

# Formally Modeling, Executing, and Testing Service-Oriented Systems with UML and OCL

Loli Burgueño<sup>1,2</sup> and Martin Gogolla<sup>3</sup>

<sup>1</sup> Universidad de Málaga, Spain [loli@lcc.uma.es](mailto:loli@lcc.uma.es)

<sup>2</sup> Marbella International University Centre, Spain [loli@miuc.org](mailto:loli@miuc.org)

<sup>3</sup> University of Bremen, Germany [gogolla@informatik.uni-bremen.de](mailto:gogolla@informatik.uni-bremen.de)

**Abstract.** One of the issues that developers of service-oriented systems currently discuss is the lack of practical, but formal modeling notations and tools that can address the many different, important aspects. This paper presents an approach to model structural and behavioral properties of service-oriented systems with UML and OCL models. Essential service-oriented concepts as service request, service provision or orchestration are formally represented by UML concepts. The models can be executed, tested and analyzed. Feedback is given to the developer in terms of the UML and OCL model.

## 1 Introduction

In recent years, service-oriented systems have become increasingly complex. There has been an explosion on the number of services available—either produced within the companies internal development process or provided by third parties—that are integrated into service-oriented applications. Although following the principles of the Service-Oriented Architecture (SOA), this fact of encompassing such a high number of software components makes the task of reasoning about the systems as a whole difficult. Another reality that has a strong impact on the complexity of these applications is that SOA systems are generally distributed and weakly-coupled among themselves.

As for any software to be developed, it has been proved over the years [4,1,3] that the modeling of SOA applications is an essential task. This is the reason why there exists a wide range of tools and frameworks. In our view, there is a lack of practical tools for reasoning about the compositions of the services that service developers, integrators and choreographers build. To the best of our knowledge, current “formal” models for service composition or choreography rely on formalisms such as process algebras, temporal logic or petri nets. These models are useful to analyze some properties, but not so easy to be practically applied from a development perspective. In this sense, a lightweight approach with strong formal foundations could provide easy and cheap formalization of systems. Specially if its models are not only useful from a theoretical perspective but also from a practical point of view.

In this contribution we have decided to make use of the tool USE (UML-based Specification Environment)<sup>4</sup>. Instead of using proprietary modeling techniques,

---

<sup>4</sup> [http://useocl.sourceforge.net/w/index.php/Main\\_Page](http://useocl.sourceforge.net/w/index.php/Main_Page)

USE is based on the Unified Modeling Language (UML) [5] extended with OCL constraints. The main motivation for our decision is the fact that, UML/OCL models can be formal and at the same time they are very useful from a practical point of view, because they can be applied to develop systems in an automated (or semi-automated) way using MDE (Model-Driven Engineering) principles, techniques and tools.

In this paper we present an approach to model structural and behavioral properties of service-oriented systems with UML and OCL models. Essential SOA concepts as service request, service provision or orchestration are formally represented by UML concepts. Behavioral properties are formally described with UML protocol state machines and operation contracts. OCL is applied for making the structure and behavior precise. Our models can be executed and analyzed for consistency, among other properties. Feedback is given to the developer solely in terms of the UML model. There is no need to work with a second verification language. Our approach supports the automatic generation of test scenarios in which, for example, the availability of services or requests can be checked. The consistency of the service model can be proved by constructing test scenarios.

The rest of the paper is organized as follows. Sect. 2 introduces the proposed approach and Sect. 3 concludes and outlines future research lines. Due to space limitations both the background to our work and related work are not discussed in this contribution, but in a full version of this work available in [2].

## 2 Service Modeling, Execution and Testing

This section explains our approach to model service-oriented systems with a case study, a *process* for an Online Test for students: A teacher designs an online test and then requests from a *service provider* to make the test online available; students as *service requester* conduct the test online; the results are recorded and are passed to the teacher for evaluation and result declaration as another service; a second service provider is the examination administration that offers a service to check for the legitimation of the students to participate in the test and record their results.

### 2.1 Case Study: Online Test

Fig. 1 illustrates the basic artifacts for our case study. The left upper part displays a question sheet stating several questions and possible answers for an online multiple choice test that is designed by a teacher and is to be conducted by students. From the sheet, an online form (in the right part of the figure) will be generated where students enter their email address and their answers to the questions. Each student reply will be recorded in an answer sheet (in the left bottom part of the figure) with a line for each student holding the student's email and her answers as well as two evaluation columns indicating the number of achieved points and a list of incorrectly answered questions.

