Teaching Consistency of UML Specifications

Klaas Sikkel, Maya Daneva
University of Twente
Enschede, the Netherlands
e-mail: {k.sikkel, m.daneva}@utwente.nl

Abstract— Consider the situation that you have a data model, a functional model and a process model of a system, perhaps made by different analysts at different times. Are these models consistent with each other? A relevant question in practice – and therefore we think it should also be addressed in our courses. However, UML modelling textbooks don’t discuss it, so we developed our own teaching materials. In this position paper we explain why and how.

I. PROBLEM DESCRIPTION

There are dozens of textbooks that can be used to teach UML modelling to students. All of them assume that you start afresh, designing a new system, and tell you how to specify requirements by means of various types of diagrams. This is useful to learn the techniques, but it creates a narrow idea of what modelling is about. Requirements analysts and software engineers do not always start from scratch! Indeed, most of the times in real life, one is confronted with maintaining or extending existing models and systems. Also, it is not at all unlikely that different partial models have been made by different teams or analysts at different moments in time. In a large project, the data analyst responsible for the data model may not share an office (or even sit on the same floor) with the business process analyst responsible for the process model. Indeed, in large companies, data and process analysts belong more often than not to two different departments within a large IT/IS organization. They also use different tools, modeling standards and model repositories.

As the teachers’ duty is to prepare students for the project realities in their professional lives, we think that a very relevant question in UML courses ought to be:

Given a particular set of UML specifications of a system, can you detect inconsistencies across these specifications?

However, for a decade and a half we did not find a textbook that addresses this question. Current books (that we have seen) still don’t mention it [1–5]. So we made the teaching materials by ourselves. In what follows we give an example.

The point that we want to make is not that we pretend to have the ultimate solution, but that we think this is an issue that ought to be addressed in courses in UML modelling.

II. CONTEXT AND SET-UP OF THE COURSE

The subject of discussion in this position paper is a bachelor course named “Information Systems” that is taught

at the University of Twente, the Netherlands. This course is mandatory in the bachelor programs in Computer Science, Industrial Engineering and Management, and Business Information Technology. It is taught in the 2nd semester of the first year, and yearly some 180 students take part (respectively 60, 90 and 30 for each of the three programs mentioned). The course is partially based on Maciaszek’s textbook [1] and supplemented by our own teaching materials.

The learning objective of the course is to get students master some specific requirements specification skills at basic level. The main skill to be learned is to “translate” a textual description of a desired system into UML.

Students of Industrial Engineering and Management use these and similar techniques (e.g. Business Process Modeling Notation) in a few follow-up courses. The most important skill for most of them is to be able to understand such models. Students of Computer Science continue using UML in more advanced software engineering courses, where UML is also applied at various levels of system design. Students of Business Information Technology follow parts of both programs, as well as a more elaborate course in Requirements Engineering in their second year.

The focus in the introductory Information Systems course is on learning to make UML diagrams, but it must be understood what the purpose is of making them. We position it as one step in a larger process that consists of (possibly multiple iterations of) requirements elicitation, specification and validation. Diagrams add precision and eliminate ambiguity from textual specifications. So they can help to clarify whether the client and the designer really understand each other and understand what is to be implemented.

To that end, the course has the following parts.

A. Learning the modelling techniques

In a series of five lectures and tutorials, students learn the following modelling techniques

- A process model consists of one or more *Activity diagrams* that capture relevant processes in the real world.
- For a functional model two techniques are used. An *event list* (not part of UML, but used as an auxiliary construct) specifies all events, i.e. signals from the outside world that enter the system. Closely related is the *use case diagram* that specifies the functions offered by the system.

- A data model is given by a *class diagram*, which specifies the information about the real world that should be modelled inside the system.
- A behavioural model, finally, is given by a *Statechart diagram* that specifies states for objects of a particular class that the system should be able to distinguish.

In the tutorials and in the exam, students get a text that should be perfectly clear and self-contained. It cannot be subject to discussion, due to the logistics of a mass course. But in the second part this constraint is relaxed.

B. Keeping the client in the loop

More recently we have incorporated assignments where the students, in a role play, meet clients with deeper domain knowledge. In the first assignment a pair of students gets an existing process model. In collaboration with a client representative they have to check whether any requirements are missing in the specification. In the second assignment, a pair of students gets a textual specification that contains some ambiguities and unclear statements and may lack useful background information. After drafting a first data model, they have a meeting with client representatives with more background knowledge. Subsequently they have to improve the model.

There is one lecture/tutorial on validation in order to prepare the students for these assignments.

C. Checking consistency

One lecture and one tutorial are devoted to this, the details are given below.

D. Examination

After a final lecture, in which we look back on the big picture, there is a written exam for parts A and C. The assignments in part B also contribute to the final grade.

III. CHECKING CONSISTENCY

Consistency of UML diagrams is known to be a difficult issue, and there are a number of different approaches to this; see, e.g. Engels et al. [6] and Huzar et al. [7]. More recently, Model Driven Engineering has become a focus of interest. This is treated in specialized master courses in Computer Science.

In a course at introductory level, we also need an answer at introductory level. In the Information Systems course, UML serves requirements capture, more than system design. Therefore, we look at *horizontal* consistency (between different models at the same level of abstraction) and leave out *vertical* consistency (between models at different levels of abstraction). In order to derive a workable technique that can be taught and examined in a mass course, we make some further simplifications. There are various kinds of inconsistencies that we do not check, e.g. synonyms and homonyms, differences in levels of aggregation.

