

Chapter 8

Animation of OCL Operation Contracts

In this chapter we present an approach to animating OCL operation contracts. By translating OCL constraints to arithmetic formulas with bounded quantifiers and solving these using our techniques presented in [Chapter 5](#), we can perform animation efficiently without relying on additional guidance from the user. We implemented our approach in the tool OCLexec that generates from OCL operation contracts corresponding Java implementations which call a constraint solver at runtime. The generated code can serve as a prototype.

In the sequel, we first demonstrate the benefits of animation by means of a case study. Then we present our animation technique in detail. We describe a preliminary analysis of operation contracts that narrows down the set of classes for which new instances may need to be created and the set of constraints that need to be considered for animation. Moreover, we show how OCL expressions can be mapped to arithmetic formulas with bounded quantifiers. Finally, we give experimental results for our animation tool OCLexec.

8.1 A Case Study

In this section we present an example of a specification that could benefit from animation.

8.1.1 The Task

[Figure 8.1](#) shows an excerpt from a possible UML model of a company. Employees are temporarily assigned to customers to carry out the customers' orders. Customers specify the skills that they would like the employee to have for handling their order (association end `requestedSkills`). Also, employees may give a list of customers that they prefer to work for (attribute `preferredCustomers`).

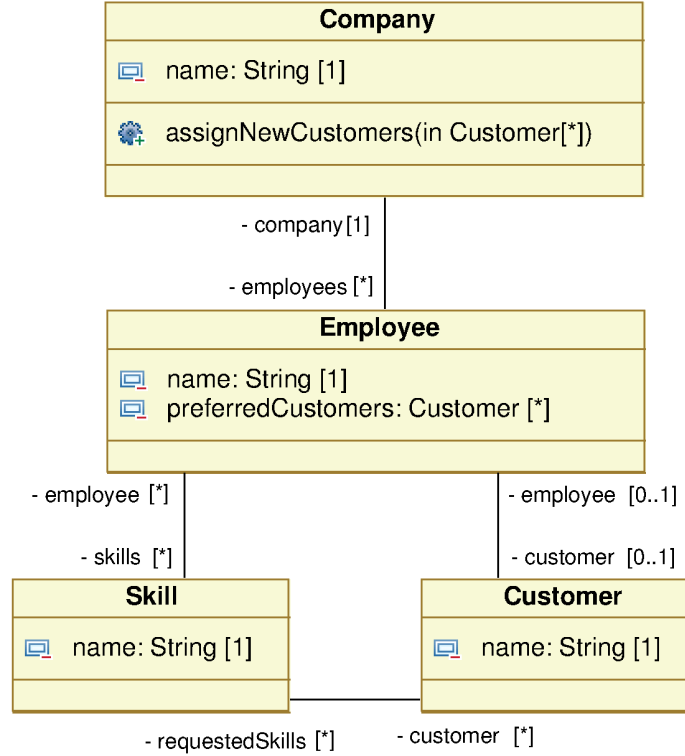


Figure 8.1: Excerpt from a possible UML model of a company

A task of the system which we are specifying is to perform an adequate assignment of employees to customers. What is sought is an assignment that respects the preferences of both the customers and the employees. This kind of assignment problem can be regarded as an instance of the prominent stable marriage problem [68]. The term *stable marriage* is inspired by the idea of matching men to women in a consistent manner. It is well-known that if the numbers of men and women are equal, it is always possible to find a *stable* assignment, i.e., an assignment in which no man and woman leave their assigned partners in order to form a new couple because they both prefer their new partner to the one that was assigned to them.

