Evaluation of the Contract-Aware Software Development Process in a Controlled Experiment

A. Derezińska, P. Ołtarzewski
Institute of Computer Science, Warsaw University of Technology
A.Derezinska@ii.pw.edu.pl

Abstract—Contract-Aware Software Development (CASD) process combines the Model Driven Engineering (MDE) approach with the Design by Contract ideas performed at the modeling level. Software engineering solutions need empirical investigation on the impact of methodology on the developed products. Therefore we have designed and performed a controlled experiment analyzing the crucial parts of the process. We focused on the CASD process specialized for UML models with contracts specified in the Object Constraint Language (OCL). Models with contracts are automatically transformed into C# code. In the experiment different development phases and their products were evaluated. As a result a high consistency between contract specification at a model and a code level was confirmed. The evidences stressed very high requirements on the tool support, and some inconveniences that still limit widespread application of the MDE paradigm.

I. INTRODUCTION

The complexity of information systems puts pressure on the constant verification during the whole system development and evolution process. The correctness rules can be specified at all process stages as contracts and transformed together with other software artifacts. This idea states behind a generic process called Contract-Aware Software Development [1] that combines the Model Driven Engineering (MDE) approach [2] with the Design by Contract ideas [3] performed at the modeling level.

The general goal of the process is quality improvement of the software artifacts, its testability and maintainability. The main potential advantages should be recognition of incorrectness at early stages of software development and preserving of consistency between different levels of software artifacts, supported by benefits of contract utilization at the source level.

However, an open question remains: how the best practices promoted in the process can be introduced in practice. We have therefore performed a controlled experiment focused on evaluation of a CASD process specialized for UML as a modeling notation [4], Object Constraint Language (OCL) [5,6] for contract specification at the model level, and C# with Code Contracts library [7] at the implementation level.

Adequate assessment of the software quality is difficult if the maintenance phase is not taken into account, though we tried to evaluate the artifacts created at each development stage as well as to question the participants of the experiment. The experiment confirmed that a high consistency of contract specification at model and code level can be achieved, but also pointed at the obstacles of the approach.

In the next Section, the background of the experiment is presented. In Section 3 we describe the basic features of the controlled experiment. Results of the experiment are discussed in Section 4. Finally, Section 5 concludes the paper.

II. BACKGROUND

A. Contract-Aware Software Development (CASD) process

Contract-Aware Software Development (CASD) is an approach to a generic process that combines features of the contract-based and model-driven development [1]. A distinguishing feature of the process is encountering of dual artifacts in various phases, from the analysis to the implementation one. Artifacts realizing certain functionality have their corresponding artifacts specifying constraint rules - contracts at a given level of abstraction. Transformations between consecutive process levels should preserve a constraint dependency between the corresponding artifacts.

The CASD process should be adapted in order to implement this general idea. One of possibilities is utilization of models in the UML notation [4]. Furthermore, the models can be specialized with dedicated profiles creating a kind of Domain Specific Language. One of many languages that can be applied for contract definition at models is the Object Constraint Language (OCL) [5,6]. It is a declarative specification language used in the UML definition.

Transformation of models with contracts at the specification level, to the corresponding source code and appropriate contracts is a crucial part of CASD. In the specialized process, discussed here, this transformation together with the profiling of UML models can be supported by T.O.F.I.C. [8]. It is an extension of the IBM Rational Software Architect tool (RSA in short) [9]. A new version of T.O.F.I.C. [1,10] is enhanced with visual facilities for efficient creating of UML models with C#-dedicated profiles. It also supports comprehensive interpretation of OCL constraints (invariants of classes, pre- and post-conditions of operations) and transform them into contracts from the MS Code Contracts library [7]. Code Contracts bring the advantages of design-by-contract programming to .NET programming languages.

The original publication is available at http://link.springer.com/chapter/10.1007/978-3-319-06764-3_45
B. OCL Constraint Transformation

With the following, simple example we illustrate how an OCL constraint is transformed into its corresponding code. In an airport model a FlightList class manages a set of flights. This class is associated with a class Flight. An end of the association is specified with “to many” multiplicity. However, it is required that the list includes at least one flight. It could be expressed with an invariant of the FlightList class stating that an appropriate collection is not empty. The appropriate code of C# is automatically generated using a contract method from the Code Contract library and C# mechanism of delegates (Listing 1). The original OCL constraint is incorporated as a comment into the source code.

```csharp
/*OCL invariant: flight->notEmpty()*/
public partial class FlightList {

    [ContractInvariantMethod]
    private void FlightListInvariant () {
        Contract.Invariant(Ocl.invoke(
            this.flight.set().notEmpty ( ) ;
        ));
    }
}
```

Listing 1. OCL class invariant and its implementation

C. Related Work

Design-by-contract paradigm was proposed and established in the context of programming languages [3]. Here, we would like to benefit from the approach, but the specification effort is shifted to the former phases of the software development.

