Using Object Teams for State-Based Class Testing

Bericht-Nr. 2004/10

Dehla Sokenou, Stephan Herrmann

ISSN 1436-9915

Using Object Teams for State-Based Class Testing

Dehla Sokenou, Stephan Herrmann

Abstract

Statechart models are often used to specify the behaviour of objects. This paper presents a technique for testing object-oriented classes using their statechart specification.

Aspect-oriented programming integrates additional code into existing classes. Program instrumentation for testing and monitoring systems is one application domain of aspect-oriented programming.

The presented approach is based on the Object Teams [17] programming model which combines aspect-oriented and role-based programming.

Object Teams are used to implement statechart specifications, thus making them executable. The executable statechart specification is bound in an aspect-oriented manner to the implementation under test and is used to dynamically validate the implementation of a class.

The paper demonstrates the advantages offered by Object Teams' characteristics for object-oriented class testing.

Keywords:

model-based testing, test oracles, UML statecharts, aspect-oriented programming

1 Introduction

The Unified Modeling Language [19] defines two different kinds of statecharts, behavioural state machines and protocol state machines. Behavioural state machines are typically used to specify system behaviour, while protocol state machines can be used to specify the behaviour of objects.

Our approach is to use protocol state machines as test oracles to validate the implementation of a class. We use the Object Teams [17] programming model to implement the statechart specifications, thus making them executable.

The Object Teams programming model combines several programming techniques, primarily aspect-oriented [11] and role-based programming [12]. [8] gives further details.

Aspect-oriented programming integrates additional code into existing classes. Program instrumentation for testing and monitoring systems is one application domain of aspect-oriented programming. In our approach, aspect-oriented features of the Object Teams programming model are used to integrate test code into classes under test in a non-invasive manner.

We implement statecharts as roles of objects under test. Roles remain independent objects with their own states even at runtime. This technique has several advantages, which are presented in this paper.

The paper is organized as follows. In Section 2, we introduce the basic formalism and give a brief overview of state-based test oracles for object-oriented programs. Section 3 gives a brief introduction to the Object Teams programming model. In Section 4, we explain our approach to implementing statechart models in ObjectTeams/Java, an implementation of the Object Teams programming model for the Java language. Some requirements for the applicability of our approach are described in Section 5, followed by an overview of related work in section 6. Section 7 contains a summary and outlook.

2 Statecharts as Test Oracles

A state-based testing strategy can detect many different kinds of faults resulting from an incorrect implementation of the specified model. Typical faults are incorrect resultant states or incorrectly accepted or refused messages. A good overview of faults detectable by test oracles based on statechart specifications can be found in [16] and [1].

A statechart model can describe the behaviour of objects in a similar way to, e.g., the life cycle model in Fusion [3]. The UML [19] defines protocol state machines for describing object behaviour. In our approach, we concentrate on class testing. A statechart specification in this context means the specification of object behaviour. Thus we focus our attention on protocol state machines.

The semantics of protocol state machines differs from the semantics of behavioural state machines in some points. Transitions in protocol state machines are triggered by call events. A call event refers to a method of the attached class. States are identified by the attribute values of the corresponding object. Protocol state machines refer to all methods that generate state changes. These methods are called update methods. Methods that generate no state change, so-called query methods, are not presented in the state machine. Transitions in protocol state machines have no associated explicit actions.

For class testing, a statechart implementation (see Section 3) is generated from the protocol state machine specification. Statechart objects are bound to objects under test. Every method call to an object under test is forwarded to the corresponding statechart objec, which calculates the expected resultant state on the basis of the actual state and the given call event. The actual state

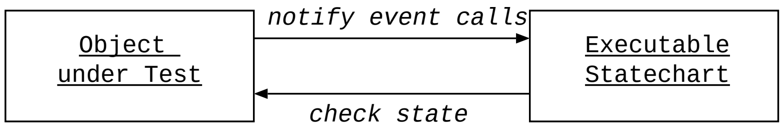

Figure 1: Statechart object and object under test.

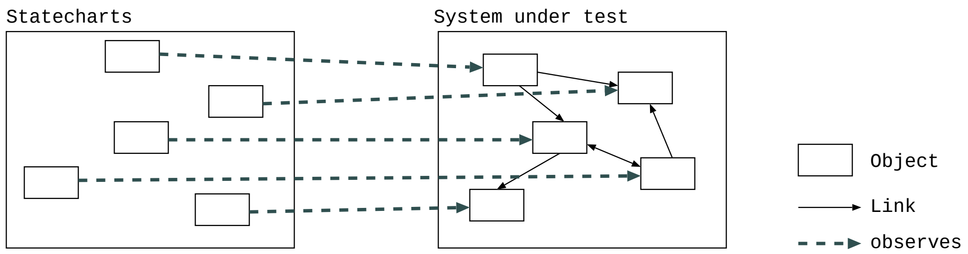

Figure 2: Statecharts system and system under test.

of the object under test is then compared with the expected state (see Figure 1).

