Parametric Introductions

Stefan Hanenberg and Rainer Unland
Institute for Computer Science
University of Essen, 45117 Essen, Germany
{shanenbe, unlandR}@cs.uni-essen.de

ABSTRACT

Aspect-oriented software development allows the programmer to identify and treat separately concerns that, subsequently, can be woven to different target applications. For this, aspect-oriented languages like AspectJ and Hyper/J provide mechanisms for defining and composing such crosscutting concerns. An introduction is a mechanism for defining certain static crosscutting concerns, i.e., concerns that affect the type of the application they are woven to. This paper discusses the implementations of introductions in AspectJ and Hyper/J and reveals their limitations by presenting typical examples of static crosscutting code that cannot be handled adequately by them. To solve these deficiencies we will present the concept of parametric introduction, which are introductions that rely on parameters that are evaluated during weave-time.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Aspect-Oriented Programming, Weaving, Crosscutting, Design Patterns

1. MOTIVATION

Aspect-oriented software development [16] deals with the modularization of concerns that cannot be encapsulated by traditional composition techniques. Without modularization such concerns would be spread over numerous modules. Hence, such concerns are called *crosscutting concerns* and the code belonging to them *crosscutting code*. There are two kinds of crosscutting code: *static* and *dynamic*. Static crosscutting code is given if a static analysis of the program definitively reveals whether such code is supposed to occur. With dynamic crosscutting code such an occurrence depends on runtime-specific elements.

Aspect-oriented programming languages like AspectJ [1] or Hyper/J [15], which extend object-oriented programming languages, provide new composition techniques in addition to the existing ones. A core tool of aspect-oriented languages is the *weaver*: it takes "self-contained" concerns and weaves them into applications. This allows programmers to treat such (crosscutting) code in separate modules and to only afterwards accomplish the crosscutting effect. Both AspectJ and Hyper/J support *static weaving*, which means that weaving cannot be performed at run-time. A static weaver analyses the base system to determine where to insert the crosscutting code. Static crosscutting code can directly be inserted into the system, while for dynamic crosscutting the weaver lays down additional conditions that determine during runtime if the woven code is to be executed.

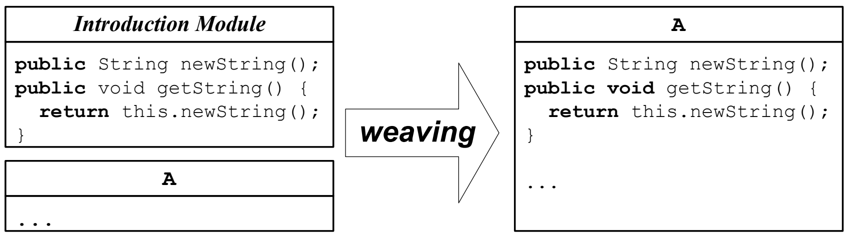
AspectJ and Hyper/J support the concept of *introduction* to define new members for existing types outside the original class or interface definition. Since both are based on a strongly typed language the question whether a class has a certain member has to be answered at compile time. Hence, woven member definitions implement a certain kind of static crosscutting code.

AspectJ and Hyper/J claim to solve the problem of crosscutting concerns. Thus, they claim to solve the problem that static crosscutting code coming from a single concern has to be implemented in different modules. However, we will present general examples of crosscutting code that cannot be modularized using these implementations of introductions. Hence, a more advanced implementation is needed.

This paper is structured as follows: In section 2 introductions are defined together with their implementation in AspectJ and Hyper/J. Section 3 presents some typical examples of static crosscutting code that cannot be modularized by AspectJ or Hyper/J. In section 4 we propose parametric introduction as a new language concept and present its implementation in our general-purpose aspect language Sally. Moreover, we show that these kinds of introductions solve the deficiencies revealed by the previous examples. In section 5 we compare parametric introductions to other relevant concepts. Finally, section 6 concludes the paper.

2. ASPECT-ORIENTED INTRODUCTIONS

The term *introduction* was originally introduced in AspectJ. It implements a mechanism for adding fields, methods and interfaces to classes [1]. This is similar to *open classes* as described in [3] and [4]. Introductions were motivated by the observation that different concerns have a direct impact on the type structure of object-oriented applications. As a consequence modularization is compromised since some elements in the type structure, like certain fields and methods, come from different concerns. Aspect-oriented techniques permit to remove these elements from the type definition and provide a mechanism to introduce them at weave time. An introduction is a strictly type increasing operation on types, since it adds new features to types, but does not permit to remove anything.



```
public String newString();
public void getString() {
  return this.newString();
}
```

```
...
```

```
public String newString();
public void getString() {
  return this.newString();
}

...
```

Figure 1. Aspect-Oriented Introductions

Figure 1 illustrates an aspect-oriented introduction. A special introduction module defines new members that are to be introduced into a target class `A`. The weaver takes the introduction module and the target class and weaves them together. Thus, all elements of the introduction module become members of the target class `A`.

