Teaching Model Views with UML and OCL

Loli Burgueño[1,2], Antonio Vallecillo[1], and Martin Gogolla[3]

[1] Universidad de Málaga, Spain. {loli,av}@lcc.uma.es
[2] Marbella University International Centre, Spain. loli@miuc.org
[3] University of Bremen, Germany. gogolla@informatik.uni-bremen.de

Abstract. The specification of any non-trivial system is normally composed of a set of models. Each model describes a different view of the system, focuses on a particular set of concerns, and uses its own notation. For example, UML defines a set of diagrams for modelling the structure and behavior of any software system. One of the problems we perceived with our students is that they are able to understand each one of these diagrams, but they have problems understanding how they are related, and how the overall system specifications work when composed of a set of views. This paper presents a simple case study that we have developed and successfully used in class, which permits students developing the principal views of a system, simulate them, and check their relations.

1 Introduction

Non-trivial systems specifications need to be decomposed in various models (views), each one focusing on a particular set of concerns and described in a particular notation. For example, UML defines a set of diagrams for modelling a system that permit describing its structure (class diagrams), the behavior of its individual objects (state machines), the collective behavior of collections of objects (communication and sequence diagrams), etc. There are two main problems that we have perceived with our students when teaching the use of models. First, they normally tend to see models as drawings, mostly used for communication among humans, and not as software artefacts that need to be processed by computers. Hence they tend to be less precise and formal than when developing programs. And second, when dealing with multi-viewpoint models, students may be able to understand each one of these diagrams, but they have problems understanding how they are related, and how the overall system specifications actually work when composed of a set of views.

This paper presents a simple case study that we have developed and successfully used in class, which permits students to develop the principal views of a system, simulate them, and understand their relationships. We use a combination of UML and OCL and the possibilities that the OCL/USE tool [2] provides to simulate and analyse the models, as proper software artefacts.

The example we present here—originally defined in [4]—is based on a production line and it is modeled using a combination of UML and OCL. We show how

these notations permit modeling the system in a formal manner using different views, and allow performing several interesting analyses on the system.

The rest of the paper is structured as follows. The next Sect. 2 describes the example and shows interesting properties using the USE modeling environment [2,3]. Section 3 presents the lessons we learned. Finally, Section 4 presents our conclusions and future work.

2 A UML and OCL Model for a Production Line System

Let us illustrate our approach by modeling a plant that produces hammers. This plant is composed of four machines: one that generates hammer heads, another that generates handles, one that assembles heads and handles, and finally one that polishes the hammers. There are also trays that are used to collect the goods in production.

The plant operates as follows: each generating machine has an associated tray, in which it places the heads or handles as soon as they are produced. From these trays, the pieces are taken by an assembler machine, which puts together one head and one handle to produce a hammer, which is placed in a different tray. From this tray, the polisher takes hammers, works on them and leaves them in a different tray.

2.1 Structural Elements

Figure 1 shows in a UML class diagram the structural elements of the system. We can see how each `Tray` has a capacity (`cap`) and each `Machine` keeps record of the time that its job takes (`processingTime`). Futhermore, the head and handle generators (`HeadGenerator` and `HandleGenerator`) keep a counter of the number of elements that have produced (`counter`). In order to perform the actions of placing or removing pieces from the trays, class `Tray` provides `get()` and `put()` operations.

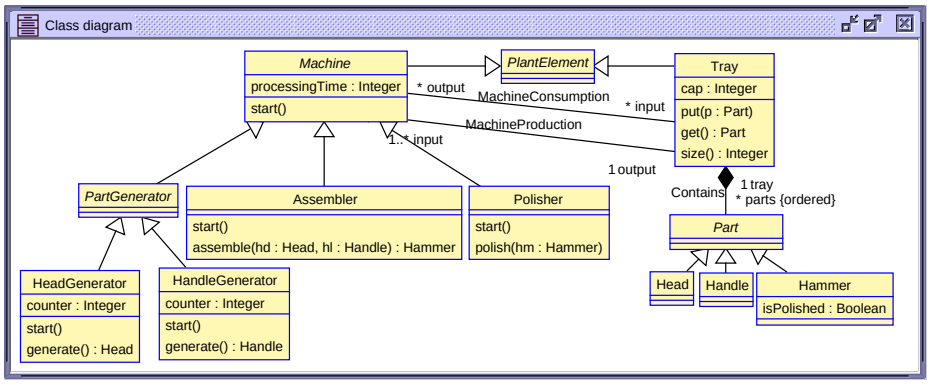

Fig. 1. Class diagram for the production line example.