A	B	C
1	Ingested Title	Question Type
2	Simple UML and OCL test	MULTIPLE_CHOICE
3	Simple UML and OCL test	MULTIPLE_CHOICE

D	E	F	G
Begin Choices			
transitions.	associations.	guards.	<<extend>> stereotypes.
Bag{21,22,21}	Bag{21,22}	Set{21,22,21}	Set{21,22}

A	B	C
1	Time stamp	E-Mail-Address
2	29.04.2017 09:12	ada.alewife@blue-uni.de
3	29.04.2017 09:13	bob.baker@blue-uni.de
4	29.04.2017 09:14	cyd.cook@red-uni.es

D	E	F
(2) Set{42,44,42}->collect(i   i div 2) =	Points	WrongAnswers
Bag{21,22}	2	
Set{21,22,21}	0	1,2
Bag{21,22}	1	1

Simple UML and OCL test

QUESTIONS ANSWERS

Simple UML and OCL test

E-Mail Address \*

E-Mail Address

(1) UML class diagrams can include

☐ transitions.
☐ associations.
☐ guards.
☐ <<extend>> stereotypes.

(2) Set{42,44,42}->collect(i | i div 2) =

☐ Bag{21,22,21}
☐ Bag{21,22}
☐ Set{21,22,21}
☐ Set{21,22}

Fig. 1. Artifacts for the case study Online Test.

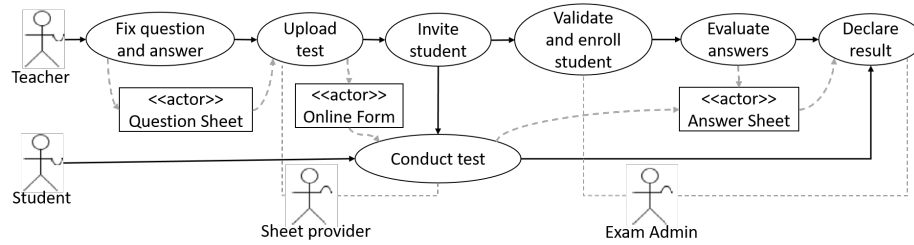


Fig. 2. Use case flow diagram for the Online Test.

The use case diagram-like representation in Fig. 2 gives an overview on the case study and shows the involved actors and use cases as well as the flow between the use cases. In the following text, actors and use cases are indicated using the codetypewriter font. We call this representation a use case flow diagram. We identify two service requesters (**Student**, **Teacher**) and two service providers (**Sheet Provider**, **Exam Admin** as a shortcut for Examination Administration). The process is initiated by the actor **Teacher** through **fixing the Question Sheet** by stating questions and answers. We represent data storages like the **Question Sheet** as (passive) actors. The use case **fix question and answer** may be repeated several times. The **Teacher** then **uploads** the online test. In this use case also the **Sheet Provider** is involved and responsible for transforming the **Question Sheet** into an **Online Form**. The **Teacher** then **invites** some **Students** to participate in the online test. The email addresses of the **Students** have to be **validated** by the **Exam Admin** before they get enrolled for the test. Every **Student** can then **conduct** the test through which the **Sheet Provider** fills the **Answer Sheet**. After closing the test, the **Teacher** can

evaluate the answers and declares the result to the **Students** and to the **Exam Admin**.

## 2.2 Structural and Behavioral Service Modeling

Fig. 3 displays the structural model in form of a UML class diagram for the case study as a screenshot from our tool USE. One identifies four important abstract classes that realize service-oriented concepts: **ServiceRequester**, **ServiceProvider**, **Orchestrated**, and **DataStorage**: (a) the first two abstract classes will be manifested with concrete classes taking the role indicated by the abstract class name (here the service requesters **Teacher** and **Student**, and the service providers **SheetProvider** and **ExamAdmin**); (b) class **Orchestrated** will be used for the orchestration of services; this class will embody protocol state machines (PSMs) that synchronize operation calls touching different requesters and providers; (c) the class **DataStorage** will realize information storages. Please note that different ‘high-level’ concepts from service-orientation (service provision, service request, orchestration, data) are formally realized by the same ‘low-level’, modeling concept (mapping of requests, provisions, orchestrations and data to object-oriented classes). Such a method that maps high-level into low-level concepts is often successfully applied, for example, when an Entity-Relationship database schema is realized by a Relational database schema, in which entities and relationships are mapped to relations.