A key question of an introduction mechanism is how to handle the self-reference `this` in the introduction module. This handling has a direct impact on the type correctness of the code that is supposed to be introduced. If `this` is bound at *weave-time*, that means when the introduced members become part of a target class, it is hardly possible to determine the correctness of *abstract introduction*. Abstract introductions are introductions that can be reused in different contexts. That means the target classes are unknown when the introduction is defined[1]. If inside an abstract introduction `this` is bound at weave-time and the introduced code sends some messages to `this` it is not possible to determine if the (unknown) target type provides an appropriate method.

In some approaches `this` is at first not bound to any type, however, is bound during the weaving process. Other approaches bind `this` already before weave-time. The binding of `this` at weave-time permits a flexible combination of introductions since an introduced method may use further introduced members. In case `this` is already bound before weave time the corresponding type can be used for type checking of the introduced code.

2.1 Introductions in AspectJ

In AspectJ introductions are declared in the class-like construct *aspect*. It syntactically consists of the member definition and the name of the target type. To add new interfaces to a target type, AspectJ provides the keywords `declare parents`.

```
class A {  ... }
interface NewInterface {...}
aspect MemberIntroduction {
  public String A.newString;
  public void A.doSomething(){...}
}
aspect InterfaceIntroduction {
  declare parents: A implements NewInterface;
}
aspect TypePatternIntroduction {
  public void (A+).doSomething2() {...}
}
```

Figure 2. Introductions in AspectJ

Figure 2 contains an aspect `MemberIntroduction` that adds a field `newString` and a method `doSomething` to class `A`. The aspect `InterfaceIntroduction` adds the interface to class `A`. The target type can be specified using so-called type patterns that permit to apply an introduction to several types at the same time. For this AspectJ provides some operators for specifying sets of target types. For example, a type pattern `(A+)` lays down that every subclass of `A` is meant to be the target class for the introduction. The aspect `TypePatternIntroduction` specifies a new method `doSomething2` for every subclass of `A`.

An often used idiom in AspectJ is the *container connection* (cf. [9]): the application of introductions to an interface that is later introduced to a class. In such a case AspectJ applies the introduction to all classes implementing the interface. Figure 3 illustrates such an introduction. An aspect `IntroductionLoader` introduces a new field `newString` to an interface `Container`. A different aspect introduces this interface to a target class. The result after weaving is that `TargetClass` contains the introduced field `newString` and the introduced interface `Container`.

The main purpose of container introductions is to apply a collection of introductions to several different target classes not know at introduction definition time. The container is then introduced to all target classes without the need to perform any destructive changes in the introduction definition. On the one hand this reduces the effort of performing introductions, since it only needs one `declare parents` statement. On the other hand introductions can be applied without knowing in detail all elements which are part of the container.

In the normal application of introductions AspectJ binds `this` at weave-time. Thus, at introduction definition time `this` does not refer to any type. This approach works without problems as long as all target classes of an introduction exist, i.e., all possible classes matching the type pattern are known when the introduction is defined. For example, in figure 2 this is true for the aspect `MemberIntroduction` since the only class that matches the type patter is a class named `A`. During weaving, the weaver binds `this` to the target class `A` and checks, if all occurrences of `this` match `A`. If this is not true, the weaver throws an exception. However, if not every possible classes exist the introduced code may contain type errors. For example, a type pattern `(A*)` refers to all classes whose name begins with an `A`. In such a case AspectJ cannot determine if the use of `this` inside the introduced code is type correct.

In container introductions `this` is bound to the container type and not to the target class (which is somehow misleading, since the introduction is performed on the target class and not on an interface).

[1] In aspect-oriented programming such abstract introductions are often used to define aspect libraries that are provided by a third party and adapted by the application programmer.

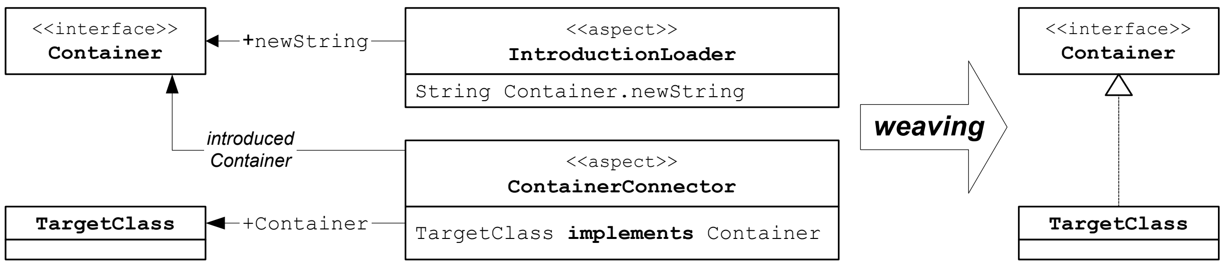


Figure 3: Container introduction in AspectJ

Hence, AspectJ realizes the binding of `this` in two ways: either at weaving time (in the usual introduction application) or at introduction definition time (container connections).

2.2 Introductions in Hyper/J

In contrast to AspectJ Hyper/J is a tool that provides different composition mechanisms for Java classes but does not extend the underlying programming language. The composition mechanisms originate from the theoretical background of *subject-oriented programming* (cf. [14]). However, there is a common agreement to regard Hyper/J as an aspect-oriented technique (cf. e.g. [22]). The terminology used in Hyper/J differs substantially from the one used in AspectJ. In the following we discuss those mechanisms of Hyper/J that are almost equivalent to introductions in AspectJ.

