Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation

Stephan Herrmann Christine Hundt* Katharina Mehner* Jan Wloka*
Technical University of Berlin Fraunhofer FIRST
{stephan,resix,mehner}@cs.tu-berlin.de jan.wloka@first.fhg.de

ABSTRACT

Many aspect-oriented programming languages employ static transformations in order to produce the executable system. Some aspects, however, should only be effective if certain conditions are fulfilled that can only be evaluated at runtime. The naïve approach of using conditionals within the advice code easily leads to scattering and tangling regarding these conditionals, suggesting that they should be separated from the advice code. In this paper we analyze how aspects can be made conditional, i.e. how their effect can be controlled based on runtime values.

We present an extension to the ObjectTeams/Java programming language that provides a flexible means for controlling the instantiation and activation of an aspect implemented by a "team" and its roles by means of *guard predicates*. We discuss different points in a program for which we consider a guard predicate suitable. We argue that this approach is more general than the ways other aspect languages control aspect instantiation and activation. Our approach makes use of the fact that in Object Teams aspects are implemented as first-class objects, which have full support for inheritance and polymorphism.

1. MOTIVATION

The goal of AOP is to reduce tangling and scattering. AOP has successfully contributed to separating orthogonal and crosscutting code into aspect modules. Besides statically defined and deployed aspects, dynamism of aspects increasingly attracts researchers' interest. In some situations, it is desirable to invoke or change aspect behavior based on the dynamics of program execution.

In previous work, we have identified three dimensions to classify dynamism of aspects [14].

- The dimension of aspect *dynamism* addresses conceptually different life-cycles for aspects. Here we distinguish (a) static and dynamic definition of aspects, (b) static definitions of aspects which can be added and removed at runtime, and (c) static and dynamic activation of aspects.
- The dimension of *adaptation scope* distinguishes between type level and instance level control, assuming that aspects can be instantiated like classes in object-oriented languages.
- The dimension of *weaving* distinguishes between different points in time for the technical weaving process ranging from pre-compile time weaving to runtime weaving.

To give programmers maximal control over aspect dynamism it is desirable that these dimensions be clearly distinguished in language support. Therefore, we examine how this can be achieved in the context of the core language elements which have already been identified for AOP: *advice* (or comparable elements) containing the actual code of the crosscutting concern, and *pointcuts* for matching events in the execution of a program.

Existing AOP approaches support the above dimensions to different extents and in different ways. We observe deficiencies especially related to the concepts of dynamic activation control and instance level activation.

In approaches with static weaving and without explicit support for dynamic activation nor instance specific activation, these concepts have to be simulated using conditionals such as `if`-statements. Hence, advice code becomes tangled with code for activation control. This is especially dissatisfying because AOP made a start in avoiding one of the principal maintenance nightmares: nested conditionals spreading all over the code, resulting in programs that are more concerned with deciding what to do than with doing.

Approaches which have taken a step further support for separation of concerns include specialized support for controlling instantiation and activation of aspects. In AspectJ [16] keywords like `perthis` and `pertarget` allow to control aspect instantiation, but only a limited set of pre-defined strategies is available. Arbitrary querying of state is possible with the `if`-pointcut [16], but here pointcuts tangle the interceptions of events with activation control.

We feel that the separation of conditionals and actions can be taken one step further by elevating the conditions used to control different kinds of activation to a prominent language feature. Between join points and the action attached to them, this paper suggests a third concept: *guard predicates*. The main contribution of this paper is to report on an experiment integrating guard predicates into the aspect-oriented programming language ObjectTeams/Java.

*This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

Relating AOP to more fundamental concepts of programming should help to identify a general system of coordinates for describing AOP languages, which should eventually lead to a common theory of AOP. Guard predicates are a candidate for such a fundamental concept.

After providing explicit support for guard predicates, other parts of AOP languages — most notably join point languages — can be stripped down and will eventually reveal those atomic concepts that are new in AOP.

The paper is structured as follows: Sect. 2 gives a brief introduction to the language ObjectTeams/Java. Sect. 3 motivates the need for more flexible control over aspect activation. Sect. 4.1 explains how we integrate guard predicates into ObjectTeams/Java for controlling aspect activation. Sect. 4.2 relates guard predicates to issues of instantiation. Sect. 5 discusses the results in the context of related work. In Sect. 6 we conclude and outline some future work.

2. AN OVERVIEW OF OBJECT TEAMS

Before introducing the integration of guard predicates into ObjectTeams/Java, we give a brief overview of this language (see also [10, 20]). Object Teams introduces two new kinds of classes: `teams` and `roles`. A role is used to decorate an existing class, called its *base*. *Teams* are a structural concept for encapsulating collaborating roles. An instance of a team encapsulates a set of role instances.