8.1.2 Anatomy of the Operation Contract

Since the operation contract in Figure 8.2 is nontrivial, we explain why it expresses the requirements. The precondition of the operation contract states that there are at least as many available employees, i.e., employees that are currently not assigned to a customer, as customers that are supposed to be matched. This condition is obviously necessary for the existence of any assignment of available

```

context Company::assignNewCustomers(newCustomers: Set(Customer)):

pre enoughEmployees: employees->select(customer.oclIsUndefined())->size()
    >= newCustomers->size()

post allCustomersAssigned:
    employees@pre->select(customer@pre.oclIsUndefined()
        and not customer.oclIsUndefined())
        ->collect(customer)->asSet() = newCustomers

post assignmentStable:
    employees@pre->select(customer@pre.oclIsUndefined())
        ->forAll(e | newCustomers->forAll(c |
            let
                matchedSkills : Set(Skill)
                    = c.requestedSkills@pre->intersection(c.employee.skills@pre),

                potentialSkills : Set(Skill)
                    = c.requestedSkills@pre->intersection(e.skills@pre)
            in
                (potentialSkills->includesAll(matchedSkills)
                    implies potentialSkills = matchedSkills)
            or
                (e.preferredCustomers@pre->includes(c)
                    implies e.preferredCustomers@pre->includes(e.customer)))

modifies only: employees->select(customer.oclIsUndefined())::customer,
    newCustomers::employee

```

Figure 8.2: Operation contract for assigning new customers to available employees

employees to all new customers. As mentioned above, this condition is also sufficient for the existence of a stable assignment. In the precondition, we use the built-in operation `oclIsUndefined` for testing whether the value of the `customer` attribute of an `Employee` object is `null` or a reference to a `Customer` object. Using this test, we can form the set of available employees and apply the built-in operation `size` to it.

The first postcondition of the operation contract asserts that after completion of the operation all customers have in fact been assigned to available employees. In this postcondition, first the set of employees that were available in the pre-state but are no longer available in the post-state is defined. Then we use the `collect` comprehension of OCL to obtain the collection of customers that are assigned to this set of employees. This collection is a bag, since OCL semantics is based on the general case that several employees may be assigned to the same customer, although this is excluded by the multiplicities in the class diagram. We use the built-in operation `asSet` to convert the bag to a set, so it can be compared to the set of new customers.

The second postcondition asserts that the assignment performed by the operation is stable. We quantify over all pairs e, c of available employees and new customers and consider the employee assigned to the customer ($c.employee$) as well as the customer assigned to the available employee ($e.customer$). This postcondition rules out that the pair $e-c$ is a better match than both $c-c.employee$ and $e-e.customer$. It does so by stating that the skills potentially provided by employee e to customer c are not a proper superset of the skills provided by $c.employee$ to customer c , or that employee e also prefers $e.customer$ if employee e lists customer c as preferred.

To complete the operation contract, we still need to specify which attribute values may be changed by the operation. We do this by adding a `modifies only` clause which states that the operation may only modify the attribute `customer` for the available employees and the attribute `employee` for the new customers. All other attribute values must be left unchanged by the operation. The operation contract in [Figure 8.2](#) does not contain an objective function, but our animation technique would also be able to take an objective function into account. `Modifies only` clauses and objective functions have not yet been incorporated into the OCL standard. See [Section 6.1](#) for details about these extensions to OCL.

We have now obtained an operation contract that precisely reflects the requirements. Note that the contract is underspecified, i.e., it does not prescribe a unique result, but allows the operation to perform any stable assignment. Moreover, the contract does not indicate how such an assignment can be found.

8.1.3 Animating the Operation Contract

The tool `OCLExec` we implemented our approach in generates Java method bodies. It inserts code that enforces the postconditions of the operation and all class

invariants. OCLexec serializes an intermediate representation of the operation contract to a file that the generated method body can access as a resource. This intermediate representation is based on the language of arithmetic formulas with bounded quantifiers introduced in [Chapter 5](#). The method body only reads the serialized file and calls a library routine responsible for animating the operation. The pre-state considered for animation is simply the pre-state of the method call. In principle, the results returned by this generated method body cannot be distinguished from results returned from a manually implemented method body that conforms to the operation contract. Note that inserting code in method bodies should not interfere with other code that may have been generated for the model. Thus, the developer can use her favorite tool for the overall code generation and then use our tool only for selected method bodies.