2.2 Behavioral Elements

Pre- and post-conditions. The behavior of the system can be expressed in UML and OCL by different means. One of the most common ways is by adding the specification of pre- and post-conditions to the operations, defining their behavior. This is shown below for `get()` and `put()` operations of class `Tray`.

```
  put(p:Part)
    pre notFull: self.parts->size()<cap
    post ElementAdded: self.parts=self.parts@pre->append(p)
  get():Part
    pre notEmpty: self.parts->size()>0
    post FirstElementRemoved:
      result=self.parts@pre->at(1) and
      self.parts@pre=self.parts->prepend(result)
```

State machines. UML permits specifying the behavior of individual objects by means of State machines that determine how their internal state changes as a result of the invocation of their provided operations. USE/OCL permits formally specifying state machines for individual objects, and then automatically deriving the corresponding UML diagram. For example, the following listing shows the specification of the *state machine* of a `Tray`. The corresponding UML diagram is shown in Figure 2.

```
psm PutGet
  states
    init: initial
    Empty  [self.parts->size()=0]
    Normal [0<self.parts->size() and self.parts->size()<self.cap]
    Full   [self.parts->size()=self.cap]
  transitions
    init -> Empty { create }
    Empty -> Normal { [self.cap>1] put() }
    Normal -> Normal { [self.parts->size()<cap-1] put() }
    Normal -> Full { [self.cap>1 and self.parts->size()=cap-1] put() }
    Empty -> Full { [self.cap=1] put() }
    Full -> Empty { [self.cap=1] get() }
    Full -> Normal { [self.cap>1] get() }
    Normal -> Normal { [self.cap>1 and self.parts->size()>1] get() }
    Normal -> Empty { [self.parts->size()=1] get() }
end
```

Behavior of operations. USE also permits to specify the behavior of operations using a simple executable language called SOIL [1]. For instance, the behavior of `Assembler::start()` and `Assembler::assemble()` operations can be specified as follows.

```
start()
  begin
    declare hd:Part, hl:Part, hm:Hammer;
    hd:=self.input->select(t|t.parts->size>0 and
          t.parts->forAll(oclIsTypeOf(Head)))->single().get();
    hl:=self.input->select(t|t.parts->size>0 and
          t.parts->forAll(oclIsTypeOf(Handle)))->single().get();
    hm:=self.assemble(hd.oclAsType(Head),hl.oclAsType(Handle));
    self.output.put(hm);
  end
```

```
assemble(hd:Head,hl:Handle):Hammer
  begin
   destroy hd,hl;
   result:=new Hammer; result.isPolished:=false;
  end
```

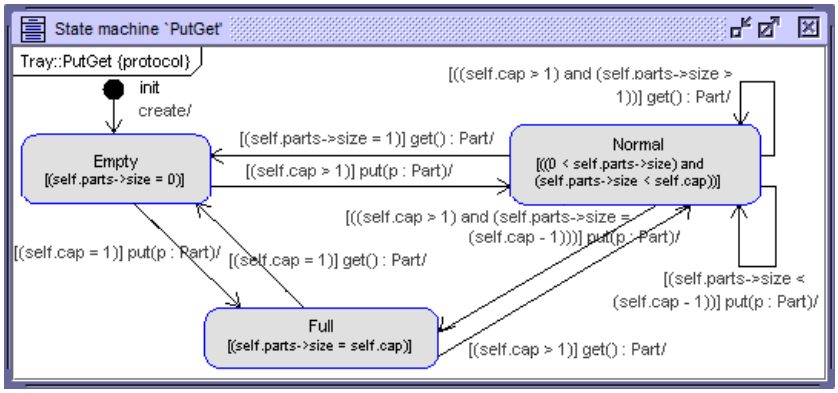

Fig. 2. State Machine for Tray objects.

Executing the specifications. Once we have the behavior of the operations, we can execute the system by providing a sequence of SOIL commands that create the initial objects of the system, their links, and invokes the operations. Based on the specifications above, for example, we can show the students how to simulate the system, by creating an initial model of the system and invoking the `start()` operation to the four machines.

```
-- Machines
!new HandleGenerator('hag')
!new HeadGenerator('heg')
!new Assembler('asm')
!new Polisher('pol')
-- Trays
!new Tray('Handle2Assem')
!Handle2Assem.cap:=4;
!new Tray('Head2Assem')
!Head2Assem.cap:=4;
!new Tray('Assem2Polish')
!Assem2Polish.cap:=4;
!new Tray('Polish2Out')
!Polish2Out.cap:=4;
-- Production Line Connections
!insert (hag,Handle2Assem) into MachineProduction
!insert (heg,Head2Assem) into MachineProduction
!insert (Handle2Assem,asm) into MachineConsumption
!insert (Head2Assem,asm) into MachineConsumption
!insert (asm,Assem2Polish) into MachineProduction
!insert (Assem2Polish,pol) into MachineConsumption
!insert (pol,Polish2Out) into MachineProduction
-- Process
!heg.start()
!hag.start()
!asm.start()
!pol.start()
```

Figure 3 pictorially shows a filmstrip of the behavior of the system as a sequence of snapshots after every operation execution. Aggregation associations are used to visualize 'ownership' between objects (e.g. a part is placed on a tray). We have also developed a video with the complete filmstrip[4].