The structural model is enriched by explicit class invariants that formulate model-specific requirements that must hold when no operation is active; during operation execution invariants may temporarily fail. For the case study, we have implemented some typical invariants (**uniqueName**, **uniqueEMail**, **oneTeacher**, **oneExamAdmin**, **oneSheetProvider** and **Points\_VS\_WrongAnswers**). The implementation of some is shown in the following listing.

```
context Teacher inv uniqueName: Teacher.allInstances->isUnique(name)
context Teacher inv oneTeacher: Teacher.allInstances->size=1
context AnswerSheet inv Points_VS_WrongAnswers:
  Rows->forAll(r|r.Points+r.WrongAnswers->size=r.answers->size)
```

The class diagram in Fig. 3 also shows operation signatures and thus determines part of the behavioral model. In order to distribute the functionality required by the use case flow diagram in Fig. 2 to the individual classes, we have applied the following method: If a class **C1s** participates in a use case **u**, that class will embody an operation **uC** (use case name **u** and **C** being the first letter of the class name) that is responsible for performing the respective actions of the use case on **C1s** objects. For example, the use case **inviteStudent** is realized with the operations **Teacher::inviteStudentT** and **Student::inviteStudentS**. The object initiating the use case performs its own actions and calls the respective operations on the other objects participating in the use case.

Fig. 4 shows central parts of the behavioral model for the case study in form of UML protocol state machines from our tool USE. One can identify in

