Aspect-Oriented Monitoring of UML and OCL Constraints

Mark Richters
EADS SPACE Transportation
D-28199 Bremen, Germany
mark.richters@space.eads.net

Martin Gogolla
University of Bremen, FB 3
Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
http://www.db.informatik.uni-bremen.de
gogolla@informatik.uni-bremen.de

ABSTRACT

We present an approach utilizing aspect-oriented programming (AOP) techniques for mapping between different abstraction levels of software. The goal is to facilitate validation and testing of a software implementation against constraints specified on an associated UML model. We use AOP techniques for defining a monitor that observes the behavior of an implementation and maps it to model behavior. The model behavior is then validated against constraints with an existing tool. Constraint violations can thus be identified and traced to a specification of the model.

1. INTRODUCTION

Aspect-oriented techniques are popular today for addressing crosscutting concerns in software development. We describe an approach utilizing aspect-oriented techniques for mapping between different abstraction levels of software. The goal is to facilitate validation and testing of a software implementation against constraints specified on its associated UML model.

The Object Constraint Language OCL is used in UML to specify constraints such as invariants and pre- and postconditions [7]. OCL is a formal language for which various forms of tool support exist (see, for example, [6]). A tool developed by our group is called USE (UML-based Specification Environment) [3]. The USE tool allows validation of a formally defined subset of UML models and OCL constraints [5, 4].

The motivation for the work presented in this paper is that it should be possible to check constraints which were specified during modeling against an implementation of the model. A common approach to check constraints in an implementation is by mapping constraints to the target language [1]. This usually implies that a constraint checker has to be implemented in the target language and becomes part of the implementation.

The USE tool already has a sophisticated OCL interpreter and constraint checker that can be used to validate models. In order to also validate and test implementations we developed an approach for reusing this facility. The general idea is to map actions that manipulate implementation objects back to actions on the model level which manipulate objects specified with UML. These actions can then be validated with the USE tool. If, for example, a Java implementation violates an invariant that has been specified in the UML model this would be detected and flagged as a constraint failure.

The task of identifying actions where model and constraint related information is manipulated is a typical crosscutting concern. We use aspect-oriented programming (AOP) techniques for defining a monitor that observes the behavior of an implementation and maps it to model behavior. This monitor aspect bridges the gap between a model and its implementation by providing a mapping between the different abstraction levels.

There are several benefits to this approach. The aspect-oriented monitor directly attaches to a real application. It therefore allows validation of applications with real user data thus increasing the fidelity of the validation results. The development process improves because constraints not only add to the specification of a software. The same constraints can now also be checked without modification against an implementation. Consistency problems are avoided because there is only one place – the model – where constraints are specified. The monitor aspect only depends on the structure of a model. It needs no change if only constraints are changed. This allows for fast test-and-respecify cycles.

The aspect-oriented approach further has the advantage that an existing implementation does not have to be modified for adding the monitoring capability. Provided that it fulfills some general requirements allowing a backward mapping to the model it just has to be recompiled with an AOP capable compiler and the monitor aspect definition added.

The paper is structured as follows. In Section 2 we give a general outline of our approach. Section 3 describes how aspect-oriented techniques are used to add monitoring capabilities to existing Java applications. A small but complete example is given in Section 4. Some requirements on the applicability of our approach are described in Section 5. We close with a summary and draw some conclusions for future

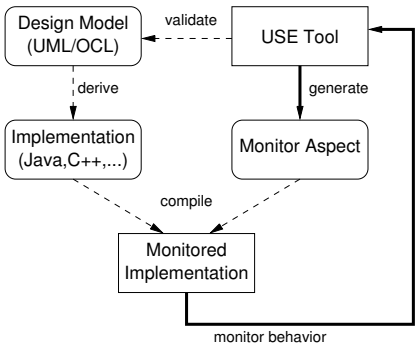

Figure 1: Overview of the process for monitoring implementation behavior

work.

2. MONITORING BEHAVIOR

The basic approach of monitoring is as follows. We monitor the behavior of an implementation, map the behavior back to state transformations on the design model, and finally validate constraints on the design model with the USE tool.

Figure 1 gives an overall picture of the process. The starting point is a UML design model with constraints and a program that implements this model in a language like Java or C++. The USE tool provides support for validating the UML model [5]. The extended USE version described in this paper additionally allows to generate a monitor aspect that is derived from the model specification. This aspect is compiled together with the implementation in order to produce a monitored implementation. When this implementation is run the monitor automatically generates data about the application's behavior. This data is then used as input for validation.

