
Generic Aspect Languages – Needs, Options and Challenges

Günter Kniesel — Tobias Rho

*Dept. of Computer Science III
University of Bonn
Römerstr. 164
D-53117 Bonn, Germany
gk@cs.uni-bonn.de
rho@cs.uni-bonn.de*

ABSTRACT. Aspect-oriented programming languages promise to provide better modularity than pure object-oriented decomposition. A typical benefit of increased modularity is ease of maintenance, evolution and reuse. However, it has been noted by various researchers that many of the first generation aspect languages do not provide the degree of reusability initially hoped for.

In this paper, we argue that the problem is due to a lack of support for aspect genericity, respectively to a lack of support for a sufficiently powerful kind of genericity. We analyze various problems of non-generic aspect languages and define genericity to be the ability to concisely express aspect effects that vary depending on the context of a join point known at weave-time. We describe the design space for generic aspect languages in terms of five basic questions that a language designer should consider. Within this space we review the different proposals for generic aspect languages made so far and compare them by showing which part of the possible design space is explored within these languages. We identify the orthogonal needs for uniform and fine grained genericity in aspect languages and discuss in which ways these concepts go beyond previous approaches to generic aspects. Last but not least, we identify open research problems that are specific to aspect genericity or aggravated in the presence of genericity.

KEYWORDS: aspect-oriented programming, generic aspect language, uniform genericity, fine-grained genericity.

1. Introduction

Aspect oriented software development (AOSD) is a powerful new paradigm for separation of concerns. Functionality that would otherwise be scattered throughout a program because it cannot be modularized in the dominant decomposition (Ossher *et al.*, 2001a) of a particular language can be encapsulated into an aspect. Filman and Friedman (Filman *et al.*, 2000) characterize AOSD as a modularization concept with two desirable properties: quantification, the ability to declare actions to be applied consistently to many places of a program, and obliviousness, which means that the modified program entities do not need to be aware of being subject of aspect activity and do not need to provide special hooks for enabling such activity.

Obliviousness of base code greatly eases evolution, since changes in aspects do not require subsequent changes in base code. However, while base code is oblivious of aspects (or should at least be as oblivious as possible), aspects are not at all oblivious of base programs. Obviously, aspects must know about the base entities that they want to influence. It is the degree of knowledge and implied coupling that has turned out to be a problem in first generation aspect languages – *AspectJ* (Kiczales *et al.*, 2001), *HyperJ* (Ossher *et al.*, 2001a), *Composition Filters* (Aksit *et al.*, 1992), *DemeterJ* (Demeter) – and later approaches that adopted and evolved one of these archetypes, e.g. (Spinczyk *et al.*, 2002).

These languages introduce strong dependencies of aspects on base code by requiring aspects to *use concrete names* of types, classes, methods, and other entities from base programs. In these languages, the only mechanism to alleviate the dependence of aspects from base entities is the provision of wildcards¹ in the join point language². However, as noted by different researchers (deVolder 2001, Hassoun *et al.*, 2003, Hanenberg *et al.*, 2003a, Gybels *et al.*, 2003), the lexical dependencies introduced by wildcard-based join point languages negatively influence aspect applicability, conciseness, reuse and evolution.

This paper makes the following contributions to the state of the art of aspect-oriented language design:

- an analysis of the problems with aspect languages based exclusively on wildcard-matching (Section 2 and 3),
- a definition of generic aspect languages (Section 4),
- a thorough discussion of the design dimensions of generic aspect languages and a classification of current work in this area (Section 5),
- a discussion of research challenges imposed by generic aspects (Section 6).

¹ In AspectJ, for instance, ‘*’ matches type, method and field names (or parts of them), ‘..’ matches a parameter list, and ‘**’, matches portions of a fully qualified Java type name.

² The terms *crosscut language* () and *pointcut language* () are used to denote the aspect sublanguage that specifies where and when aspect actions should be executed.

1.1. Terminology

Before we delve into the sketched topics we first define a few terms that we need to make the following discussion language-independent.

Aspect effect. The term *aspect effect* denotes the changes that an aspect performs on a program or a program execution. *Effect specifications* correspond, for instance, to the inter-type declarations and advices of *AspectJ* (Kiczales *et al.*, 2001), filter types of *Composition Filters* (Aksit *et al.*, 1992), or composition specifications of *HyperJ* (Ossher *et al.*, 2001a).

Aspect predicate. An *aspect predicate* is a predicate that selects places in the program or events in the program execution where aspect effects should be applied. Predicates correspond, for instance, to the pointcuts of *AspectJ* and filter conditions of *Composition Filters*.

Aspect. An aspect is a module that contains specifications of aspect predicates and aspect effects. In many aspect languages, aspects can also contain base language elements (fields, methods). For simplicity, we will assume that this is not the case, since in most cases one can factor out the base language parts into base modules.