The aspect we focus on is *incompleteness*: in some of the models, some elements are missing. The task consists of a series of pairwise comparisons of two diagrams. In each case, the student has to determine, based on the information

in one diagram, which elements have to be added to the other diagram.

As we teach 5 types of diagrams, there is a total of 20 possible types of comparison (completing a class diagram based on a statechart diagram, completing a statechart diagram based on a class diagram, etc.). In the lecture and the tutorial, the types of comparison are reduced to 9, as shown in Figure 1.

Figure 1. Nine types of pairwise comparisons between activity diagram (AD), event list (EL), use case diagram (UCD), class diagram (CD) and statechart diagram (SD)

Each arrow in Figure 1 has a source diagram and a destination diagram. In a systematic comparison, one walks through all elements of the source diagram and applies a set of rules in order to determine whether something could be missing in the destination diagram.

Below we demonstrate a systematic comparison using (g) in Figure 1 as an example. The teaching activity thus is: Given a class diagram, check the use case diagram for completeness.

For comparison (g) we can fall back on the well-known ‘CRUD’ rules (create, read, update, delete). Are all applicable operations possible for each element of the class diagram? This is spelled out in the set of completeness rules shown in Figure 2.

<p>(g) <i>Check Use Case Diagram by means of Class Diagram</i></p> <ol style="list-style-type: none"> (1) For every class there is a use case that creates (and, if applicable deletes) objects of this class (2) For every association there is a use case that creates (and, if applicable, deletes), instances of the association between objects of the associated classes (3) For every attribute in every class there is a use case that reads (and, if applicable, updates) this attribute
--

Figure 2. Completeness rules for comparison (g)

Figure 3 shows a small fragment of a class diagram from a case history about a yacht rental company. Figure 4 shows the use case diagram, of which the completeness has to be checked. We make the note that the use case diagram notation is a local variant slightly deviating from the UML standard [8, 9].

Figure 3. Fragment of a class diagram

Figure 4. Possibly incomplete use case diagram

Application of rule 1 reveals that rental contracts are created by the use case “Generate contract” but there is no way in which yacht objects can ever be created! In case of doubt, the students can check the case history, and indeed they will find that there should have been a use case to manage yacht data.

Somewhat less obvious is the second omission in the use case diagram. The return date of a yacht can differ from the end date of the contract (e.g. due to weather conditions). But application of rule 3 shows that there is no use case for updating the return date. Indeed from the case history we can infer that there should have been a use case in which a Yacht Rental employee reports that a yacht has been returned.

The CRUD rules are specific for comparisons (g) and (h). The rules for other comparisons are based on similar principles of simple logic. For example in comparison (i), there must be attributes and/or associated classes in the class diagram to store the data associated with transitions in the statechart diagram, so that it is always possible to determine the state that an object is in.

The teaching materials consist of a slide presentation of the motivation for doing this, plus the rules for each comparison with some examples. In exercises and exams,

students get an answer form with predefined tables to report missing elements in a model; they are not requested to adapt the diagrams themselves. The product of a review should be a list of follow-up actions, not a revised specification [10]. Also, if you would adapt the diagrams, you’d have to redo part of the consistency checks until a stable point is reached.

IV. DISCUSSION AND CONCLUSIONS

We think that paying attention to consistency checking helps to put into perspective the role of specifications in real projects. What we hope to achieve in this way is more on the level of attitude than on the level of skill: awareness that specifications are not infallible truths, but the products of human activity in specific settings, with all their potential shortcomings. In order to achieve that, hands-on experience should have a more lasting effect than just pointing out in lectures that consistency is an issue.

Are we the only ones to think so (as we could infer from the absence of this subject in textbooks) or is there a wider interest in this subject? If so, perhaps we could share our interests.

ACKNOWLEDGEMENTS

We are indebted to our (retired) colleague Rob van de Weg. Motivated by his industrial experience he pioneered this approach to consistency checking.

We thank the anonymous referees for constructive comments.

REFERENCES

- [1] L.A. Maciaszek, Requirements Analysis and System Design, 3rd Edition, Pearson, Harlow, UK, 2007.
- [2] M. Blaha and J. Rumbaugh, Object-Oriented Modeling and Design with UML, 2nd Edition, Pearson Harlow, UK, 2005.
- [3] J. Deacon, Object-Oriented Analysis and Design – A Pragmatic Approach. Pearson, Harlow, UK, 2005.
- [4] S. Bennett, S. Mcrobb, and R. Farmer, Object-Oriented Systems Analysis and Design, 3rd Edition, McGraww-Hill, Maidenhead, UK, 2006.
- [5] M. Priestley, Practical Object Oriented Design with UML, 2nd Edition, McGraw-Hill, Maidenhead, UK, 2003.
- [6] G. Engels, J.H. Hausmann, R. Heckel, S. Sauer, “Testing the Consistency of Dynamic UML Diagrams,” Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA, 2002.
- [7] Z. Huzar, L. Kuzniarz, G. Reggio, J.L. Sourrouille, “Consistency Problems in UML-Based Software Development,” in UML Modeling Languages and Applications, N. Jardim Nunes et al. (Eds.), LNCS 3297, Springer-Verlag, Berlin, 2005, pp. 1–12.
- [8] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modelling Language Reference Manual, Addison-Wesley, Reading, MA, 1999.
- [9] M. Fowler, UML Distilled, 2nd edition, Addison-Wesley, Reading, MA, 2000.
- [10] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques, Wiley, Chichester, UK, 1998.