A role instance intercepts method calls to the decorated object if a *callin* method binding between a role method and one or more base methods has been specified. A callin can be of type *before*, *after*, or *replace* (similar to an advice weave as found in AspectJ [16]) and inserts additional behavior into the control flow. A role together with its enclosing team can therefore be seen as an aspect.

Callin bindings of a role are only effective if the enclosing team has been explicitly activated. When a callin-bound base method is called the base object is "translated" to the corresponding role. The translation is called *lifting* [12]. Next, the bound role method is called. The lifting translation must be aware of any existing role-base pairs. To this end, every team internally has a role registry for storing and retrieving roles. If no appropriate role is found it is created on demand and stored in the registry.

2.1 Example

The following source code illustrates the concepts *team*, *role* and *callin bindings* by the example of an automatic teller machine (ATM). We start with a most simple `Account` class and step-by-step introduce new requirements which are to be added non-invasively.

```
1  public class Account {
2    private int balance;
3    public boolean debit(int amount) {
4      if (!(amount>balance)) {
5        balance -= amount;
6        return true;
7      }
8      return false;
9    }
10   ...
11 }
```

An `Account` is a regular (base) class offering the functionality of `debit`ing an amount (among other things).

```
12 public team class ATM {
13   public int payCash(
14         Account account, int amount)
15   {
16     boolean ok = account.debit(amount);
17     if (ok) return amount;
18       else return 0;
19   }
20   // role definition:
21   protected class FeeAccount playedBy Account {
22     callin boolean debitWithFee(int amount) {
23       int fee = calculateFee(amount);
24       return base.debitWithFee(fee+amount);
25     }
26     // replace callin binding:
27     debitWithFee <- replace debit;
28     int calculateFee(int amount) {...}
29   }
30 }
```

The `ATM` allows to withdraw money via the method `payCash(..)`[1]. So far only standard Java features have been used. However, the `team` modifier in line 12 denotes that this class is realized as a *team* class. An additional requirement to be fulfilled is to collect a fee when withdrawing money. This is realized by the *role* `FeeAccount` which decorates the base class `Account`. Inner classes of teams are per definition roles. The decoration relationship is stated by the `playedBy` keyword (see line 21).

Via a replace callin binding the role method `debitWithFee` is dedicated to intercept the base method `debit` (see line 27). The effect of this binding is the following: Whenever `Account.debit` is called the target is lifted to a `FeeAccount` role. The control flow is redirected to its `debitWithFee` method, which calculates the fee and calls the original base method with the increased amount via `base.debitWithFee`. In analogy to super calls, a base call uses the name and signature of the enclosing role method.[2] This *base call* is comparable to the *proceed* in AspectJ.

3. MORE CONTROL OVER ASPECTS

So far, the introduced concepts of Object Teams allow general control over the activity of aspects by activating and deactivating teams and all contained roles. In many cases, this may be enough, but in more complex applications a more flexible control will be needed.

Reconsidering the ATM example, it is questionable if the general collection of fees is fair. The role-base binding stated by the `playedBy`-clause generally attaches the fee-collecting role to every `Account` (base-)object. Actually, an ATM belongs to a certain bank and only for withdrawal from a foreign account an additional fee is debited. Let's assume also, that it should not be possible to query the balance of a foreign account.

[1] Authorization needs are omitted for simplicity.

[2] This syntax is further motivated by the goal to keep the callin method independent of any bindings to (possibly multiple) base methods.

In the following, a more realistic behavior for an ATM is achieved. The addition of a `Bank` attribute to an `Account` as well as to an `ATM` is straightforward. More importantly, in order to fine-tune the general behavior, we need a more flexible control over the activity of the aspect encapsulated in the `FeeAccount` role. So far, this is only possible with explicit hand-coded if-statements in the advice code. The following source code only shows the relevant additions compared to the first version[3]:

```
31 public class Account {
32   private Bank myBank;
33   public Bank getBank() { return myBank; }
34   ...
35 }

37 public team class ATM {
38   private Bank bank;
39   public int payCash(Account ac, int am) {...}
40   public int getBalance(Account account) {...}

42   public class ForeignAccount playedBy Account
43   {
44     callin void debitWithFee(int amount) {
45       if (ATM.this.bank.equals(getBank())) {
46         base.debitWithFee(amount);
47       } else {
48         int fee = calculateFee(amount);
49         base.debitWithFee(fee+amount));
50       }
51     }
52     callin int checkedGetBalance() {
53       if (ATM.this.bank.equals(getBank())) {
54         return base.checkedGetBalance();
55       } else {
56         throw new AccException("foreign account");
57       }
58     }
59     debitWithFee       <- replace debit;
60     checkedGetBalance <- replace getBalance;
61     abstract Bank getBank();
62     getBank            -> getBank;
63   }
64 }
```

The new requirement of restricting the query of balance, is realized as another callin method denying the call to `getBalance` for foreign accounts (see line 52). In line 45 and line 53 if-statements are used to ensure the desired behavior. If the callin methods are called for own accounts they just forward to the original base method (see lines 46 and 54). Otherwise, they perform the special behavior of foreign accounts. Lines 61 and 62 are used to define and bind the method `ForeignAccount.getBank` via *callout*[4] to the corresponding base method, which is needed in the conditions.