Figure 2 shows the system under test and the statechart objects. Every object in the system is observed by a corresponding statechart object.

Our approach can be used to support system testing with certain restrictions. One of these restrictions is that no relationship between objects is considered. Each statechart object only has access to its own object under test and no connection to other statechart objects exists. Only the set of individual object behaviours in a system is tested (see Figure 2).

3 A Brief Introduction to Object Teams

The Object Teams programming model combines several approaches of other programming paradigms and techniques. It introduces a new module concept, the team. Teams are packages that group classes for structuring systems (static view) and also composite objects that are used as containers for objects (dynamic view).

Teams can be handled like normal classes, including instantiation and inheritance. Instances of a team are containers for objects defined within the team.

Elements of a team define the roles of objects, implemented as inner classes. Roles are partial views of objects in different contexts. Roles are bound to the objects of a base system. Objects of the base system are called base objects. We

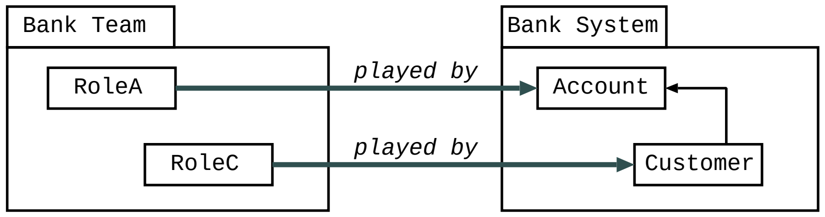

Figure 3: Team and base system

use ObjectTeams/Java, which realises the Object Teams programming model for the Java language. A base system can be any application written in Java, e.g. packed in a jar archive.

For example, we have an existing bank system[1]. We wish to adapt the bank system with a team to extend the given implementation (e.g. we may want to introduce a bonus for special transactions). In Figure 3, the adapted base system is shown on the right side, the team on the left side. The role classes in the team are bound to the bank classes, `RoleA` being played by objects of type `Account` and `RoleC` being played by objects of type `Customer`[2].

One of the basic concepts in Object Teams is the aspect-oriented [11] approach, which is used to weave code[3] into base classes. In the Object Teams world, call-in binding is the technique that realises aspects. Unlike other aspect-oriented programming languages, the Object Teams programming model does not yet have any sophisticated join point languages. In particular, there is a lack of quantification. In languages like AspectJ [10], quantification is offered by the wild card notation and a set of point-cut designators. The additional code can be executed before or after a method of the corresponding base object is executed or can either replace the method of the base object.

The opposite of call-in binding is call-out binding. Methods of the role are forwarded to methods of the base object. The role object has privileged access to the corresponding base object, and call-out methods can access all methods of the base objects including private methods. All methods of the base object called from the role object must be explicitly referenced by call-out binding. Call-out bindings offer a limited view on a base object.

Call-in and call-out methods can have mismatching signatures to the bound methods in the base system. Defining parameter mapping is optional.

In Object Teams, code weaving is done at load time. Only the teams must be compiled with the Object Teams compiler, not the base system. The base system can be compiled using a standard Java compiler, which can even be

[1]Bank account is used as a running example throughout the paper.

[2]The term *played by* for the relation between role and base object originates from role-based programming and is found as a keyword in Object Teams to bind a role to a base.

[3]*Code weaving* is the technique used to adapt an existing implementation by means of new code.

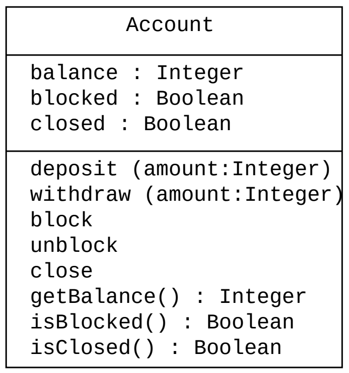

Figure 4: Static view of account.

provided by a third party. No source code is needed for adaption.

The Object Teams programming model offers lifting of base objects to their roles. Lifting means retrieving a role for a given base object. The opposite, retrieving a base object from a role object, is called lowering. Lifting and lowering are never explicitly invoked by client code. These translations are performed automatically by the Object Teams runtime system.

Teams can be activated or deactivated at runtime. If a team is active, all call-in bindings are enabled; if it is deactivated, they have no effect on the base system. An inactive team holds its state and also the state of its contained roles until it is reactivated. The Object Teams runtime system guarantees that a base object is always lifted to the same role in a team. After team activation, all base objects are lifted to the same roles they had before deactivation. The states of these roles are preserved. The activation order effects the execution order of call-in methods.

A more detailed introduction to Object Teams is given in [8].

In the next section, we explain how we use the Object Teams' features for state-based class testing and point out the specific benefits of our approach.

4 Team-Based Class Testing

To illustrate our approach, we introduce a short example, a bank account. An account can be open, blocked or closed, and money can be deposited or withdrawn. Figure 4 shows instance variables and methods of class `Account`.