In Hyper/J introductions are realized by defining classes that contain the members to be introduced and a so-called *hypermodule* that defines how to weave the participating classes[2]. The hypermodule specification is a structured file that is used as an input parameter by the weaver. To describe how to weave classes Hyper/J provides different *integration relationships* like *merging* or *overriding* (cf. [21]).

```
class A {...}
class MemberIntroduction {
  public String newString;
  public void doSomething() {...}
}
// hypermodule specification
relationships:
  ...
  compose class MemberIntroduction with
            additionally class A;
...
```

Figure 4: Introductions in Hyper/J

Figure 4 shows schematically how an introduction in Hyper/J looks like and, furthermore, lists an extract from the corresponding hypermodule specification file: the class `MemberIntroduction` contains the members to be introduced and is defined in an ordinary class. The corresponding hypermodule lays down that the class `MemberIntroduction` is to be composed with class `A`. In fact the weaver introduces the class `MemberIntroduction` as a new super-class of `A` so that class `A` has all methods defined in the introduced super-class. The use of the composition rule `compose` permits the introduction of members of one class into several different target classes. However, it does not really "introduce" the members, since members of the introduced class do not physically become members of the target class. To do so, Hyper/J provides an `equate` relationship. It permits two classes to be woven together to form one single class. However, the disadvantage of this relationship is that it cannot be used to introduce members into more than one target class.

Since Hyper/J weaves Java classes, `this` is bound to the corresponding class in all members that are to be introduced. Introduced members are regular Java members and can, therefore, already be accessed from other classes before weaving. If `equate` is used Hyper/J *forwards* all calls to the introduced members to the woven class, that means all calls to the class to be introduced are transformed into calls to the woven class. Due to this forwarding mechanism, Hyper/J does not permit to introduce members to more than one class. If it were possible, it would be ambiguous to what woven class a call has to be forwarded.

3. WHERE CURRENT IMPLEMENTATIONS FAIL

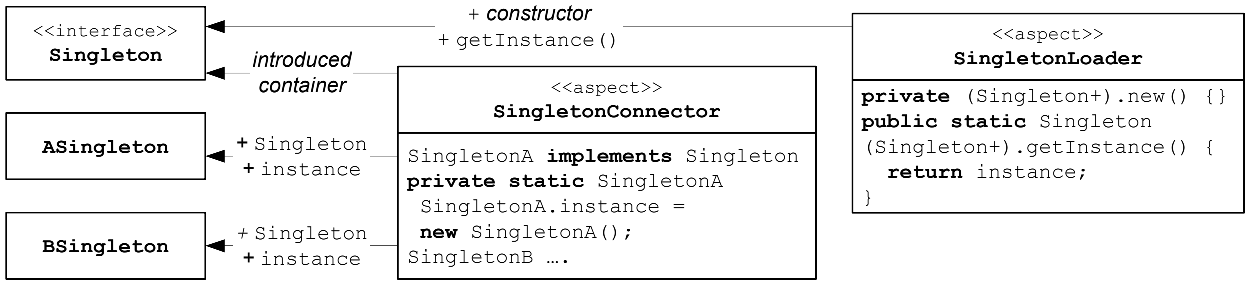
This section presents some typical examples of static crosscutting code that cannot completely be modularized by using introduction in AspectJ and/or Hyper/J. We chose some often used implementations of well-known GoF-design patterns [5] for two reasons. First, in aspect-oriented programming some of those implementations are usually regarded as aspects that are meant to be modularized and reused in different situations [13]. Second, the used implementations of GoF design patterns are commonly known. Hence, it is not necessary to motivate the implementation or to explain in what context they usually occur.

3.1 Singleton implementations

A straightforward implementation of the singleton design pattern [5] in Java is to add some members to the class that is supposed to become a singleton: a (private) static field of the same type as its class, a private constructor and a public static method that returns the singleton instance[3]. That means, every class that is supposed to become a singleton contains all these members. If the singleton pattern is applied to more than one class these members represent static crosscutting code. The code comes from the same concern ("make a class a singleton") that changes the class structure of a system, but differs slightly from class to class.

[2] Due to space limitations we skip some details here. For a more comprehensive description see [15].

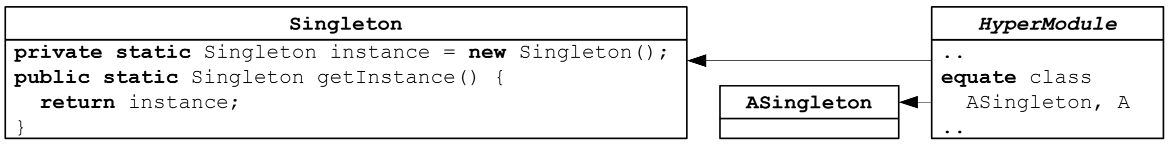
[3] We avoid discussing topics like garbage collection in the implementation of the singleton pattern. For a more comprehensive discussion we refer e.g. to [7], pp. 127-133.



```
SingletonA implements Singleton
private static SingletonA
 SingletonA.instance =
 new SingletonA();
SingletonB ….
```

```
private (Singleton+).new() {}
public static Singleton
(Singleton+).getInstance() {
  return instance;
}
```

Figure 6: Singleton Implementation in AspectJ



```
private static Singleton instance = new Singleton();
public static Singleton getInstance() {
  return instance;
}
```

```
..
equate class
  ASingleton, A
..
```

Figure 7: Singleton implementation in Hyper/J (only one target)

For example, in figure 5 two classes implement a singleton. Both have the singleton specific members whose types differ.