[4] `http://atenea.lcc.uma.es/Descargas/EduSymp17/3Hammers.mp4?dl=0`

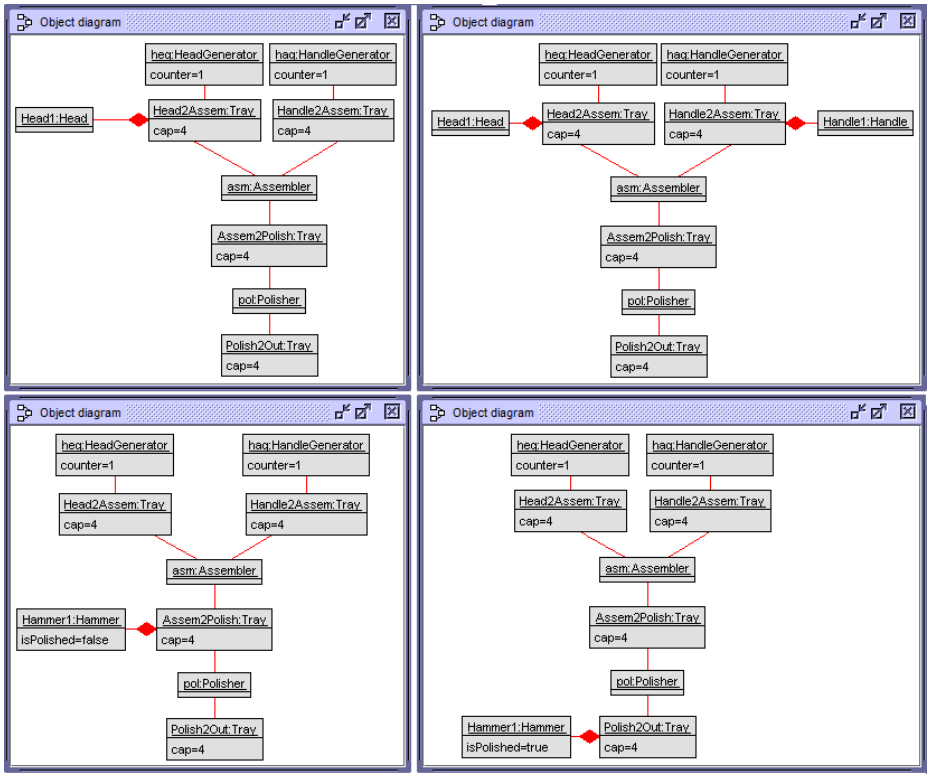

Fig. 3. Object diagram sequence displaying the behavior of the system.

Object interactions. So far we have focused on the structural view of the system and the behaviour of individual objects. UML also permits representing the interactions between objects by means of Sequence and Communication diagrams. With USE/OCL, we are able to automatically derive them from the simulation of the system's behavior described above, in terms of sequences of operations invocations. First, the interactions are recreated using the UML sequence diagram shown in Fig. 4, that is automatically constructed by USE/OCL.

The behavior can also be displayed as a communication diagram as in Fig. 5. Interestingly, for every step we can represent the current states of all the state machines of the Tray objects—not shown here for space reasons, although they are similar to the one depicted in Fig. 2 but with the current state highlighted; the interested reader can see the presentation[5] we have developed to show the state changes. Also, all the code developed for this case study is available[6].