The basic idea is to implement the statecharts as roles in Object Teams. The statechart teams are automatically generated. Below, we give an overview of the generated implementation and show which statechart features are supported.

The executable statechart is defined as a role played by the object under test, here the account object (see Figure 5).

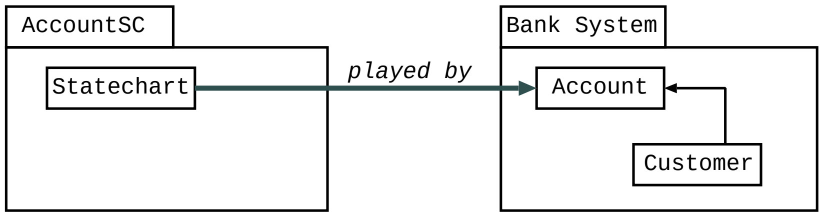

Figure 5: Statechart team and bank base system

One of the advantages of implementing the statecharts as teams is the integration of test code into the classes under test in a non-invasive manner[4]. The code of the classes under test does not require any change or recompilation.

Figure 6 on page 7 shows a code fragment of the implemented statechart for the bank account in the team class `AccountSC`. The role `Statechart` is played by objects of type `Account`.

The instance variable `state` in the statechart role is used to store the expected state of the object under test. In the example, the initial state of the bank account is open. The other instance variable, `openState`, is introduced in Section 4.1.

Every method call to an object under test is intercepted and transmitted to the statechart by the call-in mechanism. We use this to log every method call and check the state of the object under test. The state check compares the expected state with the actual state of the object under test. For update methods, there are corresponding methods implemented in the statechart that are called as a result of call-in bindings after the update methods of the account object have been executed. The corresponding methods implement the state change in the statechart object and call the method `checkState` afterwards.

In the code fragment shown in Figure 6, the call-in bindings can be found in the *`Call-in bindings`* part of the implementation[5]. The method `logMethodCall` is executed before an update method is executed, the method `checkState` afterwards. The method `checkState` is responsible for comparing the expected with the actual state of the object under test. The binding of the method `logMethodCall` is an example of parameter mapping in Object Teams. In our case, it has to distinguish different methods that are bound to `logMethodCall`.

The method `checkState` is not directly called by the call-in binding mechanism if the update method `close` is called. The method `close` has a corresponding method with the same name and signature, which is called after the method `close` of the object under test has been executed. The implementation of the corresponding method `close` can be found in the *`Trigger events`* part. The corresponding method `close` compares the expected with the actual state

[4]This is also called non-invasive instrumentation.
[5]Call-in bindings are denoted in Object Teams by the symbol `<-`.

```
public team class AccountSC {

   private Statechart myAccount;

   public AccountSC() {
        this.activate();
   }

   public void setRole(Account as Statechart asc) {
        myAccount = asc;
        myAccount.init();
   }

/* Role class */
   class Statechart playedBy Account {

       private int state = OPEN;
       private OpenSC openState;

       /* Initialize Role */
       private void init() {
           openState = new OpenSC();
           openState.setRole(this);
       }

       /* Check state by comparing expected and current state */
       void checkState() {
           if (this == myAccount) {
               if (state == CLOSED) {
                   if (isClosed()) logState ();
                   else logStateError ();
       } } }

       /* Trigger events */
       public void close () {
           if (this == myAccount) {
               openState.deactivate();
               System.out.println("Team 2 deactivated: " + getHashCode());
               state = CLOSED;
               checkState();
       } }

       /* Call-in bindings */
       void logMethodCall(int id) <- before void deposit(int amount)  with {id <- 0};
       void logMethodCall(int id) <- before void withdraw(int amount) with {id <- 1};
       void logMethodCall(int id) <- before void block()              with {id <- 2};
       void logMethodCall(int id) <- before void unblock()            with {id <- 3};
       void logMethodCall(int id) <- before void close()              with {id <- 4};
       checkState <- after deposit, withdraw, block, unblock;
       close      <- after close;

       /* Call-out bindings */
       abstract int getBalance();    getBalance  -> getBalance;
       abstract boolean isClosed();  isClosed    -> isClosed;
       abstract boolean isBlocked(); isBlocked   -> isBlocked;
       abstract int getHashCode();   getHashCode -> hashCode;

       /* Log Methods */
       [...]
   } }
```

