An Empirical Assessment of Completeness in UML Designs

Christian Lange, Michel Chaudron
Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{C.F.J.Lange, M.R.V.Chaudron}@tue.nl

Abstract

Delivering high quality software in an economic way requires advanced control over the software development process and the product in all stages of its life-cycle. The use of metrics as means of control and improvement plays an important role in software engineering.

Interviews with industrial software engineers identified incompleteness of UML designs as a potential problem for subsequent stages of development.

In this paper we propose a definition of completeness of a UML model and present a set of rules to assess model completeness. We report results from industrial case studies to assess the level of completeness in practice. The amount of completeness and consistency rule violations was very high. Different case studies showed large variations in conformance to and violations of rules.

1. Introduction

The pursuit of delivering high quality software in an economic way requires advanced control over the software development process and the product in all stages of its life-cycle. The use of metrics [10, 21] for means of control and improvement plays an important role in software engineering. Software product metrics are most frequently applied at the source code level, i.e. when the product is implemented. This is rather late in the life-cycle of a software system.

It is well known, that the cost of removing defects and enhancing (poor) designs increases with the stage of the life-cycle, i.e. early corrections are desirable [29]. This observation encourages to lift the assessment of software products to an earlier stage. This can be done by collecting metrics from earlier artifacts like the architecture and design models. These measurements of early artifacts can be automated (e.g. the SAAT tool [18]), such that metrics can be collected with little effort.

The Unified Modeling Language [20] is the de facto industry standard for modeling software designs and is gaining importance in the field of modeling software architectures [12, 17, 24]. Therefore many tools and techniques for analyzing UML models are being developed [11, 25].

In software engineering, models of different levels of abstraction are built. Typically, the level of abstraction of subsequent models decreases and as the developers gain knowledge about the system, the models become more detailed. The final and most detailed artifact is the source code, which is written in some programming language.

Programming languages are by definition formal, i.e. a compiler can check their syntax. This determines their semantics. For UML it is generally not possible to relate a unique semantics to a model.

As the solution is not totally clear in an early development stage, modeling languages are designed to describe systems on a higher abstraction level that typically allow several possible solutions. The possibility of multiple solutions leads to miscommunications and integration problems (Figure 1). As projects progress, more details are added and the number of possible solutions decreases.

By offering nine diagram types, powerful extension mechanisms and no strict semantics, the UML allows a large degree of freedom.

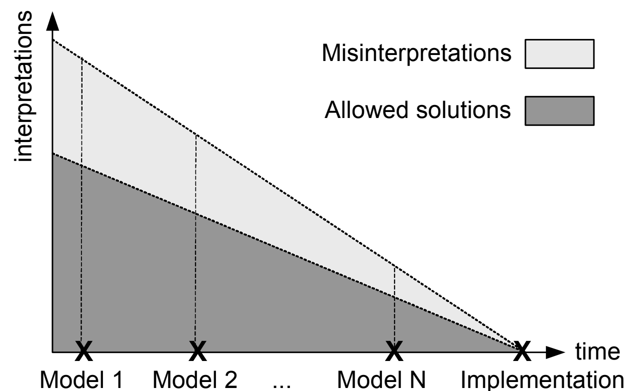

Figure 1 - Interpretation sets of design artifacts

The large degree of freedom allowed by UML models allows for conflicts between diagrams. These conflicts amplify the problems of miscommunication. Therefore projects aim to minimize the conflicts in a model. A high

number of conflicts indicates that the model is far from complete.

Industrial partners put forward another reason for minimizing the set of conflicts. They have shown large interest in using metrics based on UML models as predictors for the attributes of the final system. However, it is not clear how the degree of incompleteness of UML models affects the precision and accuracy of these predictions. This led us to investigate the magnitude of completeness of UML models in industrial practice.

In section 2 we explain the notion of model completeness and explain the techniques we developed to assess completeness. In section 3 we present the underlying meta model. In section 4 we present the results of industrial case studies that we conducted. In section 5 related work is put into perspective. Our concluding findings are given in section 6.

2. Completeness

In interviews and surveys [14] we conducted during this project amongst practitioners in large-scale industrial organizations the following problems due to doubt about model completeness were reported:

- uncertainty about accuracy and precision of estimates based on model analysis,
- miscommunication,
- integration overhead

What are the attributes of *model completeness* and how can we get a grip on them? To get insight in completeness problems in practice we started with empirical research. We will start with a decomposition of the notion of completeness, based on the two main stakeholders of a software engineering project – the client and the software maker, who have their own perspective on completeness.