In AspectJ the singleton can be implemented by introducing all singleton specific members to an interface `Singleton` that represents the container in a container connection (see figure 6). This container is afterwards connected to a target class via an introduction. AspectJ does not permit to define the object creation within `SingletonLoader`, because there is no possibility to refer to the class, to which `Singleton` will be introduced[4]. Instead, this has to be defined in the connector. Hence, AspectJ does not permit to define the singleton aspect in one single module, because instance creation is still scattered over different connecting aspects. Furthermore, the return type used in `getInstance` does not correspond to the type of the target classes (`ASingleton`, `BSingleton`). As a consequence typing information gets lost during weaving and every object requesting the singleton instance has to perform a type cast.

```
class ASingleton {
  private ASingleton () {}
  private static ASingleton
    instance = new ASingleton ();
  public static ASingleton getInstance() {
    return instance;
  }
  ...
}
class BSingleton {
  private static BSingleton
    instance = new BSingleton ();
  private BSingleton() {}
  public static BSingleton getInstance() {
    return instance;
  }
  ...
}
```

Figure 5. Singleton implementations in Java

[4] It should be mentioned that it is possible to implement the creation by using so-called advice and introspection. However, this implementation has disadvantages like additional type casts whose discussion is outside the scope of this paper.

Figure 7 illustrates an implementation of the singleton in Hyper/J by using the `equate` relationship. The result is that the type of the class variable and the return type of `getInstance` correspond to the target class since class `Singleton` and `ASingleton` and, therefore, all occurrences of `Singleton` in the woven code are replaced by `ASingleton`. Thus, the loss of type information that occurred in AspectJ does not happen in Hyper/J. However, if an `equate` relationship is chosen, it is not possible to combine the singleton class with multiple different target classes because of the forwarding mechanism. A work-around in such case is to make as many copies of the singleton class as singletons appear in the application and then perform an equate relationship. But this means to give up the separation of concerns principle, since the singleton-aspect would be spread over numerous different classes that contain all the same implementation. If a singleton class containing all singleton specific members were integrated using the compose relationship, the singleton would become a super-class of both target classes. Hence, both target classes would share the same static members.

```
interface VisitorElement {
  public void accept(Visitor v);
}
class A implements VisitorElement {
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}
class B implements VisitorElement {...}
class C implements VisitorElement {...}
interface Visitor {
    void visit(A node);
    void visit(B node);
    void visit(C node);
}
class ConcreteVisitor implements Visitor {
    void visit(A node) {.....}
    void visit(B node) {.....}
    void visit(C node) {.....}
}
```

Figure 8: Visitor implementation in Java

It should be noted here that another possibility in Hyper/J is to define new classes ASingleton and BSingleton in different packages than the target classes and apply a *merge* relationship. Although this approach is technically possible, it does not modularize the crosscutting code in any way.

3.2 Visitor implementations

A visitor [5] encapsulates polymorphic behavior outside of the class hierarchy and is used to implement operations on complex structures. Let us assume that an object structure of classes A, B, C and D is given. A visitor implements an interface VisitedElement consisting only of the declaration of the double dispatch method accept(Visitor). The interface Visitor declares a method visit for each element to be visited. The operation that is to be performed on the object structure is specified in each implementation of the visitor interface.

In the implementation there are different kinds of crosscutting code. First, classes whose objects are to be visited contain the double dispatch method. Hence, this method represents static crosscutting code. Furthermore, the interface Visitor contains the method visit for all classes implementing VisitorElement.

```
interface VisitedElement {}
interface Visitor {
  visit(A node);
  visit(B node);
  visit(C node);
}
class ConcreteVisitor implements Visitor{
  ...
}
aspect VisitedElementLoader {
  public void VisitedElement+.accept(Visitor v)
  {v.visit(this);}
}
aspect VisitedElementConnector {
  declare parents: A implements VisitedElement;
  declare parents: B implements VisitedElement;
  declare parents: C implements VisitedElement;
}
```

Figure 9: Visitor implementation in AspectJ

AspectJ permits to modularize the double dispatch method by introducing it to the interface VisitorElement. That means, every class implementing this interface will automatically possess such a method. On the other hand, AspectJ does not permit to handle the different visit methods in the visitor interface. So the interface has to be adapted by hand. As a consequence a new method has to be defined in VisitorElement for every class that is added to the object structure.

Hyper/J does not permit to modularize the visitor implementation in any way. The compose relationship cannot be used since the double dispatch method requires the dispatch method to be physically present in the target class and not only inherited. An equate relationship needs to create a new class containing the dispatch method for every visited class. Hence, this does not modularize the crosscutting code.