2.3 Further analysis

Once we have the specifications, there are different kinds of analyses that we can perform on the system that show some of the potential advantages of developing

[5] `http://atenea.lcc.uma.es/Descargas/EduSymp17/snapshots-buffer.pdf?dl=0`
[6] `http://atenea.lcc.uma.es/Descargas/EduSymp17/sources.zip?dl=0`

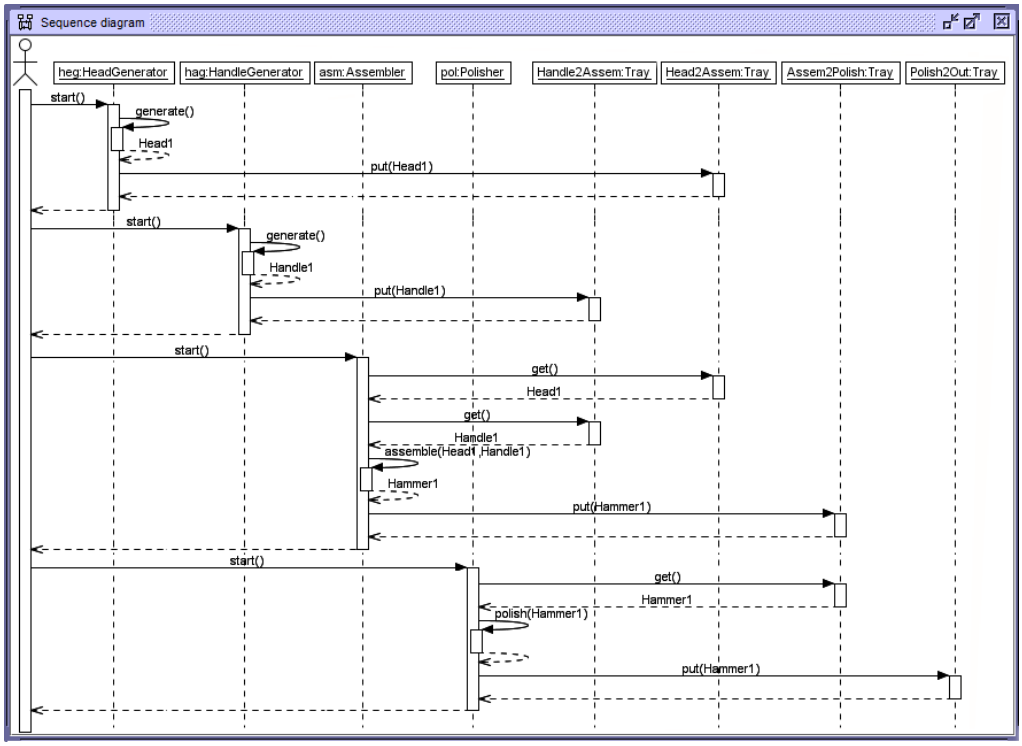

Fig. 4. Sequence diagram.

model-driven system descriptions with UML and OCL. In particular: visualization of complex structures and processes; execution and simulation of scenarios (operation call sequences); checking structural properties of the system within states by OCL queries (e.g. calculating the number of finally produced parts); checking behavioral properties (e.g., testing the executability of an operation by testing its pre-conditions); checking for weakening or strengthening model properties (invariants, contracts, guards) by executing a scenario with modified constraints; proving general properties within the finite search space with the USE model validator, such as structural consistency (i.e. all classes are instantiable); behavioral consistency (i.e. all operations can be executed); deadlocks (e.g. construction of deadlock scenarios due to inadequate buffer capacities), etc.

The relationships between the views can also be explored with this approach. One of the most interesting examples happens when one of the views is changed, and the effects of this change on the rest of the views. For example, when an invariant is added or a class is changed or removed; when an element changes its name; or when a guard for a state change is altered. Every kind of change in a model has a different impact on the validity of another model, and therefore it requires the other model view to be updated. Students can learn from the effects of each change, and understand how views are related.

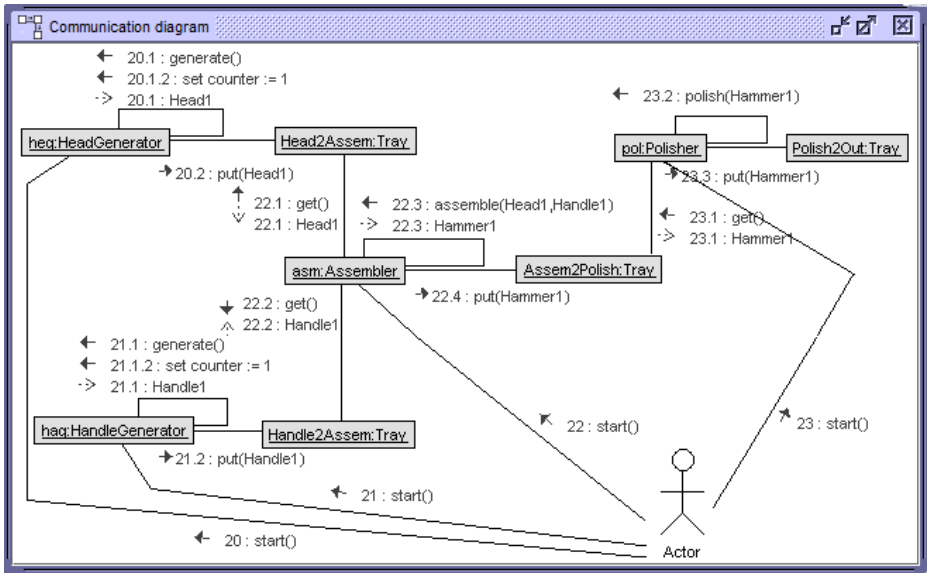

Fig. 5. Communication diagram.