The client's perspective: The client starts with an idea that is – probably in cooperation with the software maker – transformed into user requirements. Apart from certain restrictions (like cost, and delivery time) the client is interested in a product that exactly meets the user requirements. This is validated using an acceptance test.

The software maker's perspective: For the software maker, apart from delivering the product according to the requirements, it is important to develop the software in a (predictable and) economical way, i.e. low resource cost, short time-to-market. This enables the software maker to maximize profits.

The success of pursuing the objectives related to both above mentioned perspectives depends on the controllability of the software development process. Being in control of the process means knowing ones progress. Measures of completeness of a model give insight in the progress of the product, and hence of the process progress.

Figure 2 relates software development artifacts in terms of the level of abstraction (decreases from top to bottom).

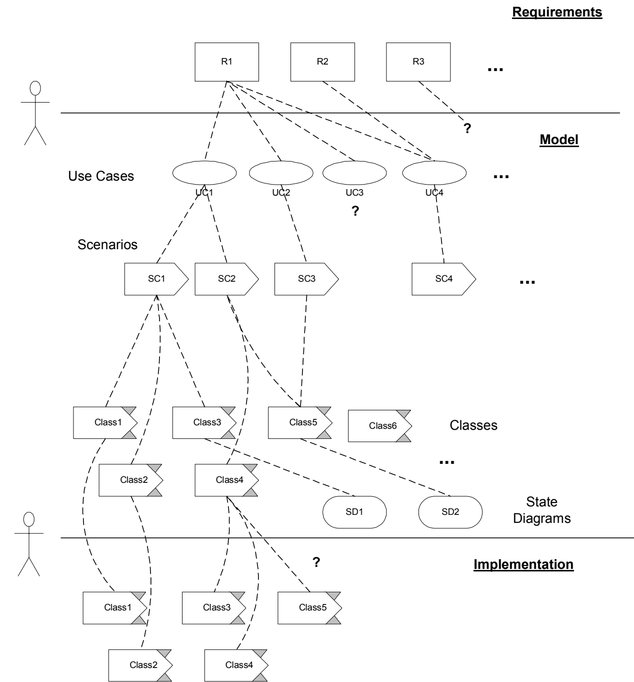

Figure 2 – Scope

The top layer represents the requirements of the system. The requirements are detailed by the use cases (indicated by the dashed lines). The sequence charts (SC) are more detailed instantiations of the use cases (a SC instantiating a use cases is indicated by a dashed line). The relations between classes and SCs mean that an instantiation of the class occurs as object in the SC. The internal behavior of classes can be described by state diagrams (this relation is also indicated by dashed lines). The bottom layer is the implementation, the actual source code. The dashed lines mean that the implementation classes are detailed model classes. The scope of this project lies on analyzing completeness of the actual UML model. The boundaries of the UML model are indicated by the solid lines. Within the model the rules and metrics can be calculated by tool support. Validating the boundaries between requirements and model, and model and implementation, needs human intervention (indicated by the actor symbol).

We relate the two mentioned stakeholder perspectives to our initial question using Figure 3. According to the client's perspective a complete model covers all requirements and ensures that the product to be delivered complies with these requirements. The tracing of the functional requirements is to a certain extent possible by automated tool-support. For non-functional requirements

this is very difficult. Validating this perspective of completeness is in fact done in the acceptance test.

The scope of modeling is given in between the two horizontal lines. This is the scope where the conception of completeness is dealt with that relates to the software maker's perspective. In the software maker's perspective the main concern is the maturity of the model description, i.e. the collection of UML diagrams. As the UML is used as a communication medium (immediately within a project team, but also to "next generations" of developers maintaining or extending a system) the main concern in this view is to ensure that the UML model is composed in such a way that ambiguity, miscommunication and integration effort are minimized. The methods developed in this project are designed to contribute to increase the model completeness according to the above described conception.

In the UML, diagram types are not mutually disjoint, but they are overlapping. This is logical, as they eventually describe one and the same model. This overlap is a possible cause of ambiguity and therefore a critical point in our further analysis.

We break the discussed concept of completeness down to the subconcepts completeness of requirements tracing (related to the client's perspective) and the completeness of UML modeling (related to the software maker's perspective). In the sequel we will focus on the latter concept, which addresses the model ambiguity.