The monitoring process allows two different modes for validation. *Synchronized validation* takes place while the application executes. This allows to observe immediately the effects of application behavior. For example, an object diagram in the USE tool can be synchronized with the current state of the application. If objects in the application change their state, this change will be visible in the diagram. Also, constraint violations are signaled as soon as they are detected. As the application is running in parallel, there should be enough context available for analyzing possible reasons that may have caused the constraint to fail. The drawback of synchronized validation is that the monitoring process implies a certain degree of communication overhead. This may lead to different results with timing sensitive applications.

The second mode of validation is *offline validation* and happens after the application has completed. The application is monitored while it is running but the data is just recorded for later use. The recorded data contains all the information that is necessary to replay the behavior in the validation tool.

3. MONITORING WITH ASPECTS

Action	Pointcut
object creation	before constructor
object destruction	after destructor (finalizer in Java)
attribute modification	on field assignment
association link insertion	on collection add
association link removal	on collection remove

Table 1: Actions and associated pointcuts

Constraint	Pointcut
invariant	before/after each public method or constructor invocation
precondition	before operation call
postcondition	after operation call

Table 2: Constraints and associated pointcuts

The monitoring code that is responsible for observing the behavior of an implementation is added to the implementation using aspect-oriented programming (AOP) techniques. The USE tool can automatically generate aspect definitions for the AspectJ compiler [2]. These aspect definitions are derived from the UML model of an application and identify places where model related behavior is located. Ideally, an existing software only needs to be recompiled with the AspectJ compiler to insert the monitoring code at the appropriate places.

3.1 Aspect Definition

The purpose of the monitoring aspect is twofold. First, it specifies pointcuts where state changes in a program happen. State changes of interest are mapped to corresponding USE actions. Second, the monitoring aspect identifies places where constraints have to be checked.

State changes are defined by the actions listed in Table 1. For each action the associated generic pointcut is given. For example, pointcuts are defined for all constructors of monitored classes. An advice instructs AspectJ to insert code before constructor executions that records the creation of a new object. Likewise, attribute modifications can be observed by monitoring field assignments. Actions related to association manipulations are identified with certain operations on selected collections.

OCL constraints that are monitored include class invariants, and pre- and postconditions on operations. The places where these constraints are checked are listed in Table 2. A class invariant specifies a contract on the public interface of a class. It has to be checked before and after any invocation of a public method or constructor. Note that private methods are considered part of the implementation of a class and are therefore allowed to temporarily break an invariant. After all, private methods can only be called from public methods of the same class and finally end with a return from a public method. At that point invariants have to hold again. Pre- and postconditions are simply mapped to entry and exit points of methods.

3.2 Aspect Example

In this section, we give an impression of how a monitor aspect looks like. A monitor aspect is defined in a single file following AspectJ syntax. Here, we only look at the part that is responsible for monitoring the creation of new objects.

The aspect generation is driven by a template. For example, the following template provides a generic pointcut definition for all monitored classes. A string of the form @...@ is a placeholder that will be replaced with a concrete term depending on the UML model to be validated.

```
/**
 * Monitored classes.
 */
pointcut pcClass() :
@POINTCUT_CLASS@
  ;
```

Assuming that our model contains classes Company and Person the concrete pointcut designator will look as follows. See [2] for a description of the meaning of terms used within the pointcut declaration. This pointcut selects the classes Company and Person occurring in any Java package and rejects any classes nested within Company or Person.

```
/**
 * Monitored classes.
 */
pointcut pcClass() :
// Start of model-dependent information generated by USE
     (within(*..Company) && ! within(*..Company.*))
  || (within(*..Person) && ! within(*..Person.*))
// End of model-dependent information generated by USE
  ;
```

The following pointcut identifies places in the previously selected classes where object creation is occurring.

```
/**
 * The constructors of monitored classes.
 */
pointcut pcConstructor(Object _this) :
    this(_this) && pcClass() && execution(new(..));
```

These pointcuts are now used to define advices for monitoring. The before advice generates an action in USE syntax to be performed by the USE tool. The tool will create a new object of the given class with a unique identifier. After the constructor has executed, a constraint check will be performed.

```
/**
 * Monitor object creation.
 */
before (Object _this) : pcConstructor(_this) {
    logLocation(thisJoinPoint,
                "before constructor execution");
    // assign object a unique ID for identification
    // in USE
    ...
    // strip off package name
    ...
    // generate USE action
    log("!create " + getUSEObjectName(_this)
        + " : " + classname);
}

after(Object t) : pcConstructor(t) {
    logLocation(thisJoinPoint,
                "after constructor execution");
    // trigger a constraint check for invariants
    log("check");
}
```

Another AOP feature used in the monitor is responsible for mapping between object identities in the implementation and the model. The USE object model assigns each new object a unique identifier. A Java object has an identity but no public identifier. The solution is to use *introduction* which is an AOP concept allowing to modify a program's static structure. We introduce a field carrying the USE object identifier into each monitored class and let the class implement an interface providing access to the identifier.