Base language. A base language is a conventional programming language that has no notion of aspects, aspect effects or aspect predicates. For instance, the base language of *HyperJ*, *AspectJ*, *Sally*, and *LogicAJ* is *Java*. *AspectC++* has C++ and *AspectC#* has C# as its base language. The composition filter model and its incarnation in the aspect language *Compose** has many different base languages.

Base module. A base module is a module (e.g. class or interface) containing only base language elements. The set of all base modules in a program is called the *base program*.

2. Limitations of Non-Generic Aspect Languages

A large class of current aspect languages allows references to base language entities (types, classes, methods, etc.) only via names or via patterns consisting of names and wildcards. We call them *wildcard-matching-based languages*. This class includes various derivatives of *AspectJ* (e.g. *AspectC++*, *AspectC#*) but also different instantiations of the *Composition Filters* model and *HyperJ* (Ossher *et al.*, 2001a). The following discussion of the problems arising from the sole reliance on wildcard matching is applicable to all the languages in this category.

2.1. Evolvability

It has been often pointed out that lexical dependencies make aspects highly sensitive to changes in base programs – see, for instance, (Tourwe *et al.*, 2003) and (Hananberg *et al.*, 2003b). If names in base programs are changed, the aspects that rely on those names either break or must be changed too. For instance, this is the

case if setter methods are described as methods whose name starts with the string ‘set’ and the naming convention is changed or abandoned entirely. Hanenberg, Unland and Oberschulte (Hanenberg *et al.*, 2003b) show that even small changes in base programs can require very complex follow-up changes in dependent aspects. Therefore, independent evolution of aspects and base programs is not possible in non-trivial cases. Tourwe, Brichau and Gybels (Tourwe *et al.*, 2003) call this the *aspect evolution paradox* because one would generally assume that improved modularity goes together with improved evolvability.

2.2. Expressiveness

Languages based on wildcard matching cannot express that certain wildcards must match the same value, which limits their applicability. As an illustration of the problem, consider the Eclipse framework. It is organized into public packages that provide stable APIs for a major Eclipse version and internal packages, whose implementation may change in every minor release. Each public package provides an API to a set of internal packages. The Eclipse *Internal Package Convention* states that an internal package may only be accessed from *its* public superpackage or from another internal subpackage of its public superpackage (EPNC01). For instance, ‘org.eclipse.platform’ is the public superpackage for all the packages named ‘org.eclipse.platform.internal.*’. Although this is a simple lexical convention, *AspectJ* cannot express it because it cannot express that different wildcards matching a package prefix must share the same value. This is illustrated in Figure 1.

```

1 aspect EclipsePackageAccessContract {
2   declare warning :
3     call(* **.internal.**.*(..))
4     && !( within(**.internal.**) || within(**) )
5     :
6     "The call violates the Eclipse package access contract.";
7 }

```

Figure 1 In order to check the Eclipse Internal Package Convention it would be necessary to express that the underlined wildcards must match the same value.

2.3. Conciseness

In order to implement a concern that has *heterogeneous* (that is, similar but not entirely equal) *effects* on different base code entities, aspect code that is similar up to the used base entity names must be written for every relevant base entity. This makes aspects highly redundant, which is inconvenient to write and a source of errors and maintenance problems.

We illustrate the conciseness problem of wildcard-matching-based aspect languages by considering ‘class posing’ as an example of a crosscutting concern that requires heterogeneous effects. A class posing statement Objective C (Pinston *et al.*, 1991) enforces consistent instantiation of a class C instead of a superclass S,

throughout a program. This is very useful for *unanticipated* object-oriented extension by inheritance³. A frequent scenario is the use of mock objects (Mackinnon *et al.*, 2002), a common technique for narrowing down the potential sources of a failure during testing. Its essence is the replacement of some of the tested classes by mock classes that provide a fixed, expected behavior. In a language that does not support class posing, the common way to use mocks of a class *S* is to implement a subclass of *S*, say *C*, that provides mock behavior and then replace all instantiations of *S* with instantiations of *C*⁴. This must be done for all the pairs of original classes and mock classes to be used in a test run. For each pair, it must be done consistently throughout the entire program, not just in the test classes – otherwise the inconsistent simultaneous use of mocks and original objects might itself be a source of errors.

An aspect implementing class posing must replace the creation of *S* instances by the creation of *C* instances throughout the program. This can be done in *AspectJ* by an around advice that replaces invocations⁵ of constructors of *S* by invocations of constructors of *C*. However, if *S* has multiple constructors, we need to repeat the same advice for every constructor, as illustrated in Figure 2. This leads to redundant code that is tedious to write, opens the door for editing errors, and hampers reuse (see Section 0).

```

aspect AJ_Replace_S_by_C {

    // Replace new S(String) → new C(String)
    S around(String arg1) :
        call(S.new(..)) && args(arg1) {
            return new C(arg1);
        }

    // Replace new S(String,int) → new C(String,int)
    S around(String arg1, int arg2) :
        call(S.new(..)) && args(arg1, arg2) {
            return new C(arg1, arg2);
        }

    // Continue like this for all other constructors of S...
}

```

Figure 2: *Heterogeneous aspect effects via redundant code: Replacement of constructor invocations by repeating largely similar advice.*

³ Note that programmers cannot foresee every possible case where ‘class posing’ might be useful in the future. Therefore one cannot assume that the entire program is written following some of the object creation patterns ().