3.3 Decorator implementations

A decorator [5] is used to extend the functionality of single objects during run-time. The typical implementation of a decorator in Java for a given class A is shown in figure 10. An interface is extracted from A and an abstract decorator implements that interface. The abstract decorator has an instance variable of the implemented interface to which all incoming messages are forwarded to. Concrete decorators extend the abstract decorator and override the methods they need to adapt.

If more than one class is decorated in an application (which is the usual case) all occurring decorators in Java look like this. Hence, the static crosscutting code consists of the following elements: the extracted interface of the decorated class and a class that implements the interface and forwards all messages. The concrete decorators are application specific and, therefore, usually differ from application to application. Hence, concrete decorators are not part of the static crosscutting code.

The problem with decorator implementations is that, although they contain a lot of static crosscutting code, they do not contain any fixed implementation. The interface Component consists of the public methods of the decorated class. Moreover, the type of the component within the abstract decorator depends on the class to be decorated. Although it is known that an abstract decorator is supposed to only forward all incoming messages to the component the corresponding signatures are not known since any arbitrary class can be decorated.

```
class A implements Component {
  public void doSomething() {...}
}
interface Component {
  void doSomething();
}
class AbstractDecorator implements Component {
  public Component component;
  public void doSomething() {
    component.doSomething();
  }
}
class ConcrDecorator extends AbstractDecorator
{
  ....
}
```

Figure 10: Decorator implementation in Java

In AspectJ it is not possible to define a decorator aspect independent of the classes to which it is applied to. A mechanism is needed that adds the public methods of a class to an interface and adds default implementations to a class that implements this interface. Since AspectJ does not permit to bind method names nor types at weave time, the introduction implementation is not sufficient to perform such a task[5].

The argumentation why Hyper/J does not permit to modularize a decorator implementation is likewise. Hyper/J cannot extract an interface out of a class and introduce a default implementation of methods with unknown signatures.

[5] It should be noted here that the mechanism for dynamic crosscutting offered by AspectJ permits to decorate classes in a different way by using the reflection API for dynamic crosscutting. However, this approach is highly complex and more a workaround than an acceptable solution.

3.4 Summary so far

The previous examples illustrated typical occurrences of static crosscutting code in object-oriented programs. The examples have in common that the static crosscutting code varies every time it occurs (cf. [11] for a further discussion). In the singleton example the return types vary, in the visitor example the parameter types of the visit methods vary and the number of introductions depends on the number of visited classes. In the decorator example method signatures vary. However, we argued why occurrences of such variations are still part of static crosscutting code and, thus, are to be modularized using aspect-oriented techniques.

We showed that neither the introduction implementation in AspectJ nor the implementation in Hyper/J are sufficient to separate the static crosscutting code in single modules. Instead, both implementations force the developer to spread code pieces over different modules. Thus, neither AspectJ nor Hyper/J follow the principle of separation of concerns. Nevertheless, there are numerous situations where both implementations satisfy the needs at hand. Hence, an extension of the current implementation mechanisms is needed that permits to perform introductions in the known way and furthermore solves the mentioned problems.

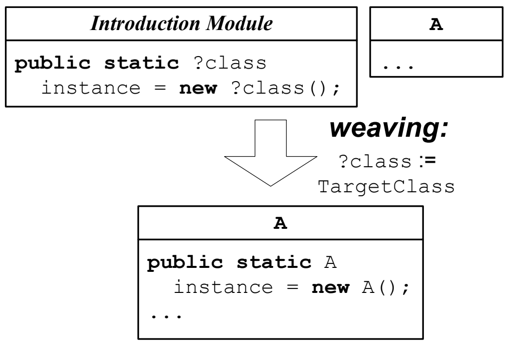


Figure 11: Parametric introductions

4. PARAMETRIC INTRODUCTIONS

Parametric introductions receive parameters during weave-time. The main motivation for this idea is the observation that sometimes static crosscutting code varies when woven to different types (cf. [11]). The code that is to be introduced contains variables that are bound by the weaver. Figure 11 illustrates an example of a parametric introduction that is motivated by the singleton implementation from 3.1. The introduction specifies a field `instance` to be introduced to a target class. The type of the field is used as a variable within the introduction. Furthermore, the variable is initialized with a new object whose class is not determined. The only parameter used within that introduction has the identifier `?class`. This introduction specifies that, whenever the field is introduced to a class, the type of the field and the type of the class to which the `new` operator is applied to are the same. In figure 11 the weaver assigns the target class to the parameter `?class`.

Parametric introductions require a mechanism to assign values to its parameters in order to enable the weaver to determine the actual woven code. Moreover, sometimes it is useful to be able to specify introductions without specifying the target classes. This permits to define libraries of introductions that can be adapted to target classes during application development without the need to perform destructive modifications in the introduction module. Furthermore, in section 3.2 we showed that it may be necessary to apply an introduction to target classes more than once with different parameter value pairs.

Parametric introductions are implemented in the aspect language Sally, which will be introduced briefly in the next sections. First, we explain the non-parametric introduction and then the parametric version.