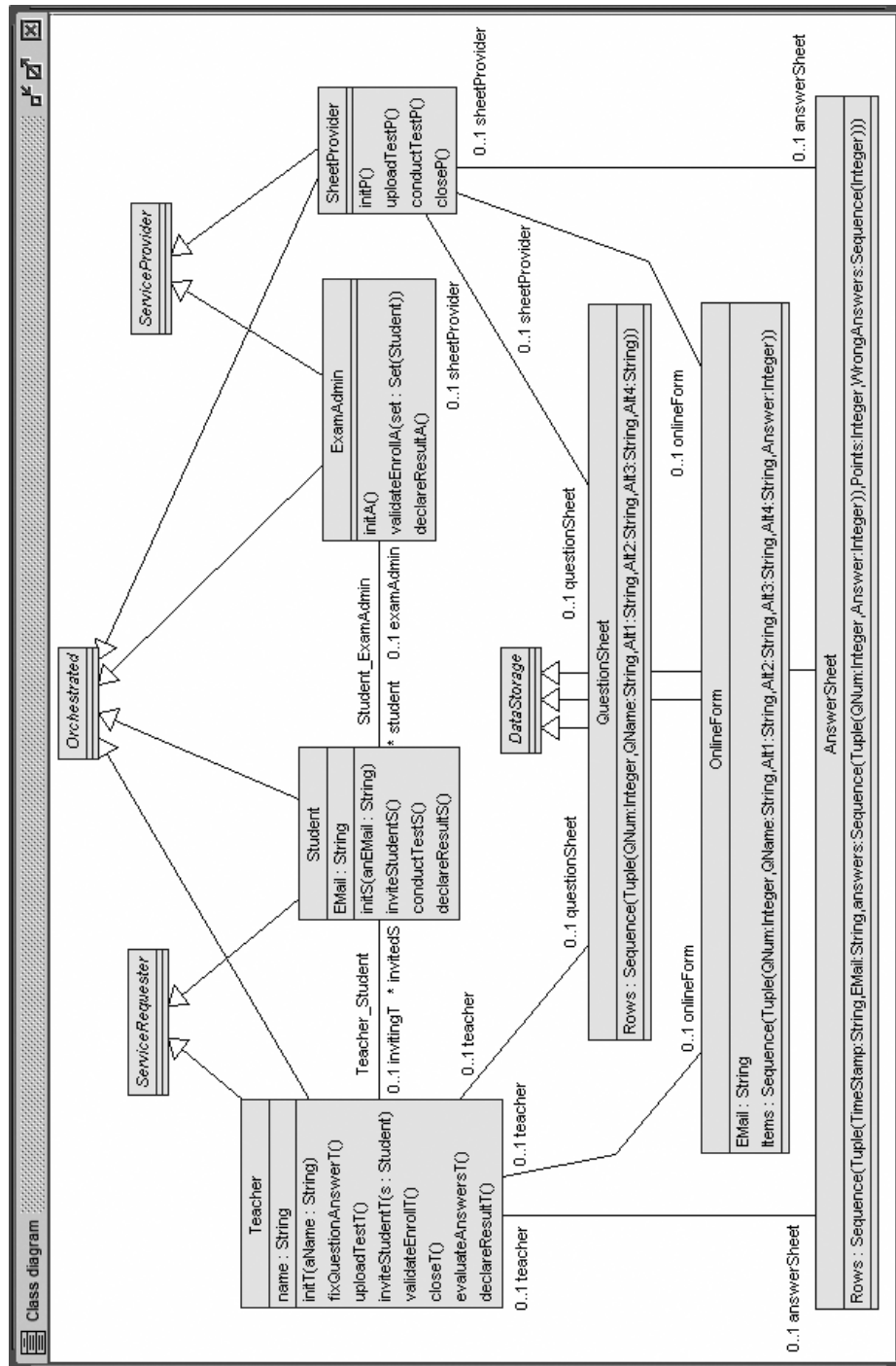


Fig. 3. UML class diagram for the Online Test.

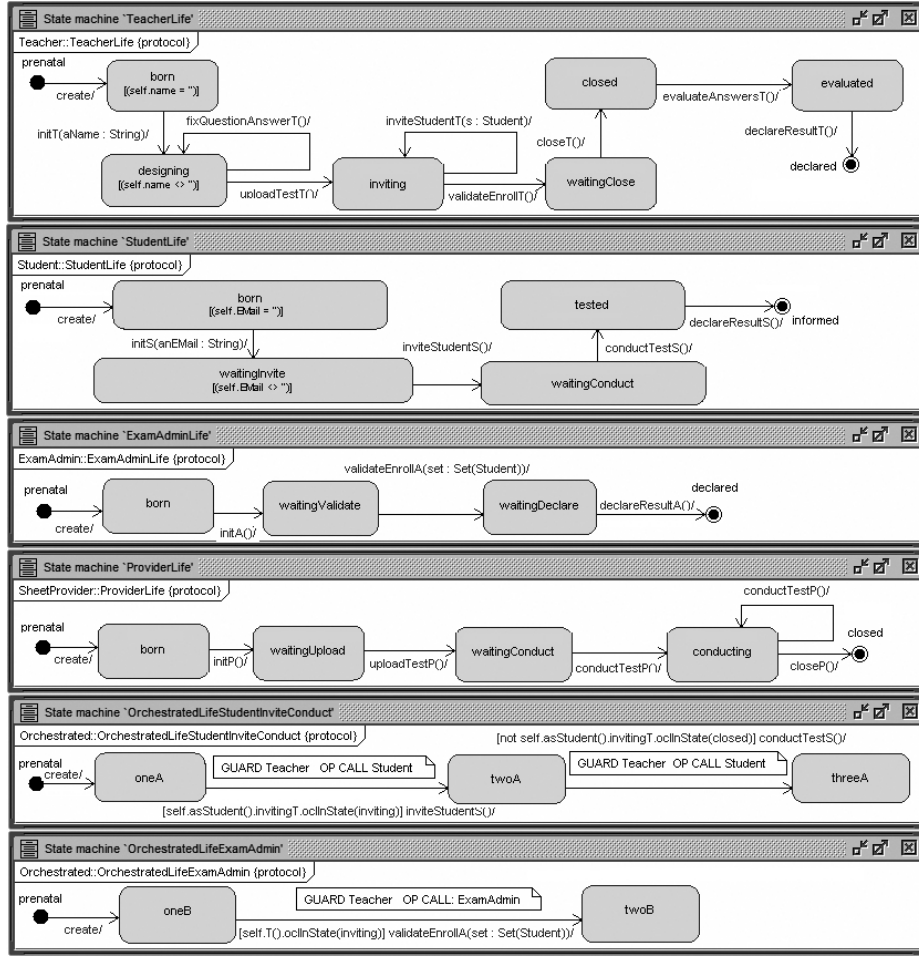


Fig. 4. UML protocol machines for the Online Test.

the top four protocol machines, one for each of the four provider and requester classes. These machines determine the order in which the services are requested or provided, i.e., it is specified in which order the operations from the respective classes are called; only guards and operations from a single class are handled here. The two protocol state machines at the bottom are responsible for the essential orchestration task. These two machines are attached to the abstract class *Orchestrated*, and the behavior restrictions are inherited to the specialized requester and provider classes. *Orchestration* in this context means that conditions (in form of guards) and events (in form of operation calls) from *different* classes are considered. The class referred to in the guard and the class belonging to the operation are different. For example, the next-to-last machine restricts a sequence of `Student::inviteStudentS()` and `Student::conductTestS()` op-

eration calls by guards that refer to the inviting **Teacher** and require that this inviting teacher is in a particular protocol state.