4. EXAMPLE

In this section, we demonstrate our approach by monitoring a small Java application. The UML class diagram for the example is shown in Figure 2. There are classes Company and Person which are related to each other by a WorksFor association. Both classes have attributes and operations. The model is enhanced with OCL constraints that are not shown in the diagram but which will be discussed shortly.

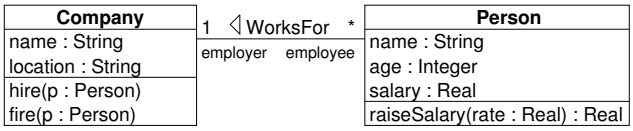

Figure 2: UML class diagram for example

The input for the USE tool is a model specification in textual form. The following is the complete USE specification of our example. It defines the classes and associations shown in Figure 2, and defines additional OCL constraints.

```
model Example

-- classes

class Company
attributes
  name : String
  location : String
operations
  hire(p : Person)
  fire(p : Person)
end

class Person
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end
```

```
-- associations

association WorksFor between
  Company[0..1] role employer
  Person[*] role employee
end

-- constraints

constraints

context Person inv:
  age >= 18

context Person::raiseSalary(rate : Real) : Real
  post raiseSalaryPost:
    salary = salary@pre * (1.0 + rate)
  post resultPost:
    result = salary

context Company::hire(p : Person)
  pre  hirePre1: p.isDefined()
  pre  hirePre2: employee->excludes(p)
  post hirePost: employee->includes(p)

context Company::fire(p : Person)
  pre  firePre:  employee->includes(p)
  post firePost: employee->excludes(p)
```

The constraints include a class invariant making sure that all persons in our example are at least 18 years old. Furthermore, we define several pre- and postconditions on the operations. The postcondition on raiseSalary asserts that an employee's salary is indeed raised after the operation and that the new value is returned as a result of the operation call. Later in the example, we will show how an incorrect Java implementation of this constraint will be detected with our monitoring approach. The precondition specified on the operation hire makes explicit our assumption that a person is not already working for a company when she is hired. Likewise, the postcondition states that the person is included in the set of employees after hiring. Constraints on the operation fire work analogously.

Next, we show a straightforward translation of this model into a Java implementation. Note that the code does not contain an implementation of the constraints. Our goal is to validate applications (fulfilling certain requirements, see Sect. 5) that do not spend extra effort on constraint checking. A great benefit of this approach is that the constraints can easily be modified in the model specification. In this case the application can be revalidated without having to change the implementation.

```
import java.util.*;

public class Company {
    private String name;
    private String location;
    private Set employee;

    public Company(String name, String location) {
        this.name = name;
        this.location = location;
        this.employee = new HashSet();
    }

    public void hire(Person p) {
        employee.add(p);
    }

    public void fire(Person p) {
        employee.remove(p);
    }
}
```

Next comes the implementation for class Person. The body of the method raiseSalary intentionally does not match the postcondition specified on the model.

```
public class Person {
    private String name;
    private int age;
    private double salary;

    public Person(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public double raiseSalary(double rate) {
        // this is wrong w.r.t. to the specification!
        return salary += 10;
    }
}
```

Finally, a short main program is added. It creates two objects and calls each of the methods.

```
public class Main {
    public static void main(String[] args) {
        Company company =
            new Company("Foo, Inc.", "Bremen");
        Person bob = new Person("Bob", 35, 2000);
        company.hire(bob);
        bob.raiseSalary(0.10);
        company.fire(bob);
    }
}
```

4.1 Validation

We now have a UML model and a Java implementation. The validation process for the given example is as follows. First, we load the specification into the USE tool. The tool uses the information contained in the model to generate an aspect definition for the monitoring process as has been outlined in Section 3. Next, the Java application together with the monitoring aspect is compiled with the AspectJ compiler. As explained earlier, we can now choose to run the application in parallel to the USE tool or to record actions into a file for later use. Parallel validation runs over a network connection. The USE tool opens a network socket allowing the monitored application to remotely control the validation process. While the application is sending commands to USE the state of the running system is updated in the USE user interface.