Model completeness can be decomposed into: well-formedness of each single diagram, consistency between diagrams, and completeness amongst diagrams (Figure 3). We discuss each of these in the next subsections.

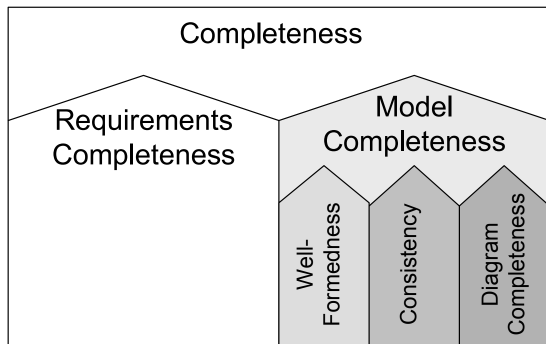

Figure 3 - Decomposition of Model Completeness

2.1 Well-formedness

The basic level of analysis is within a single diagram.

Each diagram must follow certain restrictions, so called well-formedness rules and conventions to maximize its understandability. In the semantics section of the UML specification [20] well-formedness rules for model elements are defined. The majority of the pre-defined rules is enforced by dedicated case tools.

For our experiments we consider some additional restrictions. Possible violations of this category are classes without names or state diagrams without start and end states.

These restrictions prevent designers from producing diagrams that contain serious flaws.

2.2 Consistency

Overlapping diagrams refer to common information. If two diagrams in the model contain contradictory common information, then there is an *inconsistency* between these diagrams. An instance of this condition occurs if in a message sequence chart, an object's method is called while the corresponding class in the class diagram does not contain that method.

The goal of consistency rules therefore is to help identify inconsistencies. If an inconsistency between two diagrams remains unresolved, miscommunication and high integration effort are likely to happen. Figure 4 depicts schematically an inconsistency.

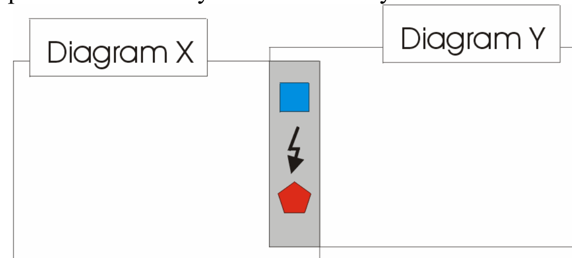

Figure 4 - Inconsistency

2.3 Diagram Completeness

The elements in the common part of overlapping diagrams are contained in both diagrams. A UML model is complete, if for each element in the one diagram it's expected counterpart in the other diagram is present. An (inter-diagram) incompleteness therefore is when there is an element in the overlapping part of two diagrams (or views) in the one diagram without matching counterpart in the other diagram.

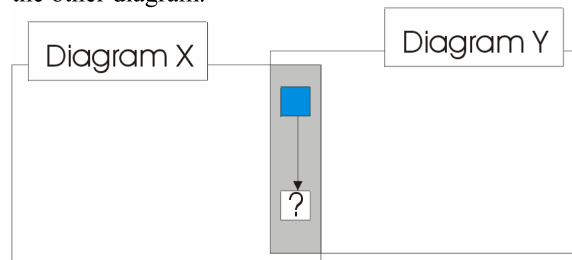

Figure 5 - Diagram Incompleteness

For example when there is a use case in an use case diagram, but no interaction diagram or classes in a class

diagram are related to the use case, the condition for incompleteness is met, as clearly the model does not contain elements implementing the functionality required by the use case. (The use case therefore is a leaf). Violations of model completeness can cause unimplemented functionality. Figure 5 depicts model incompleteness schematically.

2.4 Ambiguity of Consistency and Completeness

The border between inconsistency and incompleteness is for some rules unclear: for instance an apparently missing model element is possibly present in the model under a different name.

Figure 6 shows a class diagram and a message sequence chart (whose objects are supposed to be instantiations of the classes in the class diagram). When you consider the class diagram to be the leading diagram, we have an inconsistency in the message sequence chart, as Class B has a method named Method_y, not Method_z. But when you consider the message sequence chart to be the "correct" diagram, the class diagram is not complete, as Class B does not contain Method_z. Hence, we see, that there are cases where it depends on which diagram is the leading one whether the violation is an inconsistency or an incompleteness.