Figure 6: Account statechart implementation

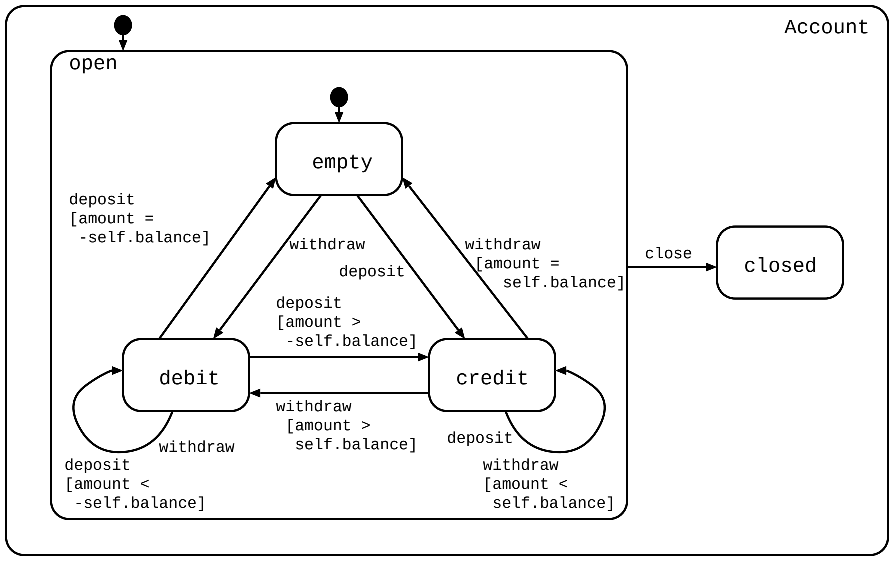
Account
open
empty
closed
debit
credit
close
deposit [amount = -self.balance]
withdraw
deposit
withdraw [amount = self.balance]
deposit [amount > -self.balance]
withdraw [amount > self.balance]
withdraw
deposit [amount < -self.balance]
deposit
withdraw [amount < self.balance]

Figure 7: Hierarchical statechart for account.

and then calls the method `checkState`. In ObjectTeams/Java, two or more call-in bindings are not permitted for the same method in the same way (before or after) to avoid nondeterministic situations at runtime. Thus the method `checkState` cannot be called directly by call-in binding in this case.

Call-out bindings are used to access query methods. They are needed to calculate the actual state of the object under test.

In the account example, call-out bindings[6] (in the ***`Call-out bindings`*** part) refer to the query methods `getBalance`, `isBlocked`, `isClosed` and `hashCode`. A mechanism not used in this example is the ability to access even private methods of the base object by means of the call-out mechanism.

After outlining the basic implementation idea, we present four different statecharts that specify the bank account in the following sections. Each statechart specification serves to illustrate how our approach supports specific statechart features. First, we present a simple statechart with hierarchy. The second statechart includes the history connector, and the third combines two parallel statecharts. Finally, we explain how we deal with nondeterminism.

4.1 Hierarchy

Figure 7 shows a hierarchical statechart specification of an account. First, we look at the hierarchy levels separately, as in Figure 8. Each level is implemented

[6]Call-out bindings are denoted in Object Teams by the symbol `->`.

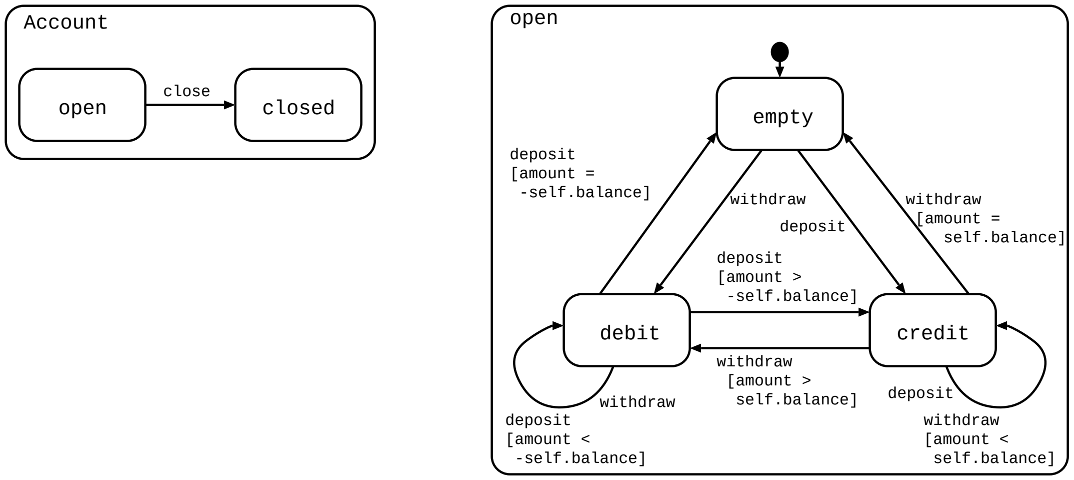

Figure 8: Hierarchical statechart for account.

by its own statechart team. In our example, we implement an account team (in team class `AccountSC`) for the top-level statechart and an open team (in team class `OpenSC`) for the sub-level statechart.

Each statechart is responsible for its own states and transitions. In our example, the account team is responsible for the transition triggered by the method `close`, and the open team for the transitions triggered by the methods `deposit` and `withdraw`.