Figure 3 shows the user interface at the end of the main program. The left part of the window shows static information from the model specification including classes, associations,

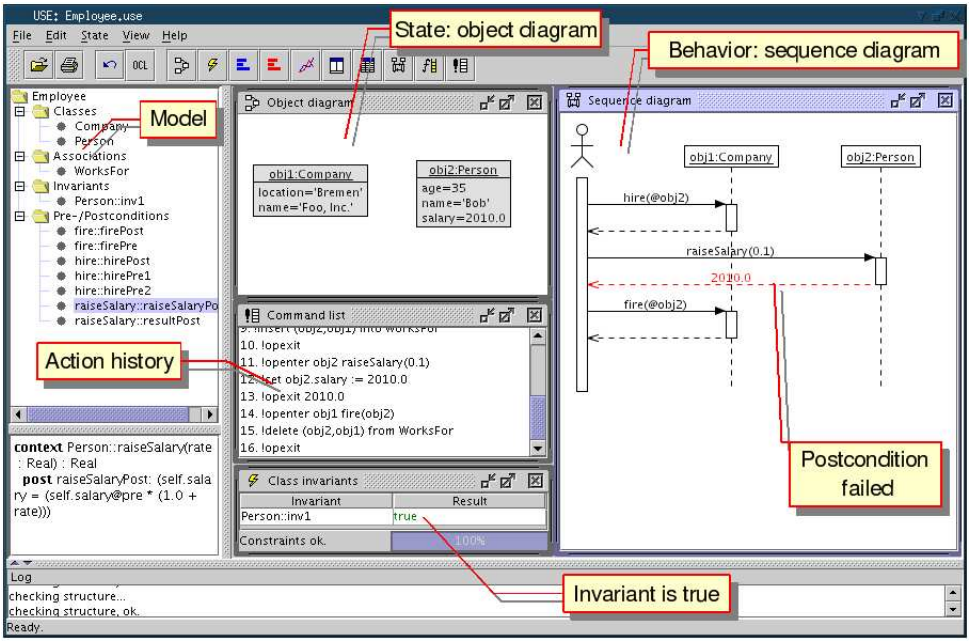

Figure 3: Screenshot of the USE tool validating the example model

and constraints. The lower left panel shows details of a selected constraint. The object diagram in the center of the window is synchronized with the current application state. There is a company object and a person object. After the hire operation has been executed there also was a link between both objects. However, as expected, the fire operation has removed the link again. The panel below the object diagram shows the history of actions. It can be seen that the fire operation has just exited. The bottom panel shows the result of all invariants in the model. In our example, there is only one invariant and it is satisfied in this system state. The sequence diagram on the right is automatically generated and shows a trace of operation calls in the system. It expands to the bottom as methods are called in the Java application.

If a pre- or postcondition on a method call fails the corresponding transition in the sequence diagram is marked with red color. Related to our example, we can observe that the postcondition on the raiseSalary operation has been violated. According to our specification the new salary value should be 2100 but the application delivered the valued 2010. By monitoring the application, we were able to identify a deviation of the implementation from its specification. With this information the Java code can easily be corrected.

4.2 Monitor Trace

We have seen how the behavior of the example program can be monitored and visualized with the validation tool. In the following, we will take a closer look at the steps performed during monitoring. Let us first see what happens when the program executes the first statement in the main method:

```
Company company = new Company("Foo, Inc.", "Bremen");
```

This is a constructor call creating a new object of type Company. The task of the monitoring aspect is to (1) generate an action for object creation (according to Table 1), and (2) to trigger an invariant check (according to Table 2). For this example, the monitoring aspect generates the following verbatim output. This output complies with the command script syntax of USE. It can therefore directly be used to control the simulated system in USE.

```
-- Company.java:8: before constructor execution
!create obj1 : Company
-- Company.java:9: setting field
!set obj1.name = 'Foo, Inc.'
-- Company.java:10: setting field
!set obj1.location = 'Bremen'
-- Company.java:8: after constructor execution
check
```

Lines starting with two dashes are comments. Here they are used to indicate the location in the Java source code where a pointcut advice has been applied. An exclamation mark starts a USE action to be performed on the model instance. The first action creates a new company object. Note that the mapping performed between Java objects and UML objects also introduces unique object identifiers such as `obj1`. A set action assigns an object attribute a value. Finally, an invariant check is triggered after the constructor execution has finished.