OCL constraints are commonly used in different MDE approaches. Automatic generation of component skeletons implementing modeled components with OCL constraints is presented in [11]. The approach is focused on Java CORBA as a target platform. In [12] business rules associated with models and written in OCL are statically validated using the Octopus/OCL tool and transformed into Java code by a prototype generation tool – AutoPA.

UML models with OCL specification can also be transformed to another formalism in which they are verified using available solvers, for example with Constraint Satisfaction Problem (CSP) tools [13]. However, there are some limitations and very high requirements on the model and constraints completeness that are usually not met in the industrial practice. Moreover such verification of the modeled contracts could be performed before model to code transfer and would supplement the process discussed in this paper.

Constraint-driven Modelling (CDM), which also uses OCL constraints, focus on generation of constraints from the source model [14], whereas in our approach constraints are used for the software verification and are the object of transformation together with software artifacts.

Implementation of MDE is still an open issue, and convenient tool support is one of its main obstacles [15]. In many CASE tools we can use OCL together with modeling facilities, but only a few of them support code generation, so important for MDE approaches. Comparison of OCL tools can be found in [16] and discussion of code generation capabilities in [10]. The most comprehensive OCL tool is Dresden OCL [17], but it does not work with C#.

III. EXPERIMENTAL DESIGN

The purpose of the experiment was investigation of basic parts of the CASD process. It was mainly focused on the impact of automatic generation of code with contracts into the development process. We also studied the maturity of the T.O.F.I.C. tool support, whether it would be helpful in completing contract modeling and its transformation.

A. Subjects

The experiment was carried out during the laboratories in Advanced Software Engineering (ASE) course in Institute of Computer Science, Warsaw University of Technology. Experiment subjects were sixth-semester students attending the Computer Science bachelor degree studies. During the previous years they had done different courses on object-oriented programming (C++, Java) and the basic course of Software Engineering. During laboratory tutorials of this course they prepared a preliminary project covering all basic types of UML diagrams with support of a CASE tool, and experienced major development activities from requirements to code generation and implementation.

Direct before the experiment, during the ASE course, the participants extended their knowledge about requirement engineering, UML modeling, and application of design patterns at the model level. They also learned about contracts and fundamentals of OCL.

B. Experimental Tasks

During the experiment a system simulating an airport was designed. The system was divided into modules corresponding to different system tasks, such as: general management, flight control, flight schedule, catering, airplane technical service management, luggage service, customs service, airport security, and crew management.

C. Experiment Procedure

The experiment scenario consists of the following main steps:

1) Functional and non-functional requirements were analyzed and specified within a CASE tool.

2) Based on the requirements, use case models were prepared. Specification of use cases was written in the form of predefined structured, textual descriptions. It included, among others, pre- and post-conditions of use cases and their invariants, if applicable. The prepared requirements of a module and use case diagrams with their specification were handed over to other participants. This procedure corresponded to a typical situation, were requirements are prepared by analysts and passed to a design team. Each participant wrote a review of the obtained specification.

3) UML class models were designed to meet the requirements of a module. The subjects were encouraged
to use design patterns in the class model. The models were specialized using C# profiles.

4) Structural models were enhanced with contracts covering class invariants, pre- and post-conditions of non-query operations, and constraints of attributes. The contracts were expressed in OCL.

5) A mapping of structural model to C# code was created. Stereotyped class model with OCL constraints was transformed into the corresponding C# code.

6) The subjects implemented selected functionality of the application with the contracts.

7) The implemented modules were tested using developed unit tests. The tests covered implemented functionality as well as the transformed contracts. Selected tests carried out activities that contradicted rules specified in the contracts. Such tests were verified whether they falsify the contract conditions, i.e. error occurrences were expected.

D. Experiment Outcome

There were three kinds of experiment outcomes. The first type of outcomes consist of intermediate results of the laboratory tasks presented during the consecutive meetings of the experiment and the final results delivered at the end of the semester. The artifacts submitted by each participant included: a requirement description, a use case model, a review of the obtained requirements and of use case specification, a UML project with classes, design patterns and OCL constraints, a C# application with contracts and unit tests.