The top-level statechart manages all sub-level statecharts. Sub-level statecharts are initialized, activated and deactivated by their enclosing statechart. If a statechart team is active, the corresponding object under test is in the state represented by the team. The set of activated teams for an object under test corresponds to the actual state configuration of this object.

In the example (see Figure 6), the instance variable `openState` holds the reference to the sub-statechart team, here the team that is responsible for the open state. A code fragment of team `OpenSC` can be found in Figure 9 on page 10. If the expected state is open, the `OpenSC` team is active, otherwise it is inactive.

Both roles, the `AccountSC` statechart object and the `OpenSC` statechart object, are played by the same account object. By calling the method `setRole`, the statechart object in `AccountSC` passes a reference of itself to the statechart object in `OpenSC`. The method `setRole` of team `OpenSC` expects an `Account` object, so the `AccountSC` statechart object is automatically lowered to its base `Account` object. The method `setRole` lifts the given `Account` object automatically to its additional role, here the `OpenSC` statechart object. This is expressed by the signature of `setRole` (`Account as Statechart asc`). Thus two roles

```
public team class OpenSC {

    private Statechart myAccount;

    public OpenSC () {...}

    public void setRole(Account as Statechart asc) {
        myAccount = asc;
    }

/* Role class */
    class Statechart playedBy Account {

        private int state = EMPTY;

        void checkState() {...}

        /* Trigger events */
        public void deposit (int amount) {...}
        public void withdraw (int amount) {...}

        /* Call-in bindings */
        void logMethodCall(int id) <- before void deposit(int amount)  with {id <- 0};
        void logMethodCall(int id) <- before void withdraw(int amount) with {id <- 1};
        void logMethodCall(int id) <- before void block()              with {id <- 2};
        void logMethodCall(int id) <- before void unblock()            with {id <- 3};
        void logMethodCall(int id) <- before void close()              with {id <- 4};
        checkState <- after block, unblock, close;
        deposit    <- after deposit;
        withdraw   <- after withdraw;

        /* Call-out bindings */
        abstract int getBalance();
        getBalance  -> getBalance;
        abstract boolean isClosed();
        isClosed    -> isClosed;
        abstract boolean isBlocked();
        isBlocked   -> isBlocked;
        abstract int getHashCode();
        getHashCode -> hashCode;

        /* Log Methods */
        [...]
    }
}
```

Figure 9: Open statechart implementation

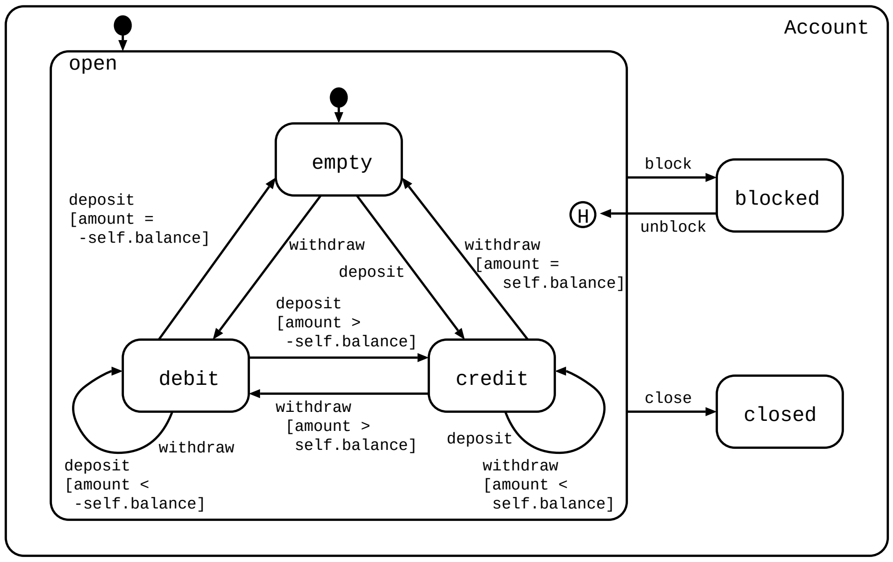

Figure 10: Statechart with history connector for account.

can easily share a common base where data flows involve lowering and lifting.

Because the enclosing statechart controls all its sub-statecharts, interlevel transitions are managed by the enclosing statechart in which the interlevel transition starts or ends. For incoming transitions, the enclosing statechart deactivates the sub-statechart that is the transition source. For outgoing transitions, the enclosing statechart activates the sub-statechart that is the transition target. This includes all in-between levels.

The activation order of the Object Teams runtime system influences the execution order of call-in bindings. The before methods of the last activated team are executed first, and the after methods of that team are executed last. This activation order matches the semantics of UML statecharts, where the innermost transition is preferred in the case of conflicting events. Other statecharts semantics like the Harel statecharts [7] are not supported in this simple way (see also Section 4.5).