Although the required behavior is now achieved some bad smells can be observed. First of all, the condition is *tangled* with the advice code. Secondly, we see a *scattering* of the (identical) condition over the two role methods. What we really want to state is that *the role `ForeignAccount` is only valid for accounts belonging to a different bank than the ATM.*

[3]Note, that the role is now called ForeignAccount, because this name better states its meaning.

[4]*Callout* bindings declaratively specify the forwarding from a role method to a base method. Reverse to callin bindings, which are denoted with the symbol "`<-`", callout bindings use the symbol "`->`".

We need a more flexible control over the activity of aspects and we want to get rid of the drawbacks observed in the hand-coded version. The following section shows how the integration of guard predicates can achieve this goal in a more elegant way.

4. GUARD PREDICATES IN OBJECTTEAMS/JAVA

In the previous section we have illustrated the problem of condition tangling as it pertains even if some form of AOP is applied. In order to remedy this problem we go back to fundamental concepts of programming and try to identify ECA elements *("event-condition-action")* in AOP:

- We interpret join points as places in a program that emit *events*.
- We consider advice weaving as an adaptation of the *action* that happens at join points.
- The contribution of this paper is the prominent role which *conditions* play as a middleman between events and actions.

In order to distinguish conditions in this sense from conditions in the imperative part of the language we will use the notion of *guard predicates*. In slight adaptation of existing ECA formalisms, we define that a guard predicate is to determine whether the firing of a given event should actually cause the adapted action to be performed. Otherwise the original behavior is performed unchanged.

In this section, we will first introduce the places where guards can be attached to specific entities of an ObjectTeams/Java program (4.1) and show how aspect activation can be controlled using guards. Also aspect instantiation can be controlled by guards as we present in Sect. 4.2. We will then relate guards to other concepts in the language (Sect. 4.3).

4.1 Controlling aspect activation

Callin-bindings as described in the previous section cause the control flow to leave a base object and enter the corresponding role object. In our model, guard predicates are used to filter these added control flows. The effect of a guard evaluating to `false` is that the current callin trigger is rejected and the base behavior is performed unmodified. If a guard evaluates to `true` (or if no guard is present) the callin trigger is effective and the adaptation defined by the corresponding role is applied. We say, an aspect (team and its roles) is active if its callin bindings are effective. Thus, guard predicates are a means to control aspect activation.

We support guard predicates at four levels of granularity. The first level refers to method bindings. This is a straight forward adoption of tests for dynamic properties as they are supported, e.g., by the `if` pointcut designator in AspectJ. Semantically, guards of method bindings do not differ from the `if` designator, we just syntactically separate the guard to be evaluated at runtime from the statically computed set of join points.

Here is the syntax of a guarded method binding (details of the `when` clause will be discussed below):

```
protected class MyRole playedBy MyBase {
  ...
  /* callin method binding with guard */
  void rmeth(int x) <- after void bmeth(int y)
      when (boolean expr. using this(refers to role) and x);
}
```

As a first means for generalization we also support guards attached to a method. In ObjectTeams/Java the action to be performed at a join point is defined by regular methods[5]. This way several method bindings may bind to the same role method. Attaching a guard predicate to a role method has the semantics of attaching the guard to all method bindings of this method. Only if a method is invoked due to a callin binding, guards are evaluated and may reject the current trigger.

In the next level of granularity guards are attached to role classes such as in the header of the following role class:

```
protected class MyRole playedBy MyBase
   when(boolean expression using this(refers to role))
{
   //class body omitted
}
```

In this case the idea is to enable or disable a role instance based on the evaluation of a guard predicate. The guard is evaluated before any callin bindings to an instance of the current role class are considered.

Finally, by attaching a guard predicate to a team class, all role classes of the team are consistently enabled or disabled based on the evaluation of the guard predicate.

The following table summarizes the locations where guard predicates can be attached and describes the respective effect. Note that guard predicates are only checked if a method is called via callin.

location/level	affected role methods
role method binding	this role method if called due to this binding
role method	this role method
role	all role methods of this role
team	all role methods of all roles in this team

4.1.1 Scope of guard predicates

First of all, all predicates may access the current team instance, because only with an existing instance of the team, any callin bindings are considered at runtime. Within a team level guard, the team instance is simply referred to as `this`. In practical examples this proves very useful, since a team instance can be used to store arbitrary context information that is valuable for evaluating the predicate. We will further discuss the role of teams for *context based programming* in Sect. 5.4.