The second type of outcomes were different software metrics calculated on the direct artifacts mentioned above.

The third type of results stands for the data collected in the survey performed at the end of the experiment.

E. Apparatus and Tools

In this point we listed tools used for experiment realization and its evolution. It should be stressed that the CASD process was not fully supported, especially we did not make automatic transformation from requirements to use cases, or from use case contracts to contracts in a design model. Moreover many backward traceability facilities, foreseen in the process, were not automated.

Software requirements of the system modules were stored and analyzed using the IBM RequisitePro tool with the MS Access data base. The requirement data base was provided as an input to the IBM Rational Software Architect (RSA)[9]. This CASE tool was also used for UML modeling and assisted in application of design patterns. Introduction of OCL constraints was supported by RSA, including constraint parsing facilities.

Model specialization towards C#, as well as transformation of classes with OCL contracts to C# code were realized using T.O.F.I.C. - an extension of RSA [1,8,10]. Final C# applications were developed in MS Visual Studio with support of the Code Contract library [7].

The analysis of UML general models and models defining C# code structure were performed with Eclipse lightweight plug-ins (so called pluglets) implemented for the experiment. The measurement of OCL contracts was also based on an implemented pluglet that checks the OCL constraints and analyses utilization of AST (Abstract Syntax Tree) constructs according to the OCL specification.

In the final C# applications, various software metrics were measured with a static analyzer tool NDepend. The unit tests were run and evaluated using MSTest tool included in the MS Visual Studio. The code coverage of tests was verified with the NCover tool.

F. Survey Procedure

The subjects expressed their opinion and evaluated the experiment in a survey carried out at the end of the experiment. The questions of the survey considered the following issues:

- self-assessment of experiences and knowledge of the object-oriented technology,
- labor intensity of the project,
- estimation of usefulness of the various tools used in the project, especially T.O.F.I.C. options supporting C# and contract modeling and transformation,
- opinion about the tool maturity,
- impact of the contract-aware software development on the code quality, consistency with the specification and task realization time,
- applicability of tools and the methodology to future projects.

The survey ended with an open question about the subject’s attitude to the experiment and observed advantages and disadvantages.

IV. ANALYSIS OF THE EXPERIMENT

A. Requirements and Use Case Models

The use case models were used for description of actor-system interactions and available services. Completeness of requirements as well as correctness and sufficiency of use case models were checked both by the supervisor and the subjects after exchanging of tasks.

A challenging activity in this step of the experiment was introduction of constraints into the textual specification of use cases. As recommended, almost all specifications of use cases included some contracts, written in a structured natural language, associated with logical equations or arithmetical relations if necessary. At this process stage, contracts were only qualitatively assessed. Presence of contracts was checked and whether they were logically correct formulated. However, we could not use a qualitative measure that evaluate the future usability of these use case contracts during the design of the contracts specified at the modeling level.

B. Evaluation of UML Design Models

Evaluation of the UML structural models concentrated on the profiled models specialized to the desired language. The calculated metrics were devoted to complexity of models, number of C# stereotypes, variety of C# structures used for the
model refinement, coverage of UML model elements with the corresponding C# stereotyped model elements.

For the verification reasons, UML models of C# implementation and code mapping were also analyzed with the Software Analyzer module of the RSA.

The class models designed for the implementation were fully stereotyped as C# code models, with the coverage from 98% to 100% of all modeling items. The correctness of association between C# profile items and modeling concepts was partially verified by the T.O.F.I.C. internal methods.

The specialized models were not complex as far as the variety of types is concerned. The mostly used C# concepts specified as stereotyped modeling items were: classes, methods, constructors, destructors, fields, properties, and their accessors. As less commonly used concepts we have observed methods, constructors, destructors, fields, properties, and their specified as stereotyped modeling items were: classes.

C. Evaluation of OCL Contracts

OCL contracts specified at the design level consisted of invariants associated with 60% of classes profiled as C# classes. The pre- and post-conditions of operations were used only for selected operations that modify system states. About 85% of operations stereotyped as C# methods were specified with such contracts.

It should be noted that the structural models of any module were designed by other subject than the one preparing requirements and use case models. In result, the design contracts often do not reflect the contracts from the corresponding use cases. Some conditions were overlooked, but some new were added. This observation confirms the necessity for a tool to support tracing of contracts for all stages of the software development, as suggested in the CASD process.

