

OCB-Based Automated Validation Method for UML Specifications

L. Ol'khovich and D. V. Koznov

St. Petersburg State University, St. Petersburg, Russia

E-mail: leo@tepcom.ru; oleo@rbcmil.ru; dim@dk12687.spb.edu; dim@tepcom.ru

Received June 9, 2003

Abstract—The paper is devoted to the validation of visual software models. The aim of the validation is to verify whether the models match the visual project language, which is a dialect of a standard visual modeling language (UML, SDL, etc.) created with regard to specific features of the particular project on software development. In the paper, an architecture of a validator designed for both interactive and batch modes is suggested. An approach to specifying visual project languages by means of OCL constraints imposed on the UML meta-model with the subsequent automated generation of validators in accordance with the architecture described is suggested. Results of approbation of the validator and the validator generator in the framework of an industry project are presented.

1. INTRODUCTION

In this paper, we suggest a method for the validation of visual models implemented by means of UML [1]. The purpose of the validation is to check whether specifications meet accepted agreements on using UML in the framework of a given project.

So far as visual modeling is used as a communication means between people, there is no need to fix these agreements, since such a communication admits the existence of diagrams based on commonly used notation, which can be understood intuitively.

However, when UML specifications are to be subjected to automated processing, such as code or text generation, and the like, the requirements for the formalization level of such agreements and for the correspondence of specifications to these agreements are considerably higher. In what follows, these agreements are referred to as visual project languages, to stress the importance of the formalization of these languages when using visual modeling.

The difference between visual project languages and the base visual language is even greater than that between a standard programming language and its project-specific dialects. In the latter case, the dialects include specific rules for naming identifiers, regulations of using a certain language subset, special macros, and the like. However, all these agreements do not change the semantics and syntax of the base language, and they are processed by a compiler in a standard way. The fulfillment of these agreements is checked, as a rule, by a human, without using special automation tools.¹

¹ It should be noted that, in reengineering and reverse engineering, the reconstruction of such dialects is a challenging problem [2].

It has been shown in [3] that the efficient automated processing of visual specifications is possible only with the use of project-specific expedients (technological solutions), i.e., with the use of a set of standard visual modeling means (CASE packages, languages, methods) adjusted to solving tasks specific to a given project and a project-specific software, which extends the functionality of the standard software. The development of means for the automated validation of a visual project language is to be a part of the creation of a technological solution.

In this paper, we suggest a method for the validation of visual specifications that is based on OCL (Object Constraint Language) specifications of a visual project language and on the subsequent automatic generation of a checking module. The latter can be built in a code generator for the sake of error diagnosis or integrated in the assembling and regressive testing means. The validator generation means has an open architecture and admits integration with any CASE package supporting UML.

In the paper, we discuss the architecture and prototype of a tool (validator) that makes it possible to generate checking modules for various visual project languages. Results of the method and software approbation on a telecommunication project of ZAO Lanit-Tepkom are discussed and analyzed.

2. A SURVEY OF APPROACHES TO CHECKING INTEGRITY OF VISUAL SPECIFICATIONS

2.1. Consistency of Different Models

In a number of approaches, the problem of the integrity of visual specifications is viewed as that of consistency of different model types. A detailed survey of var-

ious approaches to the consistency problem in UML models is given in [4]. The classification of possible inconsistencies arising in the evolving of UML specifications is given in [5]. An algebraic approach to this problem is discussed in [4]. In [6], methods for checking the correspondence between state charts and sequence diagrams by means of automated proving systems are discussed. The correspondence between UML state and transition charts and diagrams of the extended finite automaton SDL is implemented in [7]; it is based on a modification of both formalisms and development of a common metamodel.

All approaches listed above improve the consistency of the diagrams; however, they do not take into account specific features of a visual language used in a particular project.

2.2. The OCL Language

In 1994, in the framework of Syntropy approach [8], a new text language for refining visual specifications was suggested. However, this language, based on mathematical abstractions, turned out too complicated. The IBM Corporation adapted the Syntropy approach and created the OCL language [1] based on it; in 1997, OCL was included in the UML 1.1 standard.