[5]Only for `replace` callin bindings a special kind of method is required that is marked by the `callin` modifier.

All guards at the level of role classes or below also have access to the role instance whose method is about to be invoked. Also the role instance can be used to store context or history information, that will help to formulate the predicate. In a role level guard, `this` refers to the role instance and the notation *MyTeam*`.this` to the enclosing team. Guards of methods and method bindings also expose parameter values (except for the shorthand variant of callin bindings, which may omit method signatures).

With a role level guard predicate the ATM example from section 3 can now be rewritten as follows:

```
65 public team class ATM {
66   ...
67   public class ForeignAccount
68     playedBy Account
69     when (!(ATM.this.bank.equals(getBank()))
70   {
71     callin void debitWithFee(int amount) {
72       int fee = calculateFee(amount);
73       base.debitWithFee(fee+amount));
74     }
75     callin int checkedGetBalance() {
76       throw new AccException("foreign account");
77     }
78     debitWithFee      <- replace debit;
79     checkedGetBalance <- replace getBalance;
80     abstract Bank getBank();
81     getBank           -> getBank;
82   }
83 }
```

The guard predicate attached to the `ForeignAccount` role in line 69 ensures that this role and therefore all its callin bindings are only effective for foreign accounts. This way, all previously necessary partial checks are substituted by one specification, expressing exactly our intended requirement. Also a clean separation of advice code and activation conditions is achieved.

4.2 Controlling aspect instantiation

In most AOP languages, instantiation of aspects is fairly different from instantiation of regular objects. Commonly, aspects are instantiated automatically when needed without using a `new` expression. Aspect instances are yet important because they allow to store state of the aspect. Technically, aspect behavior is usually implemented as instance methods of the aspect, with the only exception of "advice" which lacks some important properties of methods.

Aspect instantiation attaches an aspect instance (in Object Teams: role object) to one or more base objects. The Object Teams approach pays very close attention to providing all benefits of instantiation also to aspect related language features. Therefore, actual role objects are used instead of augmenting base classes with additional features. In this context, role instantiation is such an essential concept that the used strategy should not be hard-coded into the language as a fixed set of options. Rather should programmers have the chance to effectively control the strategy of role creation. Different from regular objects, the *default* for role objects is an implicit creation on demand. However, client code may force the creation of additional roles and may prevent the default creation of roles.

4.2.1 Positive control

Before we demonstrate how guards may prevent role creation, let us briefly review the means ObjectTeams/Java provides for triggering role creation.

The default constructor of each *bound*[6] role class expects as its only parameter an instance of the connected base class. This constructor can be used from within the context of a team. The language implementation ensures that role objects created using this constructor are also registered in the team for consistent integration with the mechanism of lifting.

As a second option, methods of a team may declare an argument with a dual type: the caller is required to pass a base object yet the method body receives a role object. The signature for this declaration looks as follows: `teamMethod(MyBase as MyRole r)`. When calling such a method, on demand creation of roles remains implicit, however, the existence of a role object for the given base object is ensured upon entry to this method.

4.2.2 Negative control

Whenever a callin binding is about to invoke a role method, first the base object is lifted to its corresponding role object. If that role object is not yet present in the team's registry, a fresh role is automatically created. The guards we have presented so far reside at the role side of a binding. Thus, lifting happens before evaluation of the predicate. This has the advantage, that information in a pre-existing role object can be used within the predicate.

By attaching the modifier `base` to any guard predicate, the semantics is changed such that the predicate is evaluated at the base side, i.e.: prior to lifting. If a base guard evaluates to `false` no lifting occurs and the aspect has no effect at all, not even creating a role.

It is important to see, that lifting potentially has side effects. First, creation of a role is an implicit side effect, affecting the team's registry. Second, a custom role constructor may implement arbitrary side effects. Only a base guard ensures, that in the negative case no side effect has been caused. A future version of the compiler for ObjectTeams/Java will also include an analysis, whether guard predicates are in fact free of side effects.[7] Syntactically, a guard predicate may invoke any boolean method in scope.

As an example, consider again the case of a predicate attached to a method binding:

```
protected class MyRole playedBy MyBase {
  ...
  /* callin method binding with base guard */
  void rmeth(int x) <- after void bmeth(int y)
       // optional parameter mapping omitted
       base when (boolean expression using base and y);
}
```

Please note that the scope of a base predicate is at the base side. The base object at which the join point trigger occurred can be accessed using the special identifier `base`. Method parameters available in the predicate are those of the base method (here: `y`). In anticipation of the control flow into an existing team instance, the team causing the callin interception can be accessed using `MyTeam.this`.