The next statement creates a new person object.

```
Person bob = new Person("Bob", 35, 2000);
```

The generated monitor output looks very similar to the output shown above and is therefore omitted here. More interesting is the following statement which is a method call.

```
company.hire(bob);
```

If we look at the method's implementation, we see that it adds the person bob to the set of company employees. On the UML model level this corresponds to establishing a link of the WorksFor association between classes Company and Person. This is exactly what the following sequence of USE actions generated by the monitor does.

```
-- Company.java:14: before method execution
!openter obj1 hire(obj2)
-- Company.java:15: adding link
!insert (obj2, obj1) into WorksFor
-- Company.java:14: after method execution
!opexit
```

The method call as a whole is wrapped into enter and exit actions. These are the places where pre- and postconditions are checked, respectively. The modification of the employee set in the body of the hire method is recognized by the monitor aspect as an action that modifies the link set of the WorksFor association. The corresponding USE action therefore inserts a link specified by the pair of related objects.

The next statement is again a method call.

```
bob.raiseSalary(0.10);
```

The method call results in the following actions.

```
-- Person.java:12: before method execution
!openter obj2 raiseSalary(0.1)
-- Person.java:13: setting field
!set obj2.salary = 2010.0
-- Person.java:12: after method execution
!opexit 2010.0
```

In the UML model, the corresponding operation is defined to return a result value. The monitor passes this value to the opexit action so that it can be used for evaluation of the postcondition. In this example, we have supplied a wrong implementation of the raiseSalary operation that does not conform to the specified constraint. The expected value is 2100 but the monitor observed that the implementation computed 2010. As explained above, this is highlighted in the USE sequence diagram as an error return from the operation.

The last statement in the program is

```
company.fire(bob);
```

This method removes the previously established link from the WorksFor association. The corresponding USE action is a delete command wrapped between enter and exit actions.

```
-- Company.java:18: before method execution
!openter obj1 fire(obj2)
-- Company.java:19: removing link
!delete (obj2, obj1) from WorksFor
-- Company.java:18: after method execution
!opexit
```

Again, the final state at program termination can be seen in Figure 3.

5. APPLICABILITY

We make some assumptions about a model and its implementation in order to be applicable to our approach. For example, a simple requirement is that names assigned in the UML model can uniquely be mapped to equivalent names used within the implementation's source code. However, it is not required – and actually almost never the case – that *all* implementation classes are specified in the model. In general, only the important classes representing the business or application logic are modeled. Implementation specific classes are added to the program but usually these are better tested by other means, for example, with unit tests.

A more restraining requirement concerns the mapping of associations. This is a construct that has no direct counterpart in programming languages like Java. There are numerous ways an association can be implemented. Finding an aspect definition for association handling code is similar to the problem of reverse engineering UML associations from software. We have chosen a straightforward solution based on a simple pattern that can be found in many Java implementations. We assume that associations are implemented either with simple references for multiplicities of no more than one, or with Java Collection classes like sets for multiplicities greater than one. These references or sets are placed in the classes participating in the association.

These assumptions were made to provide a simple and direct mapping for which a monitor can automatically be instantiated. Of course, more complex mappings could be defined if necessary or provided manually.

6. CONCLUSION

We presented an aspect-oriented approach that facilitates validation and testing of a software implementation against constraints specified on an associated UML design model. The key component of this approach is a monitor that is added to an implementation without requiring changes of an existing application. The monitor provides a mapping between program behavior and model behavior and enforces a clear separation of abstraction levels: the model does not use Java constructs, and the implementation does not need to know about the UML model. All steps in this process can be automated so that a framework for automated tests seems possible. We are currently investigating the scalability of our approach with a larger case study. The application is a distributed multimedia conferencing tool with audio/video, chat and shared white-board components. Results so far show that not only the aspect of constraint checking aspect is useful. Also the fact that the application behavior can be visualized in terms of UML object and sequence diagrams helps to understand the way of how an application works.

7. REFERENCES

[1] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[3] Mark Richters. The USE tool: A UML-based specification environment, 2001. Internet: `http://www.db.informatik.uni-bremen.de/projects/USE/`.

[4] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[5] Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, pages 265–277. Springer, Berlin, LNCS 1939, 2000.

[6] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[7] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.