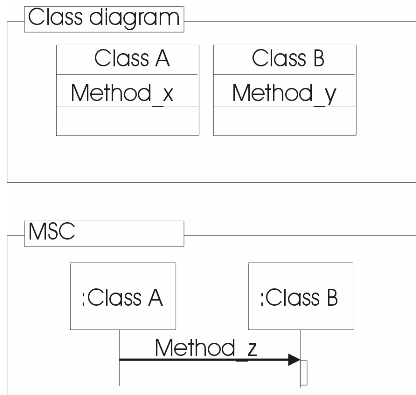

Figure 6 - Incompleteness or Inconsistency?

3. Meta Model

In this study we propose a framework to detect completeness flaws. The proposed set of rules is based on a meta model which serves as a data model for the information stored in an UML model. A possible and obvious candidate is the UML meta model. This meta model extensively covers all facilities of the UML. We have concentrated on the four most commonly used [14] diagram types: use case diagrams, class diagrams, sequence charts and state chart diagrams. Hence our meta model is a relational model that covers a subset of the UML meta model. A relational model allows the definition of the completeness rules in relational algebra.

In the following we present two examples of rule expressions. The rules are further explained in subsection 4.2.2. Let `E` denote the set of all messages, let `M` denote the set of all methods, let `CM` denote the set of all class-method-relationships, let `As` denote the set of all associations and let `class(O)` denote the class-type of object `O`.

Rule S2: Messages must correspond to Method

$(\forall e \in E : (\exists m \in M, : e.name = m.name \wedge (class(e.callee), m) \in CM))$

Rule S3: Messages only between related Classes

$\forall e \in E : (e.caller = e.callee \vee class(e.caller), class(e.callee)) \in As)$

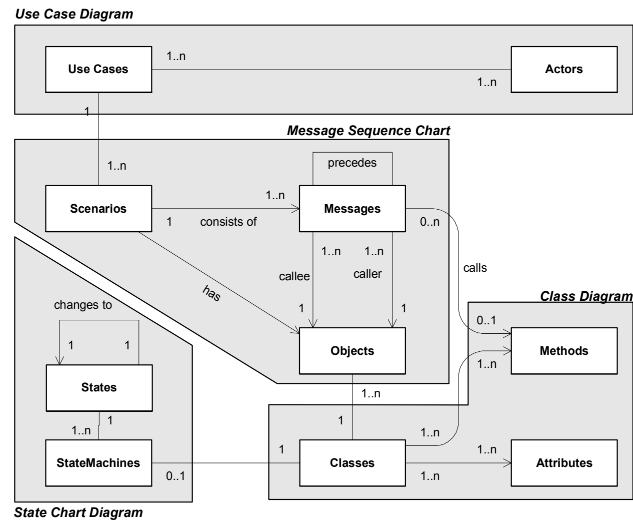

Figure 7 - Relational Meta Model

4. Case Studies

The goal of this study was to assess the status of completeness of UML models in industrial practice. Cooperation between academia and industry is of essential importance to computer science, especially to the discipline of software engineering research, but the cooperation is still far away from being perfect. It is well known that a solid basis of empirical studies is a large sample size. As industrial models contain confidential intellectual property, it is challenging to find many industrial contributions. Nevertheless we found three participating organizations providing us with a heterogeneous set of case studies.

In subsection 4.1 we give a short introduction of these case studies. In subsection 4.2 a selection of the rules and the quantitative findings from the case studies are presented. In subsection 4.3 the observations from the case studies are summarized.

4.1 Description of Case Studies

First we give some general characteristics of the considered case studies.

4.1.1 Case A: Image Processing

An organization provided us with two models (A1, A2) of different subsystems of the same system. Both models belong to the same release version of the system, whose domain is medical image processing. The two models were built by different designers within the same development team. The size of the models A1 and A2, which are design models, can roughly be indicated by the number of classes: 108 and 168 classes respectively (Table 1).

4.1.2 Case B: Embedded Controller

The models B1 and B2 are provided by a different organization, they are high-level analysis models (as described by Jacobson [9]) of an entire system, an embedded controller. The models were created by the same designer. The models describe two subsequent version of the system, B2 is not an increment of B1, but was recreated from scratch. Their size is 34 and 69 classes for B1 and B2 respectively (Table 1). Remarkable is that B1 has only and B2 has no messages. As it is an analysis model, the architect chose to focus on identifying the structure only, and therefore defining methods is postponed. The relatively low number of messages in 51 drew our attention: when we asked the architect for the reason, he discovered, that the model contained a few scratch sequence charts. It was supposed to have no sequence charts at all.