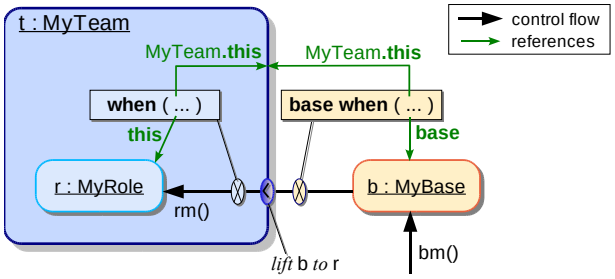

Figure 1: Characteristics of Guards in Object Teams

Figure 1 shows which objects a predicate can access, independently from the level of granularity. A call to `MyBase` intercepted by `MyRole` is controlled by a guard predicate. A `when` predicate can refer to the involved role via `this`, while a `base when` has access to the corresponding base object via `base`.

Now it is possible to further enhance the ATM example by controlling the instantiation of the `ForeignAccount` role. If an account belongs to the same bank as the ATM, it will never need the extra functionality of the `ForeignAccount` role. Using a base side guard prevents the creation of roles for such accounts, thus preventing all possible side effects.

```
84 public team class ATM {
85   ...
86   public class ForeignAccount
87     playedBy Account
88     base when (
89         !(ATM.this.bank.equals(base.getBank()))
90   {
91     callin void debitWithFee(int amount) {...}
92     callin int checkedGetBalance() {...}
93     ...
94   }
95 }
```

Line 89 shows the mentioned `base when` predicate. We now have direct access to the `Account.getBank()` method through `base` in the guard predicate and do no longer need the indirect access via callout.

4.3 Relating guards to other concepts

The idea of guard predicates originates from the fundamental issue of controlling the flow of execution. The previous section has shown how guards in ObjectTeams/Java can be used to control also the instantiation of role objects where the default behavior is otherwise an automatic instantiation per team and per base object. In this section we investigate the relationship of guards to other fundamental concepts of (object-oriented) programming.

4.3.1 Guard combinations

When introducing predicates as a new language feature, rather than relying on regular methods and method calls

[6] A role with a `playedBy` clause is *bound*.

[7] The need for side effect analysis fits nicely with other work on Object Teams where a concept of `readonly` interfaces is used to enforce representation encapsulation [11].

only, care must be taken, not to break polymorphism and dynamic dispatch. More specifically, predicates should be able to exploit the inheritance structure of roles and base classes.

In some situations more than one guard watches over the callin to a single role method. On the one hand this can occur if a role predicate is inherited and overridden in the subclass. On the other hand a second dimension of combination appears if guards at different levels of granularity affect the same role method. In both cases a reasonable strategy for combining these predicates is needed.

Guards at different positions of the inheritance hierarchy: The semantics of object-oriented inheritance is to refine the behavior in subclasses. In analogy to this, guards in a subclasses are combined by logical `and` with the guards of its superclass. Thus, the subclass can only further refine an inherited guard predicate.

Guards at different levels of granularity: If looking at a team-level guard predicate, the programmer should be allowed to assume that this predicate is effective for the whole team. For reasons of locality guard predicates of inner entities may not invalidate outer ones. This means that inner guard predicates may only refine the conditions of outer ones. Thus, they are again combined by a logical `and` with all outer predicates.

The case of *polymorphism* only seems to be more difficult when considering base guards. Here, no role instance exists yet, thus dynamic dispatch to the most suitable role class seems impossible. However, guard predicates are integrated into the language in such a way, that dispatching can indeed exploit all kinds of polymorphism involved. A detailed description of this integration would require to dive into the mechanism of *smart lifting* as defined in [12], which establishes a new level of polymorphism called *translation polymorphism*. Such details are, however, beyond the scope of this paper. For the programmer, the situation could be interpreted as a role instance that exists *virtually*, whereas the point in time of evaluating the base guard is early enough to prevent the role object from materializing.

4.3.2 Reflection

Object Teams also aims at supporting reflection with respect to teams, roles, and role-base relationships. As mentioned before, each team instance internally has a registry of known role objects indexed by their base object. Programmers may make use of this registry using the following reflective methods defined in `org.objectteams.Team`[8]. The last one of these methods does not access the registry but can be used to inspect whether a control flow has already been intercepted by at least one callin binding.

```
boolean hasRole (Object aBase);
boolean hasRole (Object aBase,
                 Class  expectedRole);
Object  getRole (Object aBase,
                 Class  expectedRole);
void    unregisterRole(Object aRole);
boolean isExecutingCallin();
```

[8]The predefined super class of all team classes

It is desirable and possible to use these methods within guards. These methods allow to write the specification of guards in a more concise and more expressive way. Determined by the signature, the first three methods can only be used in a base-level guard because they require a reference to a base object.