⁴ In the related work section we discuss the advantages of this implementation scheme over an alternative one, known as “virtual mock objects”.

⁵ Replacing the *execution* of a constructor by an invocation of a subclass constructor would lead to infinite recursion. Therefore, we can only replace constructor *calls*.

2.4. Reusability

A further problem that stems from being dependent on concrete names of base entities is the lack of reusability of aspects. The aspect from Figure 2 is specific to the classes S and C. It does not implement the class posing concern, it just implements a specific instance of it. It is clearly not reusable, not even for another pair of classes, say X and Y, within the same application.

This problem is amplified when trying to write an aspect that is reusable across multiple applications. Then, reliance on base entity names is not possible, unless the aspect programmer knows all of the applications well enough to specify relevant entities by name (deVolder 2001), (Hassoun *et al.*, 2003), (Hanenberg *et al.*, 2003a), (Gybels *et al.*, 2003).

2.5. Safety

A more reusable implementation of class posing could be achieved in AspectJ by an around advice using runtime reflection, as illustrated in Figure 3. The advice intercepts all constructor invocations (line 6). For every intercepted invocation it checks whether the constructor belongs to the ‘old’ class and whether the ‘new’ class exists (line 7-21). If both are true, it accesses the dynamic join point context to get the arguments of the intercepted constructor call (line 24-28). Then the auxiliary `resolveConstructor` method determines via reflection the constructor with the right signature from the new class (line 29-30). Finally, it uses reflection to invoke the new constructor with the intercepted arguments (line 31).

The primary drawback of this solution compared to the first one is that it cannot guarantee anymore that errors (e.g. use of non-existing classes) are detected at weave-time. They will result in unforeseen `ClassNotFoundException`s at run-time. Apart from that, the solution is lengthy, hard to read and error-prone⁶. Last but not least, it is also inefficient, since it requires run-time reflection instead of exploiting static information at weave-time, as discussed next.

3. Problem Analysis: Statically Expressing Variation

The examples from Figure 2 and Figure 3 illustrate that writing reusable and non-redundant aspect code with *AspectJ* requires falling back on plain *Java* reflection, with all the implied drawbacks. Looking at the reflection-based implementation in Figure 3, it is noteworthy that all the reflective advice code, except line 31, is concerned only with determining at *run-time* information that is of entirely *static* nature (classes, argument lists, and method signatures). Obviously, wildcard-matching based languages are not expressive enough to capture all the static context of a join point in a weave-time usable abstraction.

The need to capture static context stems from the fact that the effect of an aspect needs to vary depending on this context, as illustrated by the class posing example.

```

1  aspect AJClassPosingWithReflection {
2
3      abstract String getOldClassName();
4      abstract String getNewClassName();
5
6      Object around() : call(*.new(..)) {
7          Class class = thisJoin point.
8              getSignature().getDeclaringType();
9          Class newClass;
10
11         // Is this the class to replace?
12         if (!class.getName().equals(getOldClassName())) {
13             return proceed();
14         }
15
16         // Does new class exist?
17         try {
18             newClass = Class.forName(getNewClassName());
19         } catch (Exception ex) {
20             return proceed();
21         }
22
23         // Try replacement
24         Object() args = thisJoin point.getArgs();
25         try {
26             Class() types =
27                 ((CodeSignature)thisJoin point.
28                     getSignature()).getParamtypes();
29             Constructor constr =
30                 resolveConstructor6(newClass, types);
31             return constr.newInstance(args);
32         } catch (Exception ex) {
33             throw new RuntimeException("AspectJ:" +
34                 "Instance creation failed");
35         }
36     }
37 }

```

Figure 3 *Heterogeneous effects using runtime reflection in an advice. Constructor invocations are replaced after lengthy, unsafe and inefficient run-time checking of purely static information.*

The important point here is that the choice of which particular variant of an effect applies at a particular join point should be done at weave-time by the aspect weaver, whenever the variation is completely determined by weave-time accessible information. This frees programmers from having to hand-code the different variants, in the same way as dynamic binding frees object-oriented programmers from having to manually encode message dispatch to different receiver types.

⁶ The reflective implementation of the method `resolveConstructor` is not shown here, since it consumes additional 30 lines. The complete code of the example can be found at <http://roots.iai.uni-bonn.de/research/logicaj/examples>.

Similar observations were made by (de Volder *et al.*, 2000) and (deVolder 2001). They argue that a good aspect-oriented language is one that provides mechanisms of genericity, polymorphism and parameterization that would allow code to be reusable and to adapt to different contexts.

4. Aspect Genericity

The above analysis leads us to the defining property of generic aspect languages:

An aspect language is generic if its aspects can specify multiple variants of an aspect effect in one single effect definition (without falling back on runtime reflection).