The OCL language is designed for the specification of additional information, which cannot be described graphically. There exist several approaches to the automation of the processing of the OCL specifications:

(1) Automated check of integrity of UML models. By means of OCL, requirements on the compatibility of various UML entities in the project are specified. For example, the correspondence of values of object attributes in the cooperation diagrams to the constraints for the corresponding classes in the class diagrams is checked. This approach is implemented in the framework of the integration of the package Dresden OCL Toolkit [9] with the CASE package ArgoUML [10], in the project KeY [11] for the CASE tool TogetherCC [12], and for the CASE Package Elixir [13].

(2) Use of OCL specifications as a source of additional information when generating an executable code: Dresden OCL Toolkit [9]. OCL for Java [14], and RTEEM [15].

(3) Specification of constraints on the UML model. This approach is partially implemented in the CASE package ROCASE [12]. However, the solution reported is not portable to other CASE packages and cannot be used without the given CASE package in the batch mode.

Currently, the last approach is rapidly developing in connection with the creation and support of UML profiles.

2.3. UML Profiles

Starting from version 1.4, the notion of a UML profile—a UML release created by means of UML enhancement means, which takes into account specific features of a certain subject domain—has been introduced in the UML standard [2]. UML 1.4 includes the profiles Unified Process and Business Modeling. There exist also some other profiles, which, however, are not included in the standard.

The application of OCL to the creation of a special profile designed for specifying the arch-type and deltas [17] of the software represented as a family of software products [18] is discussed in [16]. However, no validation is provided in this approach; the user is assumed to manually create the corresponding module.

The company Objectteering Software developed a tool for creating various profiles in the framework of the CASE package Objectteering UML Modeler [19]. However, this tool uses a special built-in language J rather than OCL.

It should be emphasized that there is a fundamental difference between the UML profiles and visual project languages. Namely, the profiles are specific to the subject domains and do not suggest changes in UML in a particular project. Then, it follows that the automated support systems for them are based on dialog means built in particular CASE packages (see, for example, [19]), which cannot be used with other CASE packages.

Visual project languages, which include all UML changes, are used more frequently than the profiles. An automated support system for such a language must (i) “understand” formal specifications of the visual project language, (ii) have both dialog and batch interfaces, and (iii) be easily adapted to various CASE packages.

3. VALIDATOR ARCHITECTURE

A validator is a separate application interacting with the repository of a CASE package. It can operate in two—batch and interactive—modes. Let us consider the second mode in more detail.

In the interactive mode, the user is provided with a graphic interface, which allows him to adjust and run the validation process and to look through information about errors found. There is a possibility to open/create a project (a set of OCL constraints designed for the validation), to choose a CASE package, and to indicate the path to its depository, where the UML model being validated is stored. It is also possible to specify additional information required to run the validation.

The constraints are checked in the batch mode. When the validation is completed, the user is provided with the information about errors found, which can be looked through in a special window. This information can also be written in a file.