4.3.3 Example using reflection

We now conclude the introduction of guard predicates in Object Teams with an example which summarizes several concepts and illustrates the use of reflective predicates.

We want to extend the account example as follows: If an account is registered to collect a special bonus, every time an amount of more than 1000 is deposited, additional 1% of the amount is credited. To this end, we create another team `SpecialConditions` which is responsible for the new functionality.

```
96  public class Account {
97    private int balance;
98    public void credit(int amount) {
99      balance += amount;
100   }
101   ...
102 }

104 public team class SpecialConditions {
105   public void participate
106                (Account as BonusAccount ba) {}

108   public class BonusAccount
109     playedBy Account
110     base when(SpecialConditions.this.hasRole(
111                       base, BonusAccount.class))
112   {
113     public void creditBonus(int amount)
114       when (amount > 1000)
115     {
116       base.creditBonus(amount+(amount/100));
117     }
118     creditBonus <- replace credit;
119   }
120   ...
121 }
```

This team provides a registration method `participate` which is used to register an account for the special conditions (see line 105). Here, the explicit role creation mechanism as described in 4.2.1 is used.

The base side guard predicate in line 110 checks, if the base object already has a role in this team. If this is not the case it prevents lifting (and thus role creation). In combination with the registration method this means that `BonusAccount` roles are never created automatically via lifting but have to be explicitly registered first. This is exactly what we want for our bonus collection.

The callin method in line 113 implements the collection of the bonus. It replaces the original `Account.credit` method (see line 118) and performs a base call with the increased amount of money. In line 114 we use an additional predicate to ensure that bonus is only credited for amounts greater than 1000.

The following code snippet shows a usage example. Only

the account `a1` is registered for the bonus collection, while `a2` does not have any special conditions.

```
... // main method:
Account a1 = new Account();
Account a2 = new Account();
SpecialConditions sc = new SpecialConditions();
sc.activate();
sc.participate(a1);
a1.credit(2000); // -> balance += 2020
a2.credit(2000); // -> balance += 2000
...
```

5. COMPARISON TO RELATED WORK

There is a long tradition in specifying program behavior using triplets of events, conditions and actions (ECA). This has been made popular by the statecharts notation [9]. In this context the fundamentally different nature of events and conditions has been elaborated in depth. This suggests that also programming languages might benefit from a clear separation of pure events that happen at a specific point in time versus conditions reasoning about state that persists over a span of time.

The idea of ECA triplets has been taken up in the active database context [5]. Typically, the events, also called triggers, are changes in the database such as an insertion into a table. The condition evaluation determines whether a certain action will be executed or not.

5.1 Event-based programming

A similarity of ECA systems and AOP systems has been discussed in [4]. Also *event-based AOP* (EAOP) has been proposed ([6]). Here, aspects are defined in terms of events emitted during program execution. Identifying each hit of a join-point with an event that can trigger further behavior is well in line with the common understanding that AOP opens new options for reasoning about points in the control flow of a program. However, the EAOP approach does not provide explicit support for conditions, leaving this to the imperative part of the language. This leads to the tangling of conditions and actions, which motivated the introduction of guards predicates.

5.2 Predicates in programming

The expressiveness of predicates has been exploited for object-oriented programs by the concept of *predicate dispatch* [3, 19].

Predicate Classes were first introduced as an extension of the programming language Cecil [3]. They complement normal classes. An object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. Any of these predicate classes can override a method of that object if it formulates more specific conditions in a predicate. Consequently, method look-up depends not only on the dynamic types but also on dynamic object states (as captured by predicates). In this approach, ill-defined predicates could lead to situations where several methods or no method at all will be selected during method look-up producing fatal run-time errors. To prevent such errors, programmers have to follow specific disciplines.

Recently, an extension of Java with *predicate-based dispatch* called JPred has also been presented [19]. In this approach the problem of ambiguity and incompleteness of predicates is addressed by modular type-checking using a theorem prover.

Both approaches have the benefit that they avoid large switch statements and instance-of checks in individual methods, which are also difficult to maintain. Instead, each method is responsible for one case only, and the most specific method according to the most specific predicate is chosen. Such a system is easier to extend by new cases as each concern is encapsulated in a method. This benefit has also been identified by [22].

While these approaches improve the separation of concerns and flexibility, at the same time they introduce a complex analysis problem that either has to be dealt with by the programmer or by employing tool support based on formal methods. The question remains whether this is always tractable for the programmer.