Instead of encoding the lengthy reflective advice from Figure 3 we want to be able to write a version of the repetitive code of Figure 2 such that the repeating parts occur only once, whereas the variant parts are replaced by *parameters* that capture the varying static context. This would mean that all the advice variants from Figure 2 are replaced by a single advice that is parametric with respect to the name, arity and parameter list of the intercepted constructor call. Depending on the parameter values, it would perform different effects at different join points. For instance, it would perform a $C(\text{arg1})$ constructor call at a join point that matches a $S(\text{arg1})$ invocation and a $C(\text{arg1}, \text{arg2})$ constructor call at a join point that matches a $S(\text{arg1}, \text{arg2})$ invocation (see Section 5.1.2, Figure 6).

In order to express the specific kind of parameterization needed for a generic aspect language we need *logic meta-variables*. They are defined as follows:

Meta-variable. A meta-variable is a variable that ranges over syntactic entities of the base language. An aspect language based on *Java* could provide, for instance, meta-variables that range over packages, types, classes, fields, methods, modifiers, statements, and expressions. Thus meta-variables are special just with respect to their domain. In the remainder we will denote meta-variables by identifiers starting with a question mark, e.g. $?C$.

Logic variable. A logic variable is a variable that can only be bound to values by the evaluation of predicates that take the variable as an argument. The values of logic variables cannot be manipulated in other ways (by assignments / side-effects). The tuple of variables in a predicate is bound consistently to *tuples* of values for which the predicate is true. For instance, assume that we have an aspect language that provides the predicate $\text{class}(?Cfq, ?CName)$ for expressing that the class whose fully qualified name is represented by $?Cfq$ has the short name represented by $?CName$. For a program that contains the classes pkg1.A , pkg1.B and pkg2.B the evaluation of a $\text{class}(?Cfq, ?CName)$ query will successively bind the logic meta-variables to the tuples $\langle \text{pkg1.A}, A \rangle$, $\langle \text{pkg1.B}, B \rangle$ and $\langle \text{pkg2.B}, B \rangle$ such that the first tuple value is the binding for $?Cfq$ and the second one for $?CName$.

Logic meta-variable. This is both, a logic variable and a meta-variable.

Based on logic meta-variables we can rephrase aspect genericity from a more technical point of view:

An aspect language is generic if it enables the use of logic meta-variables in aspect effect specifications.

This redefinition fulfils the requirement of the initial definition of aspect genericity because meta-variables contained in aspect effects let the effects vary depending on the meta-variables values. Such aspects provide one concise specification of many different effects. Meta-variables play the role of generic parameters of the aspect and every tuple of values for the meta-variables of an aspect represents one instantiation of the generic effects.

Whereas other ways to achieve genericity according to our first definition might exist, we claim that they are all instances of the generally applicable concept of logic meta-variables.

Note that logic meta-variables are not specific to systems based on logic programming but are commonly used, albeit in restricted forms, in object oriented and aspect oriented languages. For instance, generic type parameters in Java, Eiffel or C++ are just very limited instances of logic meta-variables: they range only over types and their values are provided manually by specifying constants at instantiation time. Aspect languages use predicates (‘pointcuts’, ‘filter conditions’, etc.) to bind values to wildcards, which are a restricted form of unnamed meta-variables⁷. The fact that the evaluation of a pointcut is side-effect free and yields all matching values is familiar to every aspect programmer. Filman and Friedman call the latter ‘quantification’ and regard it as a basic characteristic of aspect languages.

5. Design Space for Generic Aspect Languages

The technical definition of aspect genericity in Section 4 encompasses an entire family of aspect languages that can differ depending on the answers that their designers choose for each of the following questions:

1. How are values for meta-variables provided (*meta-variable binding*)?
2. What kind of meta-variables are supported (*meta-variable kind*)?
3. Where can meta-variables be used (*meta-variable scope*)?
4. Over which base entities can meta-variables range (*meta-variable domain*)?
5. What actions can be performed on the base language entities referred to by meta-variables (*effect granularity*)?

⁷ Wildcards are like anonymous variables because two occurrences are always treated as being different. De Volder (de Volder1999) points out that this leads to a set-oriented semantics of aspects and sows why a more powerful relational semantics is needed. Wildcards are additionally restricted because they only range over strings, not base language entities.

In this section we present these language design dimensions, explain their impact on the expressiveness of a language, and discuss where existing systems fit in these dimensions. Note that this is not meant to be an introduction to the different languages. Please see the referenced papers for details of the discussed languages.

5.1. *Meta-Variable Binding*

Regarding meta-variable binding there are two categories of generic aspect languages: ones that use external parameterization and ones that use predicate based binding.

5.1.1. *Externally supplied values*

The simplest way of providing values for meta-variables is to promote the meta-variables contained in the aspect effect to parameters of the aspect definition. Then the values can be supplied “from the outside”. This *external parameterization* is analogous to the way generic types are defined in object-oriented languages where type variables get values when the generic type definition is instantiated by a client.