4.2 History

Many testing approaches do not consider the history connector (e.g. [1], [2]). Our approach offers a simple way to implement the history connector. If a state with a history connector is entered, the sub-state that was active before leaving the state will also be entered. Since a team holds its state while deactivated, in particular the expected state of the object under test, the history connector is implicitly provided by the Object Teams runtime system. Only if we do not

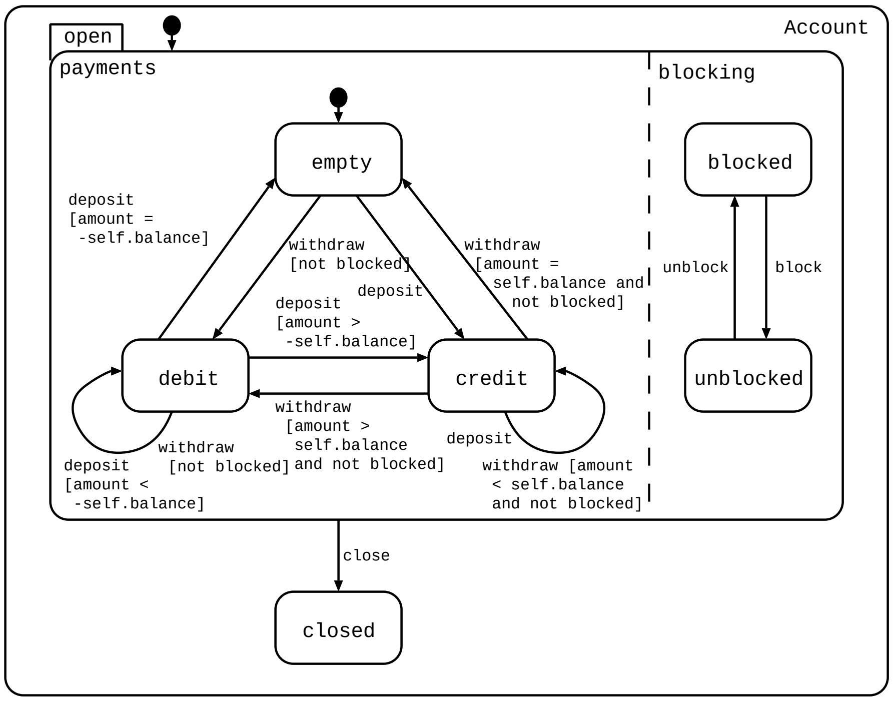
Account
open
payments
blocking
empty
debit
credit
blocked
unblocked
closed
deposit [amount = -self.balance]
withdraw [not blocked]
deposit
withdraw [amount = self.balance and not blocked]
deposit [amount > -self.balance]
withdraw [amount > self.balance and not blocked]
withdraw [not blocked]
deposit [amount < -self.balance]
deposit
withdraw [amount < self.balance and not blocked]
unblock
block
close

Figure 11: Statechart with parallelism for account.

use the history connector do we have to reset the team that implements the sub-statechart to its initial state when entering a sub-state.

Figure 10 shows a statechart with a history connector for the account. The statechart has three top-level states, `open`, `closed` and `blocked`. If the account is blocked and then unblocked, the account returns to the state that it was in before it was blocked (`empty`, `debit` or `credit`). The events triggered by the methods `block` and `unblock` activate and deactivate the team for the open state. No more has to be done in terms of the history connector.

4.3 Parallelism

To implement parallel statecharts, we take advantage of another Object Teams feature. It is possible to bind the same base object to more than one role object in the same team. The idea is to implement the parallel states (And-states) of the statechart in one team, with each statechart in its own role.

In our example, shown in Figure 11, the statechart has two parallel states, the `payments` state and the `blocking` state.

The code fragment of the implementation is given in Figure 12 on page 13. The roles `PaymentsStatechart` and `BlockedStatechart` are both played by

```
public team class OpenSC {

    private PaymentsStatechart myPaymentAccount;
    private BlockingStatechart myBlockingAccount;

    public OpenSC () {
        this.activate();
    }

    public void setRole(Account asc) {
        setPaymentsRole(asc);
        setBlockingRole(asc);
    }

    public void setPaymentsRole(Account as PaymentsStatechart asc) {
        myPaymentAccount = asc;
    }

    public void setBlockingRole(Account as BlockingStatechart asc) {
        myBlockingAccount = asc;

    }
/* Role class */
    class PaymentsStatechart playedBy Account {
        [...]
    }

/* Role class */
    class BlockingStatechart playedBy Account {
        [...]
    }
}
```

Figure 12: Parallel open statechart implementation

the same account object.

Role lookup[7] remains unambiguous because different roles are distinguished by their role class (see signature of `setPaymentsRole` and `setBlockingRole`).

4.4 Nondeterminism

Nondeterminism is another statechart feature supported by our approach. The statechart specification can be nondeterministic, but the corresponding implementation under test deterministic.