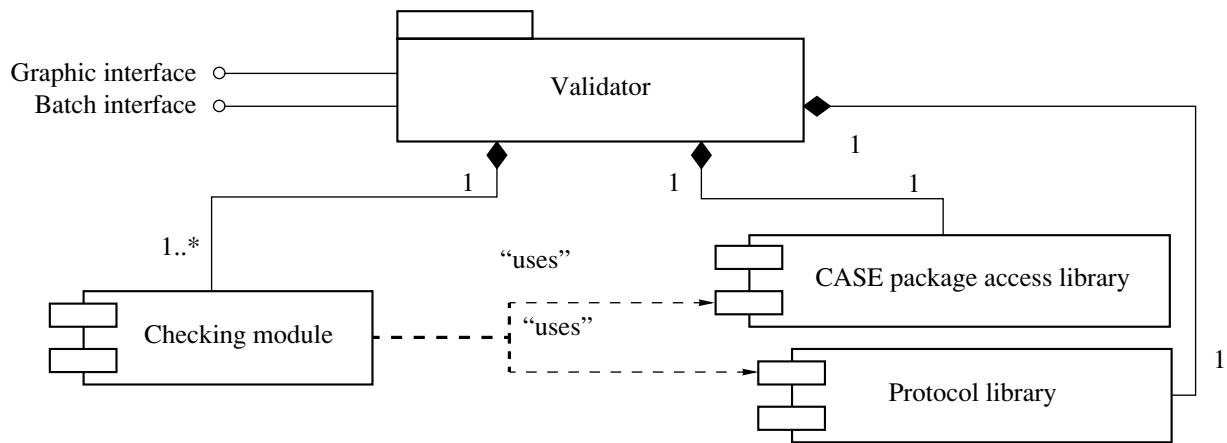


Fig. 1. Validator architecture.

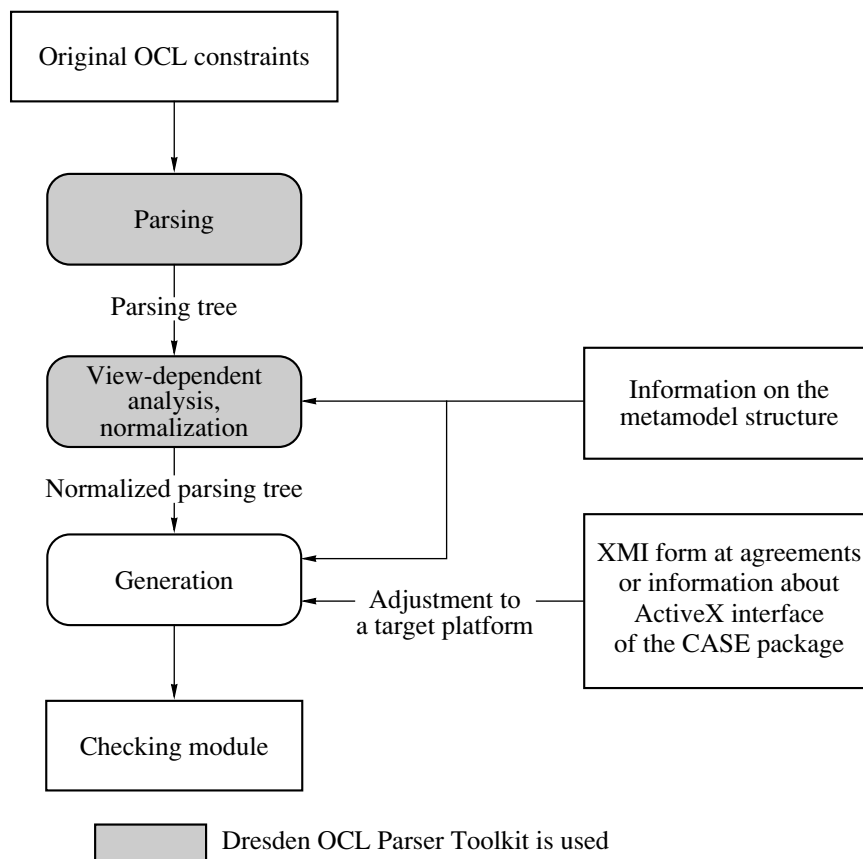


Fig. 2. A schematic of the validator generation.

The validator consists of the following modules (Fig. 1):

- user's interface (both batch and graphical) support module,
- the CASE package access library,
- a protocol library, and
- a checking module.

The validation is carried out by means of the checking module, which is generated automatically by the OCL specifications.

4. GENERATION OF THE CHECKING MODULE

A schematic of the checking module generation is shown in Fig. 2. The validator contains a special editor

for creating OCL specifications. The original text of OCL constraints is subjected to syntactic and view-dependent analysis (by means of the package Dresden OCL Parser Toolkit [9]). The analysis is based on the information about the metamodel (its entities, attributes, and links).

The approach is not specific to UML; it can be used with any visual language with a UML-like structure that admits the use of OCL for imposing constraints on its constructs. In the course of the approbation of our method, the language REAL [20] was used, in which the UML class diagrams were integrated with an extended finite automaton of the SDL language.