Figure 4 shows an example illustrating the syntax of *SuperJ* (Sihman *et al.*, 2003), a generic aspect language based on external parameterization. The *SuperJ* preprocessor creates concrete *AspectJ* aspects from generic *SuperJ* aspects by replacing formal parameters by basic variables, locations, and classes. For this it uses a binding file, whose lines each contain a set of bindings between elements of a particular base class and parameters of a particular generic aspect. Similarly, the *Lancaster Frame Processor (LFP)* (Loughran *et al.*, 2004) provides external parameters for aspect constructs and uses a macro expansion approach that replaces parameter markers in *AspectJ* code by values provided in an external binding file.

External parameterization decouples the aspect code from concrete names of entities in the base program. Thus, aspects become more reusable, in principle.

However, base entity names must still be provided statically in a binding file. In order to provide the suitable bindings of meta-variables to base entities, the creator of the binding file must have intimate knowledge of the generic aspect *and* of the base classes to which the aspect is applied. This seriously limits the reusability of externally parameterized aspects in practice. In addition, the binding file might become very large since its size depends on the number of matches in the base program, which can grow combinatorially with the size of the base program. Manually editing large numbers of matches can be the source of omissions, typos and other input errors that will lead to hard to locate errors at run-time.

Figure 5 sketches the start of a binding file⁸ necessary to express class posing via external parameterization. Its purpose is to enumerate all combinations of values for

⁸ Since the original literature does not describe the binding file contents in detail, we invented our own ad-hoc syntax. Arguments are described by their name and type.

the required meta-variables (`?className`, `?subName` and `??args`). Compare this to the solution using predicate-based binding shown in Figure 6 and Figure 7.

```

aspect Constant_DPP_Heavy(EAT METHOD, ...) extends Common {
  ...
  void around(BOUND_CLASS C) : execution(EAT METHOD) ... {
    ...
    forks = 0;
    nAssigns++;
    logRecord();
  }
  ...
}

```

Figure 4 Excerpt of externally parameterized generic *SuperJ* aspect taken from (Sihman et al., 2003, Figure 6). Underlining highlights generic parameters whose actual values can be provided when the aspect is instantiated.

```

bindings_for(?className, ?subName, ??args) :
// All combinations for C and S:
( "C", "S", "arg1:String" ) ||
( "C", "S", "arg1:String,arg2:int" ) ||
( "C", "S", "arg1:String,arg2:int,arg3:T" ) ||
( "C", "S", "arg1:String,arg2:int,arg3:T,arg4:T4" ) ||
// All combinations for C1 and S1:
( "C1", "S1", "arg1:T1" ) ||
( "C1", "S1", "arg1:T1,arg2:T2,arg3:int" ) ||
( "C1", "S1", "arg1:T1,arg2:T2,arg3:int,arg4:T4,arg5:T5" ) ||
// All other combinations:
... ;

```

Figure 5 External parameterization: Start of listing of all tuples of relevant values for the meta-variables required to express the class posing concern for one particular base program (see Section 2.3 and 5.1.2 for comparison).

5.1.2. Predicate-based binding

A much more powerful alternative to external parameterization is provided by *predicate-based meta-variable binding*. The idea is to replace manual enumeration of statically known values by weave-time evaluation of predicates that bind meta-variables to values from the base program. In this approach the aspect predicates (or pointcuts) play a double role: they select join points where aspect effects are to be applied *and* they bind meta-variables to values from the static context of the selected join points.

We illustrate the power of predicate-based meta-variable binding by the generic implementation of the concept of *class posing* introduced in Section 2. The implementation in Figure 6 uses the syntax of *LogicAJ* (Roh, Windeln): Logic meta-

variables are denoted by names starting with a question mark, e.g. “?class”. Identifiers starting with a double question mark, e.g. “??args”, denote *list meta-variables*. They can match an arbitrary number of elements, e.g. any number of call arguments or method parameters (see Section 5.5).

The pointcut of the advice in Figure 6 (line 7-14) binds the meta-variables ??args and ?sub to values that conform to all the constraints checked in the pointcut. These values are passed to the advice body. Every pair of values produced by the pointcut evaluation instantiates the body in a different way. The instantiated body is then executed. Let us see how this works in the scenario from Figure 2, with the original superclass S class and its subclass C having each four constructors. If the replace pointcut returns the bindings ?class = S and ?sub = C, then the check in line 10 succeeds and the call pointcut in line 12 successively selects all join points where constructors of S are called. For each intercepted join point the args pointcut in line 14 binds ??args to the list of constructor argument values at that join point. Thus the advice body will be executed multiple times, with ?sub always bound to C and ??args bound differently each time. As a result, it will perform a C(arg1) constructor call at a join point that matches a S(arg1) invocation and a C(arg1,arg2) constructor call at a join point that matches a S(arg1,arg2) invocation.

```

1 abstract aspect ClassPosing {
2
3     // To be defined in a subaspect (See Figure 7):
4     // Who poses for whom? May return multiple pairs of values.
5     abstract pointcut replace(?class,/*by*/?sub);
6
7     Object around(?sub, ??args) :
8         // Determine replaced class and replacing class:
9         replace(?class, /*by*/?sub) &&
10        // Check if ?sub is a subclass of ?super:
11        subtype9(?sub,?class) &&
12        // Intercept ?super constructor invocations
13        call(?class.new(..)) &&
14        // Bind ??args to the argument list of the invocation:
15        args(??args)
16    { // Return instance of posing subclass ?sub (includes
17      // weave time check that the constructor exists):
18      return new ?sub(??args);
19    }
20 }

```

Figure 6: Implementation of class posing in LogicAJ. The aspect replaces each call of a constructor from ?class with a call to the respective constructor of the posing subclass, if the subclass exists.