4.1.3 Case C: Prototyping Framework

The models C1 and C2 describe a system designed by two post graduate students for an industrial client. The design level models describe a prototyping framework. C2 describes the succeeding version of C1. The size of the models is 45 and 46 classes (Table 1). The project is significantly smaller in project staff than the other industrial projects and, deadlines were less strict than in the other presented case studies. We incorporated this case to investigate, whether the different conditions (e.g. off-critical-path, lesser practical experience, time…) result in different findings concerning completeness.

Table 1 - Basic Size Metrics

	A1	*A2*	*B1*	*B2*	*C1*	*C2*
Use Cases	0	21	32	49	11	11
Objects	200	544	297	30	103	100
Classes	108	168	34	69	45	46
Methods	340	406	1	0	142	180
Messages	613	853	705	51	210	219

Cases A1 and A2 were are parts of the same release version and were used as basis for implementation. Cases B1 and B1 were part of an early release of the system on which design activities were based. Case C2 was a release on which implementation is based. C1 precedes C2 by 2 weeks.

4.2 Results of the Completeness Analysis

In this section we present a selection of our developed rules and give the results obtained from the case studies. An overview of the results is given in Table 2, where the percentage indicates the amount of model elements not adhering to the specific rule. This normalized presentation allows comparison between different models. Table 2 is used to show the magnitude of specific rule violations. In section 4.3 the specific observations for the models are presented.

4.2.1 Assessment of Well-formedness Rules

Objects without Name. (W1 in Table 2) Objects are instantiations of classes in sequence diagrams. By naming an object the model gains in detail and understandability, especially in the case, that instantiations of the same class occur, a name is necessary to distinguish different roles. The relatively best results are scored in cases A1 and A2, with still 52% and 61,6% of all objects not named. In the other cases at least 75% of the objects do not have a name.

Classes without methods. (W2) Classes without methods cannot receive messages. Their data can only be accessed directly, which is a violation of the object-oriented paradigm (encapsulation). In this rule we count a class' own defined methods and the inherited methods.

As cases B1 and B2 are high level analysis models, it is not surprising, that no classes have methods. It is the designer's deliberate choice to omit the methods, and therefore this result is not problematic. The student's designs C1 and C2 have the best score, but still 20% of the classes neither defines, nor inherits any methods.

Interfaces without methods. (W3) According to the UML specification [20] "An interface is a named set of operations that characterize the behavior of an element". Note that our notion of "method" refers to what is called "operation" in the specification. An interface without any methods is therefore senseless. The models of A1 and A2 have about 9% of the interfaces without methods (which is still a lot). B1 does not define any interfaces and all the interfaces defined in B2 do violate the rule. This is clear evidence for the models being very incomplete with respect to interfaces. In the models of C1 and C2 about 60% of all interfaces violate the rule. This means that the designers must add a lot of information to come to an implementation. This is alarming for a design model, i.e. the model is assumed to be "close" to the implementation.

Table 2 - Results received from the case studies

ID	Well-formeness	A1	A2	B1	B2	C1	C2
W1	Objects without Name	52.00%	61.58%	91.92%	86.67%	76.70%	75.00%
W2	Classes without Methods	45.77%	51.19%	100.00%	100.00%	20.00%	23.91%
W3	Interfaces without Methods	8.82%	9.38%	N/A	100.00%	60.00%	61.29%
W4	Abstract Classes in Sequence Diagram	4.23%	0.00%	0.00%	8.70%	0.00%	0.00%
W5	Public Attributes	67.23%	5.08%	N/A	N/A	0.00%	2.44%
	Consistency	**A1**	**A2**	**B1**	**B2**	**C1**	**C2**
S1	Messages without Name	0.00%	0.00%	0.28%	0.00%	0.00%	0.46%
S2	Messages without Method	58.73%	7.62%	100.00%	100.00%	27.14%	27.40%
S3	Messages between unrelated Classes	71.94%	79.37%	77.73%	43.14%	81.90%	81.74%
	Completeness	**A1**	**A2**	**B1**	**B2**	**C1**	**C2**
C1	Classes not called in Sequence Diagram	46.48%	59.52%	35.29%	84.06%	42.22%	43.48%
C2	Interfaces not called in Sequence Diagram	100.00%	87.50%	N/A	100.00%	70.00%	70.97%
C3	Methods not called in Sequence Diagram	67.65%	77.59%	N/A	N/A	40.14%	55.00%

Abstract Classes in Sequence Diagram. (W4) The rule that abstract classes should not be instantiated (as an object in a sequence diagram) is adopted from object-oriented programming. In UML modeling this rule is arguable, nevertheless the results show that the designers stick to this rule, probably due to their backgrounds in object-oriented programming. The only two case studies where this rule is not totally followed still have results less then 10%.