4.1 Introductions in Sally

Sally is a general-purpose aspect language that is highly inspired by AspectJ. Similar to AspectJ Sally provides a pointcut language that is used to identify points in the code where weaving is meant to occur. Another analogy to AspectJ is that each pointcut definition may contain several parameters. While in AspectJ pointcuts are used for dynamic crosscutting code only, in Sally they are also used for static crosscutting code: every introduction refers to a pointcut definition. The parameters passed to an introduction are determined by the corresponding pointcut definition. In contrast to AspectJ Sally does not differentiate between classes and aspects. The aspect-specific features can be added to any class. Thus, Sally supports (like AspectJ) inheritance relationships between aspects what implies that pointcuts may be overridden. It is generally accepted that overriding pointcuts is the fundamental mechanism for reusing aspects (cf. [10] for a further discussion). Like AspectJ Sally binds `this` at weave time.

```
class A {...}
interface NewInterface {...}
class MemberIntroduction {
  pointcut targetClass(?class) =
    equals(?class,A);
  introduction introduceMembers<?class>
      targetClass(?class)
      implements NewInterface {
    public String newString;
    public void doSomething() {...}
  }
}
```

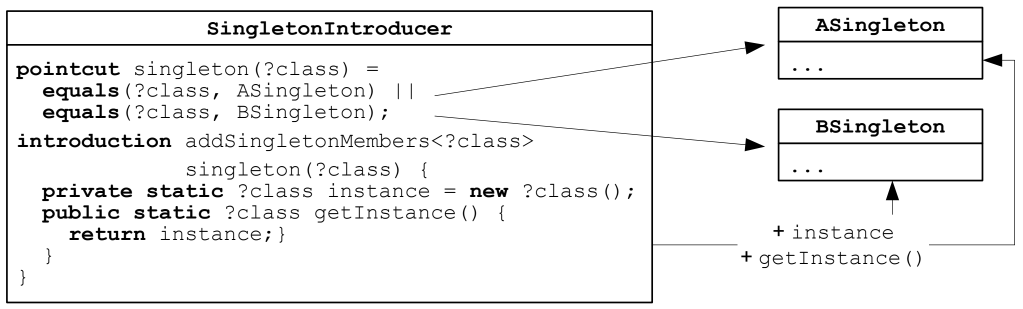
Figure 12: Introductions in Sally

In Sally introductions are members of classes and as such consist of a header and a body. The body defines the members that are to be introduced and looks like a normal class body. The header consists of the keyword `introduction` followed by an identifier, a number of parameters, the referring pointcut, and a number of super-classes and interfaces that are to be introduced. The first parameter of the introduction represents the target class[6].

In figure 12 the class `MemberIntroduction` contains an introduction with the name `introduceMembers` that has a single parameter `?class`, a corresponding pointcut `targetClass` and an interface `NewInterface` that is introduced to the target class. In the example the introduction body consists of the instance variable `newString` and the method `doSomething`.

The introduction refers to the pointcut `targetClass` that binds the parameter `?class` to the class `A`. Similar to the approaches in [25] and [8] Sally uses a logical programming language to reason

[6] The identifier of an introduction is used to reason about the woven code. Since this topic is beyond the scope of this paper it will not be discussed in more detail here.



```
SingletonIntroducer

pointcut singleton(?class) =
  equals(?class, ASingleton) ||
  equals(?class, BSingleton);
introduction addSingletonMembers<?class>
             singleton(?class) {
  private static ?class instance = new ?class();
  public static ?class getInstance() {
    return instance;}
  }
}
```

Figure 13: Singleton implementation in Sally (concrete parametric introduction)

about the object-oriented code and to express crosscuttings. The term `equals(?class,A)` comes from this underlying logical language. The result of this definition is that class `A` implements after weaving the interface `NewInterface`, has the introduced variable `newString`, and the introduced method `doSomething`.

4.2 Concrete parametric introductions

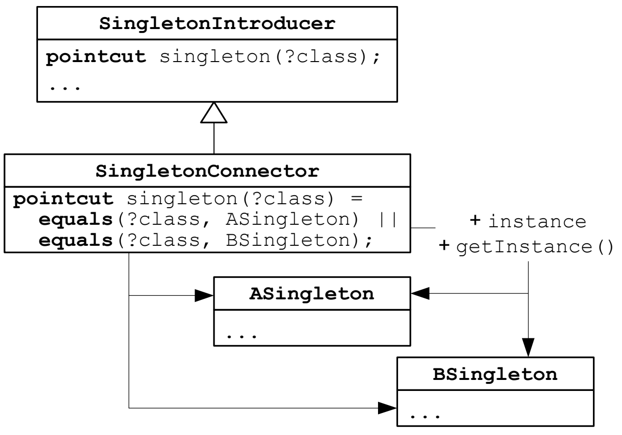
Sally permits to use the parameters passed to the introduction within the introduction body. This means, for example, that types can be passed to an introduction where they can be used as parameters. Figure 13 shows a singleton implementation in Sally. Class `SingletonIntroducer` contains an introduction `addSingletonMembers`. The introductions has a parameter `?class` that defines the class the introduction is applied to and refers to the pointcut `singleton` that defines such classes. The introduction body contains the known methods from the singleton implementation. The type of the instance variable and the return type of the introduced method are determined by the introduction parameter. The pointcut `singleton` binds the variable `?class` to the class `ASingleton` and `BSingleton`. Hence, the introduction is applied to both classes.