⁹ The subtype predicate expresses the (reflexive, transitive) subtype relation. If called as above – with all meta-variables bound to values by a previous predicate – it will just check that the values are in a subtype relation. Otherwise it will successively bind the free variables to suitable pairs of values.

```

1  aspect MockObjectConcern extends ClassPosing {
2
3      // The mock replaces the original class.
4      pointcut replace(?class, /*by*/ ?mock):
5          // Pick a class:
6          class10(?class,   11, ?className, _) &&
7          // Its mock class must have the suffix "Mock":
8          concat12(?className, "Mock", ?mockName) &&
9          class(?mock,   , ?mockName, _) ;
10 }

```

Figure 7 Predicate-based meta-variable binding: An aspect concisely defining class posing for mock classes, assuming that every class to be tested has a single mock class that defines all its mock behavior.

Comparison. It is instructive to compare the reflective example from Figure 3 to the generic one from Figure 6. Most of the code in Figure 3 is concerned with determining whether the intercepted join point was relevant at all, determining its arguments, finding the constructor with the right signature from the new class, and invoking it with the intercepted arguments. In the variant from Figure 6 only the *relevant* join points are intercepted, their specific context information is captured in meta-variables and used at the appropriate places of the aspect effect. The generic aspect from Figure 6 has the same effect as the redundant code from Figure 2 and the reflective one from Figure 3. However, it is

- precise (only the relevant join points are processed),
- efficient (no run-time weaving is required),
- safe (it provides weave-time type checking),
- concise (just 7 lines of non-redundant code), and
- reusable (applying it to other classes or entirely different base programs requires just another implementation of the `replace` pointcut in a concrete subaspect).

The last two benefits don't apply for externally parameterized aspects because of the need to provide a manually generated binding file for every base program to which the aspect is applied. For instance, the abstract `replace` pointcut in Figure 6 subsumes an explicit parameterization of the entire aspect by the meta-variables `?className`, `?subName` and `??args`. With predicate-based binding there is no need for an explicit enumeration of all possible bindings because these can be expressed concisely by stating the properties of base entities to be bound to the meta-variables. Figure 7 shows a concrete subaspect of the class posing aspect from Figure 6. Its definition of `replace` selects all the classes that have mock subclasses in *any* base

¹⁰ This use of the class predicate says that `?class` is a class with simple name `?className`.

¹¹ Underscores are anonymous meta-variables.

¹² The third argument of `concat` is the concatenation of the first and second one.

program (Roh *et al.*, 2004). See for comparison the binding file in Figure 5 enumerating all the values for `?className`, `?subName`, and `??args` for *one* particular base program.

5.2. Meta-Variable Scope

Various proposals in the context of aspect-oriented logic meta-programming (AOLMP) have shown that some of the dependencies of aspects on base programs addressed in Section 2 can be alleviated by the use of a logic-based pointcut language, e.g. (deVolder 1998), (Gybels 2001), (Gybels *et al.*, 2003). However, most AOLMP research focuses exclusively on logic-based pointcut languages limiting the scope of meta-variables to aspect predicates. Such languages enable powerful analyses but do not achieve genericity. The definition of generic aspect languages requires meta-variables to be used in aspect *effects* in order to enable effect variability. It is up to aspect language designers to choose which of the aspect effects can use meta-variables and decide whether the language additionally supports meta-variables in aspect predicates. In this section we discuss the available options and their implications.

Languages without meta-variables in aspect predicates are limited to using external parameterization (5.1.1). Use of meta-variables in predicates enables using the more powerful predicate-based binding (see 5.1.2). Obviously, the most powerful design option is to let the scope of meta-variables be an aspect effect and its associated pointcut. However, some languages do not take advantage of this option. For instance, *SuperJ* (Sihman *et al.*, 2003) and *LFP* (Loughran *et al.*, 2004) only support external parameterization, although meta-variables may be contained in their predicates.

