 1–15, Touluse, France, 2008.

[52] B. Marín, O. Pastor, and A. Abran, "Towards an accurate functional size measurement procedure for conceptual models in an MDA environment," *Data & Knowledge Engineering*, vol. 69, no. 5, pp. 472–490, 2010.

[53] B. Marín, O. Pastor, and G. Giachetti, "Automating the measurement of functional size of conceptual models in an MDA environment," in *Proceedings of the Product-Focused Software Process Improvement (PROFES '08)*, vol. 5089 of *Lecture Notes in Computer Science*, pp. 215–229, Springer, 2008.

[54] B. Marín, G. Giachetti, and O. Pastor, "Applying a functional size measurement procedure for defect detection in MDD environments," in *Proceedings of the 16th European Conference on Systems & Software Process Improvement and Innovation (EUROSPI '09)*, R.V. O' Connor, Ed., vol. 42 of *Communications in Computer and Information Science*, pp. 57–68, Springer, Madrid, Spain, 2009.

[55] G. Giachetti, B. Marín, and O. Pastor, "Integration of domain-specific modelling languages and UML through UML profile extension mechanism," *International Journal of Computer Science & Applications*, vol. 6, no. 5, pp. 145–174, 2009.

[56] G. Giachetti, B. Marín, and O. Pastor, "Using UML as a domain-specific modeling language: a proposal for automatic generation of UML profiles," in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE '09)*, P. van Eck, J. Gordijn, and R. Wieringa, Eds., vol. 5565 of *Lecture Notes in Computer Science*, pp. 110–124, Springer, Amsterdam, The Netherlands, 2009.

[57] G. Giachetti, F. Valverde, and O. Pastor, "Improving automatic UML2 profile generation for MDA industrial development," in *Proceedings of the 4th International Workshop on Foundations and Practices of UML (FP-UML '08)*, vol. 5232 of *Lecture Notes in Computer Science*, pp. 113–122, Springer, 2008.