The application of a specification language, OCL in this context, was new for the subjects, which were familiar with different imperative programing languages but not with declarative ones. We examined the usage of various concepts of the constraint specification language. The contracts were commonly specified with variable expressions, calls to attribute values and operations, and literals especially with null value. The subjects did not used more complicated OCL structures, like a variable definition, conditionals, or general iteration over a collection; probably due to lack of experience with the new language.

However, using only a basic subset of the specification language, the experiment participants were able to formulate the logically correct and meaningful contracts. One advantage of simple contracts was that they were easily to be interpreted and checked. If the contracts, which are a source of the verification code, were erroneous the whole verification activity would be in vain.

D. Code Transformation

In the final C# applications the following software metrics were measured: IL - number of virtual lines of the intermediate language of .NET corresponding to the implementation, LOC - number of logical source code lines, NM - number of methods, NF - number of fields, NT - number of types.

The measurement covered only the parts of the code that could be modified by the subjects, excluding modules of MS Code Contracts and of OCL standard library delivered by T.O.F.I.C. We summarized average metric values for the blank applications, i.e. obtained direct after transformation of model and constraints (Table I). The third column includes average metric values for the applications with completed method bodies for selected functionality.

Evaluating the amount of code that was manually supplemented to an application, we developed an Implementation Factor \( IF(Metric) \) for each metric. It was calculated as a ratio of the difference between a metric of a blank application and its completed equivalent, over the metric in the completed application.

\[
IF(Metric) = \frac{Metric(ComplApp) - Metric(BlankApp)}{Metric(ComplApp)} \tag{1}
\]

The last column of Table I gives an average implementation factor for given metrics. The factor is averaged over the modules implemented by different subjects.

The implementation factor was decreased for some subjects who deleted a part of automatic generated code. There were two reasons of such situations. First, the generated code was not necessary in the implementation of the selected functionality. Second, the idea was not properly modeled, and the valid implementation was given in the code without updating the corresponding model. The second case points out the necessity of backwards traceability (from the code to models), which is still not sufficiently tool supported.

Taking into account the implemented functionality, at least one-third of the code of the final applications was automatically generated. To this code belonged implementation of all contracts.

Except of the above mentioned basic metrics, the NDepend tool examined some code quality factors. None of such factor was invalidated. The warnings of the code quality concerned a more advanced programming features, such as suggesting of refactoring of the complex operations, introducing a static type, introducing a sealed type for classes without ancestors, etc. Summing up, the transformed code has satisfactory quality and some application shortcomings were caused by the limited experience of participants in C# programming.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Application Blank</th>
<th>Application Completed</th>
<th>Implementation Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL</td>
<td>2649</td>
<td>3668</td>
<td>0.4</td>
</tr>
<tr>
<td>LOC</td>
<td>194</td>
<td>347</td>
<td>0.5</td>
</tr>
<tr>
<td>NM</td>
<td>175</td>
<td>198</td>
<td>0.22</td>
</tr>
<tr>
<td>NF</td>
<td>70</td>
<td>81</td>
<td>0.21</td>
</tr>
<tr>
<td>NT</td>
<td>27</td>
<td>31</td>
<td>0.2</td>
</tr>
</tbody>
</table>
E. Application Testing

Unit tests were implemented for each module. Participants tested and compared the execution of the application without and with contracts. Two kinds of tests were designed:

- **positive tests** that should end with success when the Code Contracts functionality was incorporated,
- **negative tests** that should fail, invalidate a contract, when the Code Contract facility was switched on in the application. These tests could end successfully when the Code Contracts were switched off.

The code of contracts was covered by tests from 64 to 90%. The rest of the application was only covered in 7 to 68%. But it should be noted, that the tests were designed mainly for those parts that were responsible for realization of selected, implemented functionality.

The running of tests gave results that confirm the expectations and showed a correct execution of contracts.

F. Survey Results

All participants of the experiment answered all questions of the survey. The subjects estimated the labor intensity of the tasks comprised in the experiment. This intensity was expressed as a number of hours spent to complete each step. Average values per one participant are shown in Tab. II.

The most of subjects spent a few hours on code modeling or implementation. The high effort (more than 30 hours) was devoted to these activities only in exceptional cases, such as one student who preferred low level programming and in the contrary to his declaration did not know the C# language, or another student who rewrite the whole code in order to apply a desired, special C# library.