Then, the code of the validator checking module is generated. The validator either works with an XMI representation of the UML model or uses ActiveX interface of the CASE package. It uses information about the correspondence of names and types of the metamodel (i.e., the names contained in the OCL specifications) to the XMI tags or to the elements of the ActiveX interface. The support of both methods for accessing the validated model is reasonable because not all known CASE packages support currently the XMI format.

The module obtained is compiled and assembled with the protocol libraries and CASE package access libraries (Fig. 1) resulting in a ready-to-use executable module.

In contrast to the method discussed in [21], in our approach, an executable code is generated by the OCL specification, which is not interpreted. This considerably improves the performance of the validation process in the batch mode.

5. EXAMPLES OF VALIDATED PROPERTIES

Below are three examples in OCL determining constraints on the UML metamodel that require automated checking.

Example 1. The prohibition of using inheritance for creating descendants of nonabstract classes (due to specific features of the database generation in the technological solution REAL-IT [22]):

```
context Generalization
inv: parent.isAbstract
```

Example 2. The prohibition of using the “many-to-many” relation, which is usually imposed on a data model in the case of the automated code generation in SQL/DDL:

```
cocontext Association
inv: self.allConnections ->select(
  multiplicityRange -> select(upper >
1) ->
  notEmpty).size < 2
```

Example 3. The requirement that all classes contained in a package with the stereotype “Persistent”

also must have the stereotype “Persistent” (available in the technological solution REAL-IT):

```
cocontext Package
inv: self.stereotype.name.body = 'Persistent'

implies allOwnedElements ->
  forAll (item: ModelElement |
    item.ocIsTypeOf(Class))
implies item.stereotype.name.body =
'Persistent')
```

6. APPROBATION

The suggested validation method has been approbated in the framework of the telecommunication project provided by the ZAO Lanit-Tepkom. For modeling system structure and event-oriented algorithms in the project, the CASE package REAL was used. A checking module for the validation of eleven different OCL constraints has been created. Its operation time on the project repository was about 30–40 minutes for the processor Intel Celeron-900 MHz. The validator was run in the batch mode every night for one month when a current functional block of the system was implemented. The use of the validator made it possible to find 16 errors.

By the results of the approbation, the following conclusions on the efficient use of this validation method have been derived:

1. The method is efficient when a longstanding modeling process precedes the automated processing of the visual model. In this situation, it makes sense to tune the base line [23] as soon as possible and to keep it working by regularly running the validator in the batch mode. Otherwise, as our experience shows, the adjustment of the generation process, compilation of the generated code, and debugging become too complicated.

2. The method is efficient when, in the course of the development, the model is considerably modified, for example, as a result of the implementation of additional functionality. In this situation, modifications can be introduced by somebody who is not the author of the corresponding model fragment; therefore, it is important to reveal violations of the model integrity as soon as possible.

3. The method is not efficient when local modifications are introduced in the model by the authors. The number of such modifications, as a rule, is not large, and our experience shows that they almost do not contain errors.

4. The method is not efficient when one works with a program code located in a model (through some insertions written in the target programming language or as calls of the corresponding procedures) rather than with the model itself.

7. CONCLUSIONS

Currently, we have implemented the support for the majority of the OCL constructs. The validator has been integrated with the CASE package REAL. We plan to implement the complete support for OCL and to integrate the validator with other CASE packages (including the SDT product by Telelogic AB, which implements the SDL language). We also plan to study various scenarios of the interactive operation of the validator in the framework of the CASE package.

ACKNOWLEDGMENTS

This work was supported by the Russian Ministry of Education, project no. PD02-2.8-145.