Other languages support meta-variables in predicates but limit the use of meta-variables to a subset of the expressible aspect effects, supporting either *generic advice* (meta-variables in pointcuts and advice) or *generic introductions* (meta-variables in pointcuts and introductions). For instance, Andrew¹³ (Gybels 2001) supports only generic advice, whereas the initial design of *Sally* (Hanenberg *et al.*, 2003a) supported only generic introductions. These restrictions limit the expressiveness of a language, since most non-trivial applications require the joined power of introductions and advice. For instance, (Kniesel *et al.*, 2004) analyze design patterns as a typical use case for generic aspect languages. They show that a language without generic advice cannot (1) express instantiation of the decorator pattern for multiple Component classes, (2) enforce that all clients use the decorator instead of the original component, and (3) implement object based inheritance. A language without generic introductions cannot create decorators at all, since it cannot fill them with the required forwarding methods.

¹³ Andrew has meanwhile been renamed to "Carma", see prog.vub.ac.be/~kgybels/Research/.

		Meta-variable values used in ...		
		Introductions only	Advice only	All effects
Meta-variable values bound by	External Parameters	Externally parameterized introductions: —	Externally parameterized advice: —	Uniform external parameterization: (Sihman <i>et al.</i> , 2003), (Loughran <i>et al.</i> , 2004)
	Pointcuts	Generic Introductions: (Hanenberg <i>et al.</i> , 2003a)	Generic Advice: (Gybels 2001), (Silaghi <i>et al.</i> , 2003)	Uniform Genericity: (Roh <i>et al.</i> , 2004), (Windeln 2003), (Hanenberg <i>et al.</i> , 2003a) ¹⁴

Table 1 Design subspace for generic aspect languages defined by the options for meta-variable binding and meta-variable scope. The table shows where known languages fit within this space. The symbol “—” indicates an empty category.

Uniform Genericity. Combining the above insights that meta-variables should be usable in all aspect effects and that their scope should be an effect *and* its associated pointcut leads to the call for uniform genericity. A language provides *uniform genericity* if it allows use of meta-variables in *all* its aspect effect specifications *and all* its aspect predicates. Currently, we know of only two uniformly generic aspect languages: *LogicAJ* (Roh *et al.*, 2004, Windeln 2003) and *Sally* (Hanenberg *et al.*, 2003a)¹⁴. They unify the concept of introduction and advice letting both be guarded by a prior pointcut definition that enables gathering of context-dependent information in meta-variables. Uniform genericity is clearly the most powerful option in the design space discussed so far and summarized in Table 1.

5.3. Meta-Variable Domain

Generic aspect languages can differ significantly by the domain of meta-variables. For instance, (Silaghi *et al.*, 2003) propose a very limited form of generic advice for *AspectJ*, where meta-variables can only range over types. Most other approaches support meta-variables ranging over types, fields, and methods. *LogicAJ* additionally provides meta-variables ranging over code blocks but currently only supports their use for matching entire method bodies.

Fine-grained genericity. Different researchers have pointed out that certain cross-cutting concerns, e.g. code-coverage or program optimization (Harbulot *et al.*, 2003), can only be modularized if aspects are able to address base language entities at the finest level of granularity: individual statements and expressions. Accordingly, we say that a generic aspect language supports *fine-grained genericity* if its

¹⁴ The version of *Sally* described in (Hanenberg, *et al.* 2003) only supports generic introductions. However, as of its Nov. 2004 release, *Sally* also provides uniform genericity – see <http://dawis.informatik.uni-essen.de/site/site/research/aosd/sally/>.

meta-variables can range over *all* syntactic entities of the respective base language. In Java, this would cover the entire spectrum from packages, types, methods and fields, down to annotations, statements and expressions. Currently, fine grained genericity is not achieved by any known generic aspect language.

5.4. *Effect Granularity*

Within a class of languages that have the same meta-variable domains it is interesting to compare whether all possible effects are indeed applicable to all supported domains. *LogicAJ* currently seems to be the only language that does not restrict the effect granularity in any way. In particular, it supports generic type introductions – that is the generic creation of a set of types, whose names are not statically fixed in the aspect code (Roh *et al.*, 2004). Generic type introductions are very useful in many scenarios that require generative capabilities. An example is the generic creation of abstract decorator classes that are specific for a particular type of decorated objects (Kniesel *et al.*, 2004, Figure 5).

5.5. *Meta-Variable Kind*

The example from Figure 6 also illustrates the distinction between two kinds of meta-variables: singular meta-variables and list meta-variables.

Singular meta-variables match only one base language element at a time, e.g. a type, field, etc. Singular meta-variables are the minimum that any generic aspect language must provide. With singular meta-variables only, we are still forced to write sometimes redundant code that differs only in the length or depth of a nested structure, e.g. in the number of arguments of a method. This is clearly undesirable and limits the expressive power, since it is practically unfeasible to enumerate an infinite number of arbitrary length structures.

Therefore, *LogicAJ* (Roh *et al.*, 2004), (Windeln 2003) and *Josh* (Chiba *et al.*, 2004) additionally provide *list meta-variables* that match a sequence of elements from the base language at a time. *Josh* only supports the \$\$ list meta-variable that can be used in a proceed call to match an arbitrary list of arguments. *LogicAJ* supports list meta-variables wherever structures of statically unknown length or depth must be matched in aspect predicates or generated in aspect effects. Examples are argument and parameter lists, the elements in a *Java* package name, or the statements in a block. The concise implementation of class posing from Figure 6 is not possible without the list meta-variable `??args`.