The usefulness of different functionalities supported by T.O.F.I.C. and RSA was estimated as a number of a range 0 to 10. The average results are summarized in Table III. The best scores were assigned to the functionalities supporting UML modeling specialized with profiles (first three positions). Stereotypes of the profile can be easily allocated to one or many model elements, or alternatively, model elements with stereotypes are selected from a specialized palette and ready placed in a diagram. Without such tool support the specialized modeling would be a tedious work, hardly acceptable by model developers.

<table>
<thead>
<tr>
<th>TABLE II</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DISTRIBUTION OF ESTIMATED LABOR INTENSITY OF THE PROJECT</strong></td>
</tr>
<tr>
<td><strong>Project phase</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Some support concerning code generation was on average counted as less useful. Although, the opinions were diversified, e.g. separation of compilation units from the non-modified code (7th position) were found very useful by some subjects (mark 8-10), whereas as useless (mark 0-2) by others. Impact of the technology on the project development was evaluated using the following scale: negative impact (-1), neutral (0) and positive (+1). The following three aspects of the development process were taken into account: code quality, its consistency with the specification and a task realization time. The most of experiment participants positively charged compliance of the application with the specification. Noone assessed this feature in a negative way, and the average grade was above 0.6. However, different scores were given to the impact on the application quality: negative, positive and neutral, whereas the negative predominated (-0.125 on average). The most of problems concerned a general MDE practice of code modeling and transformation and not the contract specification. The third factor, impact on the development time, was also differently evaluated, but the overall averaged score was negative (-0.25). The necessity of creating of models extends the whole time in the contrast to direct code development. However the participants admitted that some delays originated from the fact that they used the crucial technologies and their environment (OCL, T.O.F.I.C., Code Contracts) for the first time. During development of a next project some activities could be realized faster.

G. Threats to Validity

While interpreting experiment results several threats to validity should be taken into account [18].

Mono-operation bias is a threat as the experiment was conducted on a single development project.

Threats to external validity are conditions that limit the ability to generalize the results of experiments. The main source of such threat concerns students that were subjects of the experiment. However, students are often accepted as a substitute of more experienced professionals [19]. Moreover,
when new approaches are concerned, the educational level in the reference to a contract specification language or modeling facilities can be comparable to subjects from the industry, which have the same or less comprehensive educational background. The threat was also lowered by cooperating with students from the advanced course, attending the sixth semester of their studies and participated as volunteers in this kind of laboratory. They declared from few to about 80 months (38 of average) of active, usually industrial, experience in program development using OO languages.

In general, all subjects had a similar level of experience working with OO modeling method in UML, specifying contracts in OCL, using the CASE tool and development environment. It was based on the courses and laboratories comprised in the common curriculum. Whereas, the most discrepancy referred to the comprehension level of the C# language and skills of using the MS Visual Studio.

The statistical conclusion is limited by a small number of participants in the experiment (eight person). Therefore we can give some averaged results, but cannot evaluate statistically significant measures.

The survey results were provided to the supervising teacher after the end of the semester, i.e. after the project results had been evaluated. Therefore the collected opinion has no influence on the scores of the course given to the subjects and could be more outspoken.

V. CONCLUSION

Summing up our experiences and opinion of the experiment participants, the main advantages of the approach realized by T.O.F.I.C. can be summarized as follows:

- A structure of a project and C# code features can be precisely represented using appropriate stereotypes.
- Separation of modeling of a project and UML code increases flexibility of the solution and allows specifying various mappings for a single code model.
- Extended GUI covers graphical representation of model elements refined with the stereotypes.
- The refined UML model is validated in the accordance to the C# code generated from the model.
- OCL contracts are handled in models and transformed into calls of the Microsoft Code Contracts library.

The automatic generation of code contracts improves the consistency between specification and the source code. On the other hand, if developers are not involved in the maintenance phase or at least do not consider this activity are less prone to take care on this consistency. The most criticism is directed to the modeling activities. Many subjects believe that the same result can be directly implemented with contracts in a programming language, but without models.

The CASD process and other similar model-driven methodologies need strong tool support. Application of the T.O.F.I.C. tool with the RSA led to the high productivity, but

the solution has still some limitations. Especially helpful was the full integration of the tool with the used modeling environment. Without a user-friendly support at each development step, effective traceability, and full reverse engineering, an MDE approach can hardly be successfully introduced into the industrial projects. Learning of a new technology requires some time and this is also an obstacle.

REFERENCES