4.3 Abstract parametric introductions

The applicability of introductions can be increased substantially if it could be left open at introduction definition time to what classes the introductions are to be applied to. This permits to define libraries of introductions independently of their application. The main benefit is that the introduction can be applied in a different module than its definition. Thus, its application avoids the execution of destructive modifications in the introduction module. This permits a higher level of reusability of aspects. In AspectJ such an abstract introduction is achieved by the container introduction. In Hyper/J such a separation of introduction definition and application is much more natural, since the members to be introduced are normal Java members, while the introduction application is defined within a hypermodule.

Like AspectJ Sally permits to define abstract pointcuts that can be overridden in a subclass. That means that the abstract pointcut in a super-class does not refer to any point in the program, while the overridden pointcut does. Introductions can be abstract by leaving the pointcut the introduction refers to abstract; i.e., the introduction is not connected to any target class. To connect it a subclass has to be defined that overrides the abstract pointcut and binds the parameters to values. Thus, the introduction is realized by the overriding class, although it is defined in a super-class.

Figure 14 illustrates the usage of an abstract introduction. Class `SingletonIntroducer` is specified as in figure 13 except that the pointcut is declared abstract. Hence this class does not perform any introduction for a given application. Class `SingletonConnector` overrides pointcut and binds the parameter `?class` to `ASingleton` and `BSingleton` and, by that, implements the introduction. The benefit of this kind of introduction is that the module that defines the introduction (`SingletonIntroducer`) does not need to be modified if the introduction is applied to different classes. In order to apply the introduction to other classes than the ones mentioned here either `SingletonConnector` has to be modified or another subclass of `SingletonIntroducer` has to be created.



```
SingletonIntroducer
pointcut singleton(?class);
...

SingletonConnector
pointcut singleton(?class) =
  equals(?class, ASingleton) ||
  equals(?class, BSingleton);
```

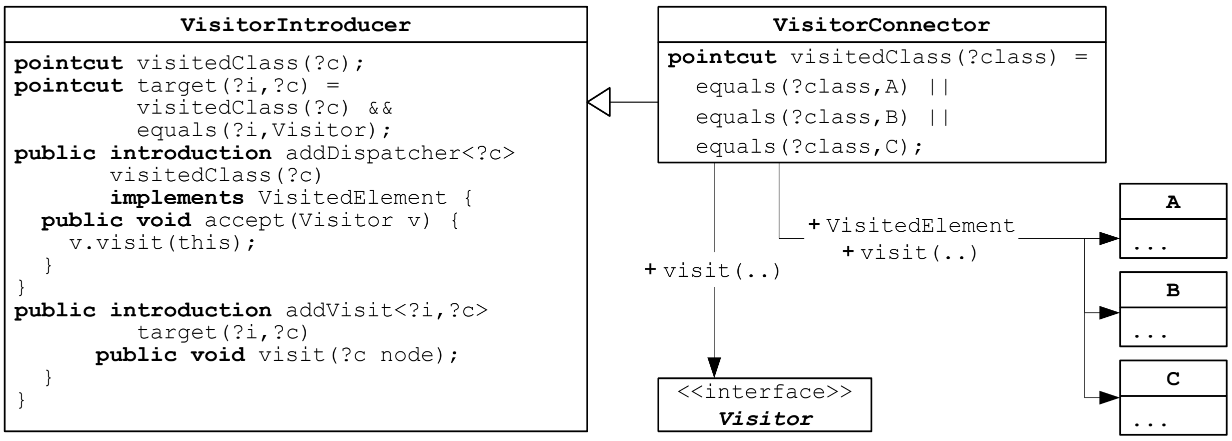
Figure 14: Abstract parametric introductions in Sally

4.4 Parametric multi-introduction

In the visitor example we motivated the necessity to apply an introduction more than once to a target class: the method `visit` needs to be introduced to the interface `Visitor`, however, each time with different parameter types. In Sally introductions are executed as long as different values for the introduction parameters exist.

Figure 15 shows the corresponding implementation of the visitor example in Sally. The class `VisitorIntroducer` contains the introduction `addDispatcher` that introduces the double dispatch method and the interface `VisitedElement` to each class to be visited. Since the referring pointcut `visitedClass` is abstract, the classes need to be defined in a subclass that overrides this pointcut.

The introduction `addVisit` expects two parameters (`?i` and `?c`) from its pointcut `target`. The parameter `?i` is used to de-



```
VisitorIntroducer

pointcut visitedClass(?c);
pointcut target(?i,?c) =
         visitedClass(?c) &&
         equals(?i,Visitor);
public introduction addDispatcher<?c>
       visitedClass(?c)
       implements VisitedElement {
  public void accept(Visitor v) {
    v.visit(this);
  }
}
public introduction addVisit<?i,?c>
         target(?i,?c)
       public void visit(?c node);
  }
}
```

```
VisitorConnector

pointcut visitedClass(?class) =
  equals(?class,A) ||
  equals(?class,B) ||
  equals(?class,C);
```

Figure 15: Visitor implementation in Sally (multi-introduction)

termine the target interface, `?c` is used within the method declaration as parameter. Thus, `addVisit` makes use of a parametric introduction and introduces a method `visit` for each visited class to the visitor interface with the corresponding type name. The pointcut `target` binds the parameters `?i` to the interface `Visitor` and `?c` to each class defined in `visitedClass`. Since `visitedClass` is declared abstract `target` does not bind any variables as long as `visitedClass` is not defined.