In addition to the protocol state machine, the behavioral model is determined by giving an operation an imperative implementation, which is formulated on the modeling level without going into programming language details and is written in the language SOIL (Simple OCL-like Imperative Language). The behavioral model can be further sharpened by stating the operation effects in a declarative way with operation contracts in form of OCL pre- and postconditions. The operation implementation in terms of SOIL is guided and must respect the operation contracts. Correctness of the operation implementation relative to the operation contracts is checked in USE when test cases are run. As an example we show the implementation and the contract for one operation.

```
Teacher::inviteStudentT(s:Student)
begin
  insert (self,s) into Teacher_Student;
  s.inviteStudentS()
end
pre  studentHasEMail: s.EMail<>null and s.EMail<>''
pre  notInvited: self.invitedS->excludes(s)
post invited: self.invitedS->includes(s)
```

### 2.3 Service Model Execution

Fig. 5 shows an example execution run of the complete model. The twelve executed operations are stated as a listing in the lower right corner. The execution run involves exactly one object from every class, and each operation from every class is called once. Therefore, this run demonstrates that the behavioral service model can be instantiated and that the model is consistent and free from contradictions: All protocol state machines work properly together, and the reached final system state as well as the intermediate system states satisfy the model-inherent constraints (wrt multiplicity) and all explicit invariants from the class diagram; all operation contracts are satisfied. The sequence diagram shows lifelines for the single objects. The operation calls are indicated as message arrows from one lifeline to another lifeline. On the lifelines, the reached protocol states of the respective object are indicated. Thus the development of the objects from one protocol state to the next protocol state can easily be traced.

Due to space limitations, our testing approach to service-oriented systems is not presented in this paper. It is available in our technical report in [2].

## 3 Conclusions and Future Work

This paper presents an approach in which service-oriented systems are modeled using UML in combination with OCL. These models do not only focus on one aspect in service-orientation, but consider requests, provisions, orchestrations and data in a coherent manner. Based on the models of a system, properties

**Fig. 5.** UML sequence diagram for an example execution run.



such as the consistency and instantiability of service request, service provision and service orchestration can be verified by automatically building test scenarios where both processes and data are considered. The application of our approach to practical cases requires developer expertise in UML and OCL. The static system properties must be formulated with UML class diagrams and OCL invariants, and the dynamic properties with UML protocol machines and OCL contracts. The developer is supported in the process by USE in semi-automatically constructing test scenarios.

Currently, the test cases generated by our approach are system states that embody (a) structural aspects in form of object attributes and links between objects and (b) behavioral aspects in form of object states referring to the dynamic behavior and the orchestration in form of protocol state machines. We can check for the applicability of services, i.e. operations. In the future, we will consider system state sequences with service request and service provisions as transitions in between and properties of such system state sequences. Our plans also include the extension of our approach in order to allow the definition of initial states and support for checking whether a particular generated system state can be reached via service requests/provisions from the initial states. This implies an extension of the model validator in order to handle protocol states. Direct support for concepts like requester, provider, or data will be provided as well. To do so, we will equip such classes with predefined PSMs that can be extended according to the application needs. Furthermore, we will ease the definition of PSMs by allowing the definition of regular expressions over operations and derive PSMs from them. We also plan to provide predefined interfaces among requesters and providers supporting a direct and better communication between them. In order to check the applicability of our approach, firstly, we will work on larger and existing case studies based on real data and; finally, on the integration of our approach with existing SOA systems that needs to be modernized or integrated with others.

**Acknowledgments:** This work has been partially funded by Spanish Research Project TIN2014-52034-R.

## References

1. Barjis, J.: The importance of business process modeling in software systems design. *Science of Computer Programming* **71**(1) (2008) 73 – 87
2. Burgueño, L., Gogolla, M.: Formally modeling, executing, and testing service-oriented systems with UML and OCL. Technical report (2017) <http://www.db.informatik.uni-bremen.de/publications/intern/BG2017.pdf>.
3. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: *Proceedings of the FOSE 2007, IEEE Computer Society* (2007) 37–54
4. Mohammadi, M., Mukhtar, M.: A review of SOA modeling approaches for enterprise information systems. *Procedia Technology* **11** (2013) 794 – 800
5. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual*. Pearson Higher Education (2004)