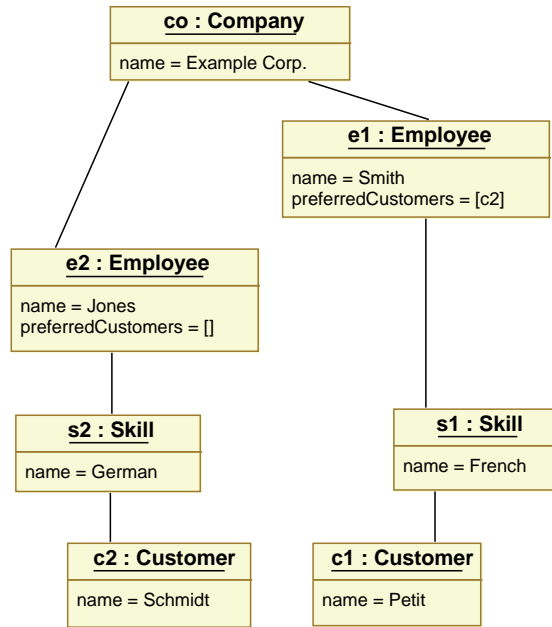
[Figure 8.3\(a\)](#) depicts a very simple system state in which the operation `assignNewCustomers` can be called. The company employs two staff members whose names are Smith and Jones. There are two skills: French and German language skills. Smith speaks French while Jones speaks German. There are two customers, called Petit and Schmidt, who ask for French and German language skills, respectively, from the employee that is assigned to them. Moreover, employee Smith prefers to work for customer Schmidt. [Figure 8.3\(b\)](#) shows a possible outcome of calling the generated method for the two customers in this system state. Employee Smith is assigned to customer Schmidt and employee Jones is assigned to customer Petit. Unfortunately, neither customers' request for language skills is met. However, the assignment is stable, since employee Smith is now assigned to his preferred customer Schmidt and therefore not interested in changing the assignment.

Depending on the needs of the company, this result of the operation call may not be sufficient. It may well be that the customers' demands for skills are deemed more important than the preferences of the employees. If this is the case, animation would have revealed an important flaw of the specification. Note that this kind of unforeseen behavior cannot be discovered if the constraints are only tested on system states that the specifier has designed to be correct or incorrect.

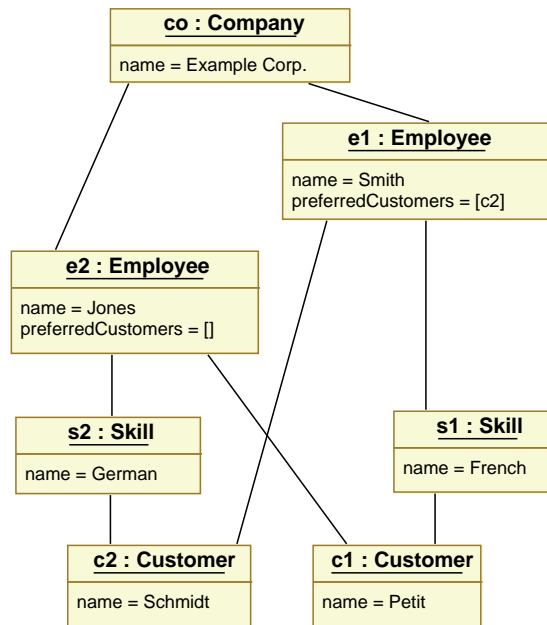
If the operation is not performance-critical and sufficiently efficient code can be generated for it, animation may allow to skip or postpone its implementation. Such an opportunity saves implementation effort and helps avoid coding errors. Moreover, a larger part of the development can be carried out on a higher and platform independent level of abstraction. In this sense, animation of operation contracts can be regarded as a contribution to Model-Driven Development.

8.2 Execution of Animation

We implemented an animation technique that is based on a translation of the operation contract to an arithmetic formula with bounded quantifiers as described



(a) State before animation



(b) State after animation

Figure 8.3: Effect of animating a call to the operation `assignNewCustomers` on a system state