Public Attributes. (W5) Encapsulation, one of object-oriented programming's basic principles, states that the data in classes should only be accessed via the class' methods. To ensure this, the attributes, i.e. the data, should not be declared public. Except for A1, which does not adhere to this rule with 67% of public attributes, all other models adhere to this rule relatively strictly.

4.2.2 Assessment of Consistency Rules

Messages without Name. (S1) Sequence diagrams describe the interaction of class instantiations. Messages that are not labeled with a name introduce a lack of clarity whether it is unclear which method they call or which purpose they satisfy. In the scrutinized case studies this rule is hardly violated. In B1 two messages out of 705 (0.28%) and in C2 one message out of 219 (0.46%) do not have a name.

Messages without Method. (S2) This is a stronger version of rule S1: each message must correspond to a method in the receiving class. This rule is less applicable to high abstraction-level models, as details are hidden there. This can be observed in the models B1 and B2, which have no methods defined (by purpose), therefore we discard this rule for these cases. About 27% of the messages in the models C1 and C2 violate this rule (at least from the previous rule we know that aside from one exception all messages have a name). In A2 this rule is followed relatively strictly (only 7.6% violations) the (different) designer of A1 leaves almost 60% of the messages unrelated to a method.

Messages between unrelated Classes. (S3) This rule is adopted from the Law of Demeter [15] from object-oriented programming, which says "only talk to your immediate friends that share your concerns". This rule is not adhered to in the considered models. Interesting is the observation, that for all (except B2) models the amount of violations is around 80%. Case B2 is regarded as an outlier here, because it is hardly described by sequence diagrams (Table 1).

4.2.3 Assessment of Completeness Rules

Classes not called in Sequence Diagram. (C1) Sequence diagrams illustrate the interaction of classes. Therefore it is desirable that the sequence diagrams cover all classes of the model. When a class is not called in a sequence diagram, its interactive behavior is not specified in the model. In the case studies we observe that four models (A1, B1, C1, C2) have values between 35% and 46%, which are the relatively best scores in this field, but

is still a lot. We have seen earlier, that in B2 sequence diagrams are scarce, which is reflected in the very large value of 84%.

Interface not called in Sequence Diagram. (C2) As interfaces are sets of methods their behavior should be described in sequence diagrams. The case studies show that this is not done in the field. We can omit B1 and B2 (no interfaces, respectively no methods defined), but for the other cases at least 70% (even all) interfaces are not described in sequence diagrams. This observation might be caused by the fact, that the designers assume from object-oriented programming, that interfaces cannot be instantiated (which is true for programming, but not for modeling).

Methods not called in Sequence Diagram. (C3) This rule is similar to the rule forcing classes to be called in sequence diagrams, except for that this rule applies at method level. The underlying thought is that uncalled methods are either obsolete or the model is incomplete with respect to behavior description. The results show similar (even worse) characteristics as for classes: C1 has the best score with 40.1% uncalled methods. The other cases range up to 77.6% uncalled classes. B1 and B2 are not considered, as no classes are defined in these models.

4.3 Observations from Case Studies

Based on the presented quantifications of rule violations we assessed the completeness of UML models in industrial practice. The models should ideally score low numbers of rule violations. Therefore it is alarming that for most rules the results are tremendously high.

We obtained the following observations:

- In cases A1 and A2 strong differences in the habits of the two different designers were encountered, especially for the rules W5 and S2.
- The designs (C1 and C2) have relatively the best scores. This might be based on the fact that the project was off the critical-path.
- When comparing C1 and its succeeding version C2 we observe a slight degradation with respect to completeness. This is mainly caused by added methods in the class diagrams, where the sequence charts were not changed to include the new methods. The automated evaluation helped the designers to identify this flaw.
- B2 has the overall worst scores. Especially the sequence chart related rules W4, S3 and C1 show outstanding results. For W4 (abstract classes in sequence diagram) B2 shows by far the largest score. The same holds for C1 (classes not in sequence diagram). Remarkably is that for S3 (messages between unrelated classes) B2 shows the lowest score with 43%, whereas all other models have results between 72% and 82%. When these results were presented to the designers, they discovered that the wrong version of the model had been released: a version where all sequence diagrams only were very premature scratches. Whereas the model with detailed sequence diagrams had accidentally not been released.