Some languages, for instance *Sally*, only support one kind of meta-variables but allow them to match indiscriminately singular values and list values. This does not allow to distinguish conditions where exactly one value is required (`?x`) from those where a (possibly empty) sequence of values is required (`??x`). Doing so requires additional predicates for checking list structure, thus bloating the pointcut language and the expressions programmers need to write.

6. Challenges for Generic Aspect Languages

Within the discussed design dimensions we have identified some blind spots that have not yet been filled by existing aspect languages. For instance, it is still unclear what would be a good design for a language that integrates uniform and fine-grained genericity. In addition, it is to be explored how well current weaving technologies would perform if they were to evaluate predicates and perform effects at a very fine granularity. Filling each of these gaps is certainly a challenge. However, in this section we focus on challenges that are orthogonal to the introduced classification.

Static analysis. Generic aspect languages improve over the use of run-time reflection, among others by enabling weave-time type checking. However, *modular* static analysis of aspect code, which is largely unsolved even for non-generic aspects, is even more difficult in the presence of meta-variables. For instance, an expression like `?metaVar.m()` cannot be type checked in the same way as `var.m()` because we know nothing about the value of the meta-variable – not even its syntactic nature. It could be a class, in which case the expression is the invocation of a class method or it could be a field, in which case the expression is the invocation of an instance method in that field’s type.

Syntactic typing. In order to prevent, for instance, substitution of classes where fields are expected and vice-versa, meta-variables need to be *syntactically typed*, that is every meta-variable needs to have a type that determines the kind of syntactic entity from the base language that may be substituted. Syntactic types can either be declared or inferred from the definition of the predicates that are used to bind the respective meta-variable values. Syntactic typing enables modular, pre-weave-time checking of whether values substituted for meta-variables in aspect effects will always produce legal base language programs.

Semantic typing. In addition, we need a second level of typing, in order to determine also the base language type that a meta-variable will take on. We call this the *semantic type* of the meta-variable, because it refers to the type notion defined in the static semantics of the base language. For instance, for our above invocation of `m()` we need to know whether `?metaVar` is guaranteed to be bound only to base language elements whose base language type contains the method `m()`.

Interference detection. The added expressive power of generic aspect languages does not come for free. Last but not least, as long as we know neither the semantic nor the syntactic type of a meta-variable, let alone the precise entity to which it will be bound at weave-time, *aspect interference* checking becomes much harder than it already is in non-generic aspect languages. Performing modular interference analysis for generic aspects is a challenge that has not yet been addressed by any published paper that we know of.

¹⁵ See the AspectJ reference for the definition of `declare warning`, `call` and `within`.

8. Conclusions

In this paper we have reviewed the needs, design options and technical challenges related to generic aspect languages. We started from an analysis of the limitations of non-generic aspect languages regarding, evolvability, expressiveness, conciseness, reusability and type safety. The common cause of the identified problems is the reliance of aspects on concrete names of base program entities and wildcards, which prevents statically expressing context-dependent variation of aspect effects. We defined generic languages to be those able to express the desired variation and identified the ability to use meta-variables in aspect effects as their common technical essence. Our review of existing or proposed generic languages revealed five basic questions that language designers must consider when extending their aspect language towards genericity:

- What kind of meta-variables are supported (*meta-variable kind*)?
- How are the values of meta-variables provided (*meta-variable binding*)?
- Where can meta-variables be used (*meta-variable scope*)?
- Over which base entities can meta-variables range (*meta-variable domain*)?
- What actions can be performed on the base language entities referred to by meta-variables (*effect granularity*)?

With respect to the first two design dimensions we were able to identify *list meta-variables*, *predicate-based binding* and *uniform genericity* as the options of choice, whose power is already demonstrated in existing languages (*LogicAJ* and *Sally*). Reaching beyond the current state of the art we have discussed two main challenges for future generic aspect languages: the combination of uniform and *fine grained genericity* and the development of *modular static analysis* techniques for generic aspects.

9. References

- Aksit M., Bergmans L., Vural S.: “An Object-Oriented Language-Database Integration Model: The *Composition Filters* Approach”, *ECOOP 1992 proceedings*, Springer-Verlag, 1992.
- Alvarez J., “Parametric Aspects: A Proposal”. Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, March 2004.
- Chiba S., “Load-time structural reflection in Java”, in *ECOOP 2000 proceedings, LNCS 1850*, pp. 313–336, Springer-Verlag, 2000.
- Chiba S., Nakagawa K.: “Josh: An Open AspectJ-like Language”, in *AOSD 2004 Conference Proceedings*, pp. 102-111, ACM Press, 2004.
- Compose* Homepage: <http://composestar.sf.net/>
- Demeter/Java, <http://www.ccs.neu.edu/research/demeter/releases>
- de Volder K.: Type-Oriented Logical Meta Programming, *PhD thesis, Vrije Universiteit Brussel*, Belgium, 1998.