REFERENCES

1. *OMG Unified Modeling Language Specification*, Version 1.4, <http://www.omg.org>.2002.
2. Boulychev, D., Koznov, D., and Terekhov, A.A., On Project-Specific Languages and Their Application in Reengineering, *Proc. of Conf. on Software Maintenance and Reengineering*, Budapest, 2002.
3. Koznov, D.V., Visual Modeling of Component Software, *Cand. Sci. (Phys.-Math.) Dissertation*, St. Petersburg: St. Petersburg State Univ., 2000.
4. Asteziano, E. and Reggio, G., An Attempt at Analyzing the Consistency Problems in the UML from a Classical Algebraic Viewpoint, *Proc. of Workshop on Consistency Problems in UML-Based Software Development*, Dresden, 2002, <http://www.ipd.bth.se/uml2002/>.
5. Egeyd, A., Heterogeneous View Integration and Its Automation, *Tech. Rep. 90089-0781*, Center for Software Engineering, Univ. of Southern California, Los Angeles, 1999.
6. Tsiolakis, A., Integrating Model Information in UML Sequence Diagrams, *Electronic Notes Theor. Comput. Sci.*, 2001, vol. 50, no. 3 (*Proc. of the Satellite Workshops of the 28th ICALP (GT-VMT 2001)*).
7. Ivanov, A.N., Koznov, D.V., and Murashova, T.S., RTST Behavioral Model, *Zap. Seminara Kafedry Sistemnogo Programirovaniya "CASE Means RTST++"*, no. 1, St. Petersburg: St. Petersburg State Univ., 1998, pp. 37–49.
8. Cook, S. and Daniels, J., *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, 1994.
9. Hussmann, H., Demuth, B., and Finger, F., Modular Architecture for a Toolset Supporting OCL, *UML2000—The Unified Modeling Language: Advancing the Standard, Proc. of the Third Int. Conf.*, Evans, A., Kent, S., and Selic, B., Eds., York, UK, 2000, Lecture Notes in Computer Science, vol. 1939, pp. 278–293, Berlin: Springer, 2000.
10. Robbins, J.E., Hilbert, D.M., and Redmiles, D.F., Argo: A Design Environment for Evolving Software Architectures, *Proc. of the 1997 Int. Conf. on Software Eng.*, Boston, 1997, pp. 600–601.
11. Ahrendt, W., Baar, T., Beckert, B., Bubel, L., Giese, M., Hoehnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., and Schmitt, P.H., The KeY Tool, *Tech. Rep.*, Dept. of Computing Sci., Chalmers Univ. and Göteborg Univ.; *Computing Sci.*, 2003, no. 2003-5.
12. Chiorean, D. and Cojocari, D., Impelementation of OCL Support in UML CASE Tools—the ROCASE Experience: Objectives, Proposals, Perspectives, *Proc. of 4th Int. Conf. on Information Systems Modelling, ISM'01*, Hradec nad Moravici, Czech Republic, 2001.
13. *Elixir CASE User Manual*, version 1.2, Elixir Technologies, <http://www.elixirtech.com>.
14. Akehurst, D., *OCL4Java Library*, Canterbury Univ.
15. Murray, D.J. and Parson, D.E., Automated Debugging in Java Using OCL and JDI, *Proc. of the Fourth Int. Workshop on Automated Debugging, AAADeBUG 2000*, Ducasse, M., Ed., Munich, 2000.
16. Monestel, L., Ziadi, T., and Jezequel, J.-M., Product Line Engineering: Product Derivation, *Workshop on Model Driven Architecture and Product Line Eng., SPLC2 Conf.*, San Diego, 2002.
17. Bassett, P., *Framing Software Reuse—Lessons from Real World*, Prentice Hall, 1997.
18. Norton, L.M., SEI's Software Product Line Tenets, *IEEE Software*, 2002, pp. 32–41.
19. *Objectteering Software*, <http://www.objectteering.com/>.
20. Terekhov, A.N., Romanovskii, K.Yu., Koznov, D.V., Dolgov, P.S., and Ivanov, A.N., Real: A Methodology and CASE Means for the Development of Real-Time and Information Systems, *Programmirovaniye*, 1999, no. 5, pp. 44–51.
21. Shen, W., Compton, K., and Huggins, J.K., Validation Method for a UML Model Based on Abstract State Machines, *Proc. of EUROCAST 2001*, pp. 220–223.
22. Kondrat'ev, A.M., CASE Means and Object Databases, *Ob'ektno-orientirovannoe vizual'noe modelirovanie (Object-Oriented Visual Modeling)*, St. Petersburg: St. Petersburg State Univ., 1999, pp. 57–78.
23. Humphrey, W., *Managing the Software Process*, Addison-Wesley, 1989.