By contrast, guard predicates in ObjectTeams/Java do not select *necessary* methods for execution, but only filter whether *additional* aspect behavior will be triggered or not (before/after). If `replace` bindings are involved, semantic consistency is of course subject to the implementation of the replacing callin method, but this problem is intrinsic to replace/around aspects and even to regular method overriding.[9] In our approach each explicit method call (base level) will always select exactly one method. In *no* situation the evaluation of guards will result in a run-time exception signaling the inability to dispatch the method call.

Although this type safety in ObjectTeams/Java is achieved by constraining the original idea of predicate dispatch, a role class can be used in quite similar ways as a predicate class. The difference is, that a base object does not completely change its type based on its state, but it may acquire additional roles which can augment its state and adapt its behavior.

5.3 Guards in Aspect-Oriented Programming

The *Rondo* model [18] defines aspect–like elements called *adjustments*. In Rondo adjustments have an event and/or a condition. Events are explicitly sent using a `raise(...)` primitive. When the event associated with an adjustment is fired, the adjustment is added to a given base module (another adjustment). While this model has an explicit distinction between events and conditions, both parts are optional. Also a strategy for evaluating conditions on demand is included. Unfortunately, this model has not been followed up after its definition.

AspectS [13] supports *activation blocks* by which method wrappers can be configured to filter trigger events based on program state. The examples seem to imply that activation blocks are designed for the purpose of defining join-points from a set of pre-defined activation blocks. While the concept is actually quite similar to guard predicates in Object Teams, AspectS supports activation blocks only at the level

[9] In ObjectTeams/Java, flow analysis is used to warn about replace callin methods, that do not issue exactly one base call on each control flow.

of method wrappers, thus leaving activation at the level of larger program units to the discipline of the programmer[10].

In the *Composition Filters* approach [2], filters are used to achieve method interception. Filters can accept certain messages and thus intercept their execution. The matching process is based on the target object, the message selector and an optional *condition*. Conditions are reusable side-effect free boolean methods, which offer an abstraction of the state of the implementation object. They allow the separation of the filter itself from the implementation details of the object. Thereby the reusability of the filter is increased.

The successful application of predicates in the Composition Filters approach suggests that predicates are indeed a desirable concept for AOP languages. In Composition Filters, filters are very specific entities, existing for the sole purpose of filtering messages. Conditions serve the purpose of controlling the flow of execution. By contrast, team and roles in ObjectTeams/Java are regular classes producing regular instances.

We could not find prior work concerning AOP where predicates where used to control aspect instantiation as it is supported by `base` predicates in ObjectTeams/Java.

5.4 Context based programming

Teams define a context for all executions of role methods. It is an advantage that this context is available already when evaluating guards. When acting as a mediator between its role objects, a team can be used to detect certain execution patterns at the base level. Mediated by a team, a sequence of callin triggers can be regarded as a *situation*, which includes a memory of passed triggers as well as a place for storing data that where collected along the path of execution. In this setting, guards are used to filter events depending on the current state of the team and on any data available at the join points. Thus, Object Teams supports different levels of designing program *modes*: Team activation defines which team instances influence the system at a given point in time. Guards allow an active team to focus on specific subsets of emitted events. This could mean to consider certain base objects only, it can exclude method calls based on their parameter values, or different role classes can be turned on and off in different modes of a team.

Other approaches exist that allow to define behavior based on some context information. Lasagne [15, 24] defines per-client contexts for distributed applications. Also, context relations [23] and environmental acquisition [7] share some ideas with Object Teams. Caesar [17] and Chameleon [8] are other programming models similar to Object Teams. Of these approaches, only Lasagne supports a limited notion of predicate dispatch. In Lasagne a deployment-specific interceptor can be used for "global predicate dispatch". On the other extreme, methods may be guarded by a precondition for so-called "local predicate dispatch". An in-depth comparison of our approach with the details of the recently published thesis [24] will certainly be very instructive, but from a first look it seems, that predicates in Lasagne cannot be used to control aspect instantiation.

It is a contribution of our approach to provide guards at four different levels of granularity and at different points in the control flow (at the base vs. at the role). Also among the presented approaches the additional dispatch mechanism of Object Teams (lifting) is unique. By their close connection to the lifting mechanism, guards can also be used to control role instantiation, which is not parallelled in the other approaches.

6. CONCLUSION

We have introduced guard predicates as a clean way to control the activation and instantiation of aspects. The feature has been implemented in the OTDT, the eclipse based Object Teams Development Tooling[21]. First experiments support the predicted benefits.

We are convinced that explicit support for guard predicates in aspect languages helps to untangle language concepts, thus leading to more orthogonal language designs. In this sense, the introduction of guards as a separate construct allows to construct a leaner join point language. It was an important step of AOP to extract many conditionals from the imperative code, because cluttered conditionals may become a real danger to comprehending a program over its evolution. We take extracting conditionals one step further by introducing guard predicates at a prominent level in the language.