In situations where more than one transition is enabled to fire, we are faced with two problems. First, the statechart object does not know which transition was selected, so we have to store all possible resultant states. Second, the subsequent state check must consider all possible resultant states.

In Figure 13, in the state `debit`, for example, a call of method `deposit` can lead to the states `credit` or `empty` or can trigger a self-transition to the state

[7]The role lookup mechanism lifts a base object to a suitable role.

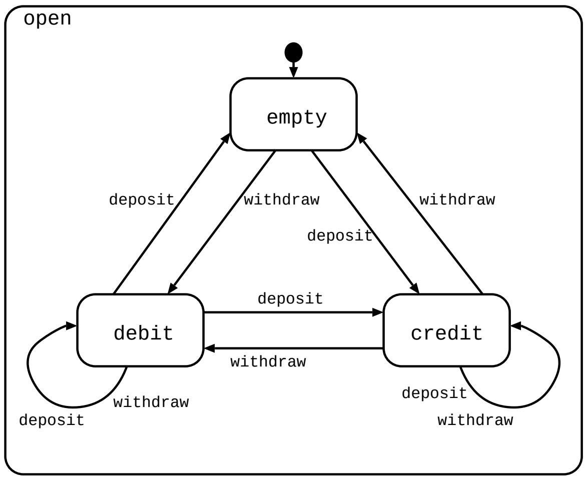

Figure 13: Nondeterministic statechart for account.

`debit`.

In our implementation, we have two extensions for such a situation. First, we set the instance variable `state` to an undefined state and store all possible resultant states in another instance variable, here the states `credit`, `empty` and `debit`. The method `checkState` and all update methods are extended, so they can consider all states in the set of possible states for their calculation. All states in the set of possible states are simple states, not composite states. We have to make this restriction to avoid instantiating a team for each possible composite state. The state is undefined until `checkState` can reconstruct the actual state of the object under test. Reconstruction is possible if the set of possible states filtered by matching the actual state of the object under test to all possible states is reduced to one state.

This kind of implementation checks that one of the possible paths through the statechart is correct, but does not know which of the paths the object under test has taken. Nondeterminism impairs the testability of the implementation under test. Nondeterminism should be avoided if possible, but it can be handled by our approach.

4.5 Conclusion

In the previous sections, we have presented our implementation technique for statecharts. We now discuss the advantages and drawbacks of our approach.

We have implemented our technique in ObjectTeams/Java, a Java extension to support Object Teams. So far, we have automatically generated teams only for Java systems, but our approach can be applied to all supported languages, like Ruby Object Teams and Object Teams for C++.

The basis of the code generation is a non-custom format for statecharts. Our next step is to transform XMI [21] models to our format. This will allow us to define statechart models with every modeling tool that supports XMI.

The Object Teams programming model offers a simple way to implement statechart specifications, in particular, history is supported by the runtime activation technique, and parallelism by the option of defining two or more roles in one team for the same object.

Unfortunately, the execution order of activated teams is predefined. Allowing it to be choosen by the programmer could be helpful. In our approach, the execution order matches the UML statechart semantics. A simple switch to other statechart semantics is not yet possible.

The base object can be accessed even by its private methods. So far, however, base attributes cannot be accessed using call-out bindings. For testing purposes, we have to enforce an implementation of query methods for all needed attributes. Later versions of the Object Teams compiler will support call-out bindings to attributes, disregarding restricted visibility.

We had few problems with the lack of a sophisticated join point language. For our approach, quantification as in AspectJ [10] is not needed. The approach would benefit from a join point language that distinguishes between update and query methods. Such a join point language is offered by the aspect-oriented language Java Aspect Components (JAC) [9] and planned in later versions of the Object Teams compiler.

Another advantage is the technique of load-time weaving. The implementation under test does not have to be recompiled, only the additional code must be compiled with the Object Teams compiler. Configuration management and deployment are also facilitated, because the instrumented application does not need to be stored on disk as a variant of the original system.

The overhead incurred by using teams instead of normal role objects (in our approach a team instance normally contains only one role instance) may lead to lower performance of the system under test. Weather aspect-oriented or conventional, each instrumentation necessarily impacts the performance of the considered system because of the additional code. Our approach is only suitable for conformance testing. Performance testing must be done by other tools.

Despite some drawbacks, the advantages of our approach are still significant, especially the support of Object Teams for statechart features like history. Some of the drawbacks will be fixed in later versions of the Object Teams compiler.

5 Applicability

This section lists the requirements for the statechart models and the implementation under test for use with our approach.

For instance, a simple requirement is the matching of names between the model and the implementation. Names assigned in the statechart model have to be mapped to equivalent names in the implementation under test. Other approaches like [18] make similar requirements.

We expect that all states in the statechart models are defined based on query methods of the object under test. The definition does not have to be deterministic, but a deterministic definition would improve testability and therefore the effectiveness of the test (see also Section 4.4).