The results obtained can be used for two reasons:

- assessment of model completeness and the identification of “incomplete” spots. This can be used for instance to focus improvements.
- comparison between different models (different versions, different submodels, different processes…). This can be used to monitor progress.

5. Related Work

The main areas of related work are: empirical studies of software product metrics, and consistency analysis of UML.

The work on software product metrics attempts to quantify quality attributes of software design. The work on software product metrics has been oriented at the level of source code. In recent years, some work on metrics has been applied at the level of software design [1,2] and software architecture [19].

Empirical case studies have been reported that investigate the relationship between object-oriented design metrics and internal and external product attributes: Emam et al. [5] and Yu et al. [26] present models to predict faulty classes. Briand et al. [3] investigate relationships between design measures and software quality in an student project. These studies use metrics based on information gathered from the source code. Baudry et al. [2] and Genero et al. [7] present studies based on UML models aiming to build models to predict testability respectively maintainability.

Software architecture and design models differ from code in that they exhibit incompleteness and inconsistency. Hence, design metrics for incompleteness and inconsistency might be good predictors for the quality of the resulting systems. The research described in this paper is the first attempt known to the authors that proposes quantification of completeness.

The area of consistency in UML can broadly be divided into the complete approaches and the partial approaches. The complete approaches aim to define a fully formal semantics for the complete UML notation. Notable initiatives in this direction are the ‘precise UML’ project [28] and the ‘Omega’ project [27].

Partial approaches select a subset of the UML notation, typically related to one view such as the process view, and proceed to develop a well-defined meaning for that subset with the aim of automatically identifying inconsistencies in this partial design. Starting point for literature on this

line of research is the workshop on consistency problems in UML [13].

The partial approaches can be divided into two principle types: a formal approach that is concerned with mapping (parts of) UML designs onto formal methods in order to give a meaning to the UML constructs. The second is a design-oriented approach. Here emphasis is on modelling the UML in order to analyse properties of the design. In this approach UML is modelled using a meta-model or using OCL [24]). Subsequently, consistency rules for actual UML designs are defined in terms of the meta-model.

The formal approaches attempt to map partial designs that describe system dynamics such as sequence charts and state charts onto operational formalisms such as process algebra's and Petri-Nets. For example, Engels et al. [6] have presented consistency checking for UML state chart diagrams by providing a mapping onto CSP. McUmber and Cheng provide a mapping of UML onto Promela [16]. An example of a formal approach that formalizes the functional/structural view of UML designs using Z is described by [22].

In the design-oriented approach, Hnatkowska et al. [8] present a number of consistency rules that are defined in terms of the OCL. Also our approach falls in this category. We use a subset of the UML meta-model and formulate consistency rules in terms of that meta-model. As we have shown in the paper, this approach supports the detection of incompleteness and inconsistencies as well as the quantification of completeness, while it does not require the more complex machinery of fully formal methods.

6. Conclusions & Future work

In this paper, we studied the completeness of UML designs. To this end, we firstly defined a collection of rules that capture completeness constraints in terms of a meta-model that covers multiple views. Subsequently, we analyzed the occurrence of violations of these rules in a number of industrial case studies.

In these case studies, significant variations in conformance to and violations of the completeness rules occurred. These case studies showed, that different designers have substantially different amounts of occurrences of violations. Further investigations should address this occurrence in more details and should analyze other factors that cause the variations. Already it is clear that the use of tools for identifying completeness violations can aid in enforcing more uniform design conventions.

The large amount of detected rule violations emphasizes that incompleteness of models cannot be neglected in predicting system quality attributes. We suggest future research to look into the relationship between completeness of UML models and the precision and accuracy of system attributes that are predicted using design metrics.

We expect the degree of completeness to be a useful indicator for controlling and managing the process of software engineering projects. Further research should be done to investigate the relationship between completeness of models and the project progress.

We are continuously collecting additional case studies to build a larger base of reference numbers.

7. References

[1] L.B.V. Basili and W. Melo, *A validation of object oriented design metrics as quality indicators,* IEEE Tr. SE, Vol.22, no. 10, pp. 751-761, 1996

[2] B. Baudry, Y. Le Traon, G. Sunyé. *Testability analysis of UML class diagram*. In Proceedings of Metrics02, pages 54—63, Ottawa, Canada, June 2002

[3] L. Briand, J. Wüst, John W. Daly, V. Porter. *Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems*. Journal of Systems and Software, 51(2000) p 245-273.