The number of approaches that use guards as one of their core concepts illustrates the fundamental importance of guards. With explicit language support for guards, AOP can actually be explained as a specific kind of ECA system. Join points define *events* that are emitted whenever the control flow passes a certain point. Guards define the *conditions* by which events are filtered. *Action* in aspect languages actually means an adaptation of an existing action in the base program further specified as either before, after or replace.

Of course, role guards (`when`) could always be encoded by weaving the condition into all relevant role methods. We see three main reasons for raising guards to the level of a language feature:

1. Guards at the level of classes (role and team) avoid the scattering that would occur in the hand-coded solution.
2. Guards as a language feature encourage programmers to think in terms of modes, conditions, situations etc.
3. Base-level guards allow to prevent the creation of role objects that should not exist, an effect that can only be obtained by a close connection to the runtime semantics of callin binding and lifting.

After the introduction of guard predicates, ObjectTeams/Java now provides these styles of controlling aspect instantiation: First, teams are instantiated and activated explicitly, which gives programmatic control not only over which

[10]Being a MOP-based approach, AspectS certainly has the flexibility to support many different styles, but it can give no static guarantees like team-level predicates, e.g., can.

aspects have an effect on the system, but also on their relative priority. Second, role creation can be triggered by constructor calls from within the team or by declared lifting in team-level methods. Third, role creation caused by a callin binding can be rejected by a guard. So the default behavior for roles is implicit creation on demand, but client code may explicitly add roles and prevent other roles from being created.

6.1 Future work

After academic experiments have identified many situations where guard predicates indeed facilitate very modular designs, an industrial case study using ObjectTeams/Java is currently in progress.

The language design of ObjectTeams/Java is not complete, yet. To date, the only way to specify join points is to enumerate methods by name. Therefor, a full analysis of orthogonality and expressiveness can not yet be done. However, we consider explicit support for guards more fundamental than the exact choice of a query language by which join points can be identified. With guards separated from join points we can now focus on which events actually should be observable.

7. REFERENCES

[1] M. Aksit, M. Mezini, and R. Unland, editors. *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

[2] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. ISBN 0-32-121976-.

[3] Craig Chambers. Predicate classes. In *Proc. of ECOOP'93*, 1993.

[4] M. Cilia, M. Haupt, M. Mezini, and A. Buchmann. The convergence of aop and active databases: Towards reactive middleware. In *Proceedings of 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *LNCS*, pages 169–188, 2003.

[5] K. R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of a ADBMS features. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 3–20. Springer, 1995.

[6] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[7] J. Gil and D. Lorenz. Environmental Acquistion — a new inheritance like abstraction mechanism. In *Proc. of OOPSLA'96*, pages 214–231. ACM, 1996.

[8] Kasper B. Graversen and Johannes Beyer. Chameleon, August 2002. Masters thesis. IT-University of Copenhagen.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] Stephan Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In Aksit et al. [1].

[11] Stephan Herrmann. Confinement and representation encapsulation in object teams. Technical Report 2004/06, Technical University Berlin, 2004.

[12] Stephan Herrmann. Translation polymorphism in Object Teams. Technical Report 2004/05, Technical University Berlin, 2004.

[13] Robert Hirschfeld. AspectS - aspect-oriented programming with squeak. In Aksit et al. [1].

[14] Christine Hundt. Introducing Dynamic AOP to Object Teams. Poster at European Interactive Workshop on Aspects in Software EIWAS'04, `http://www.topprax.de/EIWAS04`, 2004.

[15] B. Jørgensen and E Truyen. Evolution of collective object behavior in presence of simultaneous client-specific view. In *Proceedings of the 9th international Conference on Object-Oriented Information OOIS'03*, volume 2817 of *LNCS*, pages 18–32. Springer Verlag, 2003.

[16] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of AspectJ. In *Proc. of 15th ECOOP*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001.

[17] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proc. AOSD'03*, Boston, USA, March 2003. ACM Press.

[18] Mira Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publisher, 1998.

[19] T. Millstein. Practical predicate dispatch. In *Proceedings of OOPSLA 2004*, October 2004.

[20] Object Teams home page. http://www.ObjectTeams.org.

[21] Object Teams Development Tooling download page. http://www.ObjectTeams.org/distrib/otdt.html.

[22] Doug Orleans. Separating behavioral concerns with predicate dispatch, or, if statement considered harmful. In *Workshop Advanced Separation of Concerns in Object-oriented Systems at OOPSLA'01*, 2001.

[23] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.

[24] Eddy Truyen. *Dynamic and context-sensitive composition in distributed systems*. PhD thesis, Katholieke Universiteit Leuven, October 2004.