We assume that query methods have no side-effects because they are used to report the actual state of the object under test. Our approach excludes query methods from the test, so so we cannot be sure whether they really do not change the state of the object under test. We are therefore developing additional test techniques, also based on Object Teams, to make up for this disadvantage (see Section 7).

6 Related Work

In [20], the Model-View-Controller paradigm is reported as another domain for the use of Object Teams. Despite the different domains, this work has many parallels to our testing technique. In both approaches, roles share a common base. In our approach, a base object is shared by different statechart objects, in [20] a model object is shared by different controller and view objects. The statechart implementation can be regarded as an abstract view of a given system.

There are some approaches that use aspect-oriented techniques for program instrumentation. Most of them focus on monitoring or testing systems, e.g. [4], [5], [15], [14] and [2]. An exception is [6], which uses program instrumentation for reverse engineering.

All of these approaches are based on AspectJ [10]. The advantages of our approach based on Object Teams result from the characteristics of Object Teams. AspectJ weaves code statically at compile time. All existing classes have to be recompiled. Ther is no activation mechanism for aspect code at runtime as in Object Teams. On the other hand, AspectJ has a complex join point language. As mentioned in Section 4.5, this kind of join point language is not needed for our purposes. Base methods can easily be enumerated by the code generator.

A similar aspect-oriented approach for testing based on statechart specifications is found in [2]. This approach focuses on testing Java components. No details are given as to how it deals with statechart features like history and parallelism. The presented example includes only hierarchy, which is flattened before testing.

Load-time weaving for test instrumentation is presented in [13]. Unlike our approach, instrumentation is done for code coverage. Generation of the instrumentation code is integrated in the load-time weaver. In our approach, we propose a separation of load-time weaving and code generation, which leads to aspect-oriented source code. Thus our technique can easily be applied to other instrumentation techniques or other programming languages. The load-time weaver remains unchanged, only the code generator has to be replaced or extended.

7 Summary and Outlook

We have presented an implementation technique for statecharts using the Object Teams programming model. Aspect-oriented programming supports instrumentation for test purposes. The Object Teams programming model also offers additional features that support a simple strategy for statechart implementation.

In the future, we wish to extend our approach to test query methods as well. The test could be extended by first testing query methods and then using the tested query methods to test update methods, as proposed in [1].

Using the same technique, the validation of pre- and postconditions and class invariants in OCL can be integrated and combined with our statechart implementation. The UML 2.0 defines clear semantics for the combination of OCL constraints and statecharts.

Languages other than Java should be supported by our approach. Of interest here is the development of an implementation of Object Teams for CORBA components. Our approach could then be applied to component software.

References

[1] R. V. Binder. *Testing Object-Oriented Systems*. Object Technology Series. Addison-Wesley, 2000.

[2] J.-M. Bruel, J. Araújo, A. Moreira, and A. Royer. Using Aspects to Develop Built-In Tests for Components. In *AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, USA, 2003.

[3] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[4] M. Deters and R. K. Cytron. Introduction of Program Instrumentation using Aspects. In *Workshop of Advanced Separation of Concerns in Object-Oriented Systems, 16th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM Sigplan Notices, Tampa, USA, 2001.

[5] R. E. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. In *Workshop on Foundations of Aspect-Oriented Languages, 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, Netherlands, 2002.

[6] T. Gschwind and J. Oberleitner. Improving Dynamic Data Analysis with Aspect-Oriented Programming. *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, 2003.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8), 1987.

[8] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays Conference)*, volume 2591 of *Lecture Notes In Computer Science*, Erfurt, Germany, 2002. Springer-Verlag.

[9] Jac Homepage. http://jac.objectweb.org/.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, Hungary, 2001. Springer-Verlag.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finland, 1997. Springer-Verlag.

[12] G. Kniesel. Objects Don't Migrate: Perspectives on Objects with Roles. Technical report, Universiät Bonn, Germany, 1996.

[13] G. Kniesel and M. Austermann. CC4J - Code Coverage for Java: A Load-Time Adaption Success Story. In *Component Deployment (CD)*, volume 2370 of *Lecture Notes in Computer Science*, Berlin, Germany, 2002. Springer-Verlag.

[14] T. Low. Designing, Modelling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). In *Workshop on Aspect-Oriented Modeling with UML, 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, Netherlands, 2002.

[15] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *Proceedings of The 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC)*, Crystal City, USA, 2002.

[16] B. Marick. *The Craft of Software Testing (Subsystem Testing)*. Prentice Hall, 1995.

[17] Object Teams Homepage. http://www.objectteams.org.

[18] M. Richters and M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, USA, 2003.

[19] *Unified Modeling Language Specifications, Version 2.0.* Object Management Group (OMG), *http://www.uml.org*, 2004.

[20] M. Veit and S. Herrmann. Model-View-Controller and Object Teams: A Perfect Match of Paradigms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston, USA, 2003.

[21] *XML Metadata Interchange Specification, Version 2.0.* Object Management Group (OMG), *http://www.uml.org*, 2003.