[4] S. Chidamber, and C. Kemerer, *A Metrics Suite for Object-Oriented Design*, IEEE Tr. On SE, vol. 20, no. 6, pp. 476-493, 1994

[5] K. El Emam, W. Melo. *The Prediction of Faulty Classes Using Object-Oriented Design Metrics*. Technical Report of the National Research Council of Canada, NRC/ERB-1064. NRC 43609. 1999

[6] G. Engels, J. H. Hausmann, R. Heckel. S. Sauer. *Testing the Consistency of Dynamic UML Diagrams*. IDPT 2002, Pasadena, California. June 2002.

[7] M. Genero, M. Piattini, C. Calero. *Empirical Validation of Class Diagram Metrics*. 2002 Proceedings of IEEE International Symposium on Empirical Software Engineering (ISESE'02)

[8] B. Hnatkowska, Z. Huzar, J. Magott, *Consistency Checking in UML models,* 4th Int. Conf. on Information Systems, Modeling ISM'01, 2001.

[9] I. Jabcobson. *The Object Advantage – Business Process Engineering with Object Technology*. Addison Wesley. ISBN 0-201-42289-1, 1994.

[10] S.H. Kan. *Metrics and Models in Software Quality Engineering*, 2nd Edition. Addison Wesley, 2003.

[11] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, K. Koskimies. *Towards tool assisted UML development environments*. In: 7th Symposium on Programming Language and Software Tools, Szeged, Hungary, June 2001

[12] P. Kruchten. *Architectural Blueprints – the "4+1" View Model of Software Architecture*. IEEE Software 12 (6), pp. 42 – 50, November 1995

[13] L. Kuzniarz, G. Reggio, J. L. Sourrouille, Z. Huzar, M. Staron. *Workshop Materials: 2nd Workshop on Consistency Problems in UML-based Software Development.* UML 2003. Blekinge Institute of Technology (Ronneby, Sweden), Research Report 2003:6

[14] C.F.J. Lange. *Empirical Investigations in Software Architecture Completeness*. Master's Thesis. Eindhoven University of Technology, September 2003.

[15] K.J. Lieberherr and I.Holland. *Assuring Good Style for Object-Oriented Programs*. IEEE Software, September 1989, pages 38-48.

[16] W.E. McUmber and B.H.C. Cheng, *A General Framework for Formalizing UML with Formal Languages*, IEEE, 2001.

[17] N. Medvidovic, D. Rosenblum, J. Robbins, D. Redmiles. *Modeling software architectures in the Unified Modeling Language*. ACM Transactions on Software Engineering and Methodology, Volume 11, pp. 2-57, Issue 1, January 2002.

[18] J. Muskens. *Software Architecture Analysis Tool*. Master's Thesis, Eindhoven University of Technology, Faculty of Mathematics and Computer Science, April 2002.

[19] L. Nenonen, J. Gustafsson, J. Paakki, A. I. Verkamo, Measuring object-oriented software architectures from UML diagrams; In: *Proc. 4th Intl. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, France, June 2000, 87-100.

[20] Object Management Group. *Unified Modeling Language*, specification 1.5. formal/2003-03-01, March 2003.

[21] S.L.Pfleeger, N.E. Fenton. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, 1996.

[22] M. Shroff and R.B. France, *Towards a formalization of UML Class structures in Z*, Proc. of the 21st COMPSAC conference, pp. 646-651, IEEE CS Press, 1997.

[23] D. Soni, R. Nord, C. Hofmeister. *Applied Software Architecture*. Addison Wesley Pub Co., 1st edition, 1999.

[24] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1999

[25] J. Wüst. *SD Metrics – The Software Development Metrics Tool for the UML*. www.sdmetrics.com, v.1.7, October 2003.

[26] P. Yu, T. Systa, H. Müller. *Predicting Fault-Proneness Using OO Metrics: An Industry Case Study*. Proceeding of the Sixth European Conference on Software maintenance and Reengineering (CSMR'02), pp. 99-107, March, 2002.

[27] *Correct Development of Real-Time Embedded Systems Project* http://www-omega.imag.fr/

[28] *The Precise UML Group*. http://www.puml.org/

[29] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.