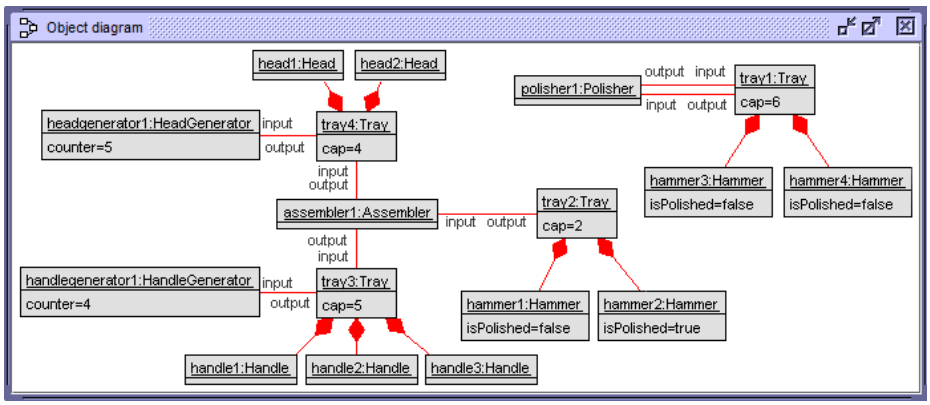

Fig. 6. Generated test case for Producer-Consumer-Tray configuration.

Finally, we want to highlight the importance of running structural tests on the metamodels. One of them concerns their instantiability and their ability to faithfully represent the application domain. For example, we decided to ask the USE model validator [3] to generate a plant configuration using the system metamodel. The resulting object diagram is shown in Fig. 6. Interestingly, the produced system is wrong. The polisher is not connected to any tray that provides it with hammers. This motivates the need to develop additional, currently missing invariants (on the structural system level) and demonstrates the potential usefulness of this approach for validating these kinds of properties which are normally overlooked for being considered obvious.

3 Lessons Learned

Our experience of teaching UML views with USE/OCL has been very positive. At the beginning the students feel uneasy with OCL because they are used to build UML models that are always right (As Bertran Meyer used to say: "Bubbles never crash"). But they progressively discover the advantages of being able to check that the models they are building are correct. Furthermore, they discover the relations between the different views of a system, and how changes in the model manifest in the views.

Another interesting benefit of our approach is that we can follow an incremental development process for building the models. Starting from a simple class diagram we can incrementally add classes, associations, attributes, operations, invariants, contracts, soil operation implementations and protocol state machines. Another feature that students like very much is creating object models. They then discover that models are more than *pictures*, but assertions on the objects that conform a system (and their relationships). This incremental development process supports direct feedback and model improvements. They discover that modelling is similar to programming, in the sense that you write a program and execute it to check whether its behavior is as you expected. Students can check their models using various functionalities available in USE: the diagrams, the evaluation browser, the class extent, the single object window, the class invariants window, etc. And they also learn that views are not completely independent. On the contrary, they are all *projections* of an underlying model.

4 Conclusions

In this paper we have presented a case study modeled with UML and OCL/USE that has been successfully used to teach modeling in class. It does not only focus on the different views of the system but also on the relationships the different models have among them. Furthermore, we disclose several of the advantages of modeling with UML/OCL and USE.

Acknowledgments *This work is partially funded by the Spanish Research Project TIN2014-52034-R. We would like to thank the anonymous reviewers for their valuable comments and suggestions.*

References

1. Büttner, F., Gogolla, M.: On OCL-Based Imperative Languages. Science of Computer Programming **92** (2014) 162–178
2. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comp. Prog. **69** (2007) 27–34
3. Gogolla, M., Hilken, F.: Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In Oberweis, A., Reussner, R., eds.: Proc. Modellierung (MODELLIERUNG'2016), GI, LNI 254 (2016) 203–218
4. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing rule-based behavioral semantics of visual modeling languages with maude. In: Proc. of SLE'08. Volume 5452 of LNCS., Springer (2008) 54–73