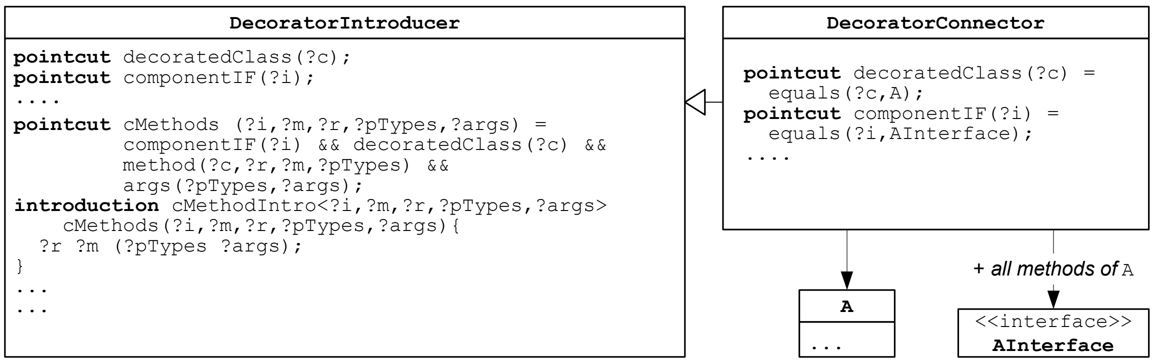
Class `VisitorConnector` extends `VisitorIntroducer` and overrides the abstract pointcut `visitedClass` and binds the parameter `?class` to the classes `A`, `B,` and `C`. Hence, from the point of view of the connector both pointcuts are concrete and bind variables. The variable bindings of `visitedClass` are: (`?c=A`), (`?c=B`) and (`?c=C`). The bindings of pointcut `target` are: (`?i=Visitor, ?c=A`), (`?i=Visitor, ?c=B`) and (`?i=Visitor, ?c=C`).

Hence, the introduction `addDispatcher` is applied to the target classes `A`, `B`, and `C`, Thus, the method `accept` is added to each of these classes. The introduction `addVisit` uses the first parameter as the target class. All three tuples of `target` belong to the same target class (the interface `Visitor`), but differ in the value of their second parameter (`?c`). Hence, the introduction is applied three times to `Visitor`, always with different parameters. Because `?c` is used as a parameter within the introduction, `VisitorConnector` adds three additional `visit` methods to the visitor interface. They differ in their parameter types.

4.5 Unnamed introduction

In section 3.3 we motivated why methods need to be introduced to interfaces whose signatures are not known at introduction definition time. The problem in this context was twofold: first, at introduction definition time it is unknown how many methods are to be introduced and, second, the signatures of the methods to be introduced are unknown. The first problem can be handled by the previously presented multi-introductions. The latter is solved by introducing unnamed methods. That are method introductions whose signatures are determined at weave-time.

Figure 16 shows schematically a decorator implementation in Sally. For reasons of simplicity we only concentrate on the introduction of unnamed methods to the component interface and ignore the rest of the implementation. `DecoratorIntroduction` contains 2 abstract pointcuts for declaring the class to be decorated and the component interface. The concrete pointcut `cMethods` determines all methods included in the decorated class. The introduction `cMethodIntro` uses the passed parameters to create the method signature that will be introduced to the component interface. Special attention has to be paid to the handling of the parameters of the method: only two introduction parameters are used here (`?pTypes` and `?args`). A pointcut may bind a list of values to an introduction parameter instead of binding atomic values only. If a list is bound to an introduction parameter, Sally generates a corresponding list from the parameters. In this concrete example, Sally tests if lists bound to `?pTypes` and `?args` have the same length and introduces a list of parame-



```
DecoratorIntroducer

pointcut decoratedClass(?c);
pointcut componentIF(?i);
....
pointcut cMethods (?i,?m,?r,?pTypes,?args) =
         componentIF(?i) && decoratedClass(?c) &&
         method(?c,?r,?m,?pTypes) &&
         args(?pTypes,?args);
introduction cMethodIntro<?i,?m,?r,?pTypes,?args>
    cMethods(?i,?m,?r,?pTypes,?args){
  ?r ?m (?pTypes ?args);
}
...
...
```

```
DecoratorConnector

pointcut decoratedClass(?c) =
  equals(?c,A);
pointcut componentIF(?i) =
  equals(?i,AInterface);
....
```

Figure 16: Extract of the Decorator implementation in Sally (unnamed introduction)

ter type and value pairs. The decorator connector overrides `decoratedClass` and `componentIF`, hence, the introduction `cMethodIntro` can be executed. The pointcut `cMethods` binds the variables in the following way: `?i= AInterface`, `?m=doSomething`, `?r=void`, `?pTypes=[]` and `?args=[]`. Since `?pTypes` and `?args` is bound to empty lists, the method `doSomething` is introduced to the interface `AInterface` without any parameters.

5. RELATED WORK

In this paper we compared parametric introduction directly with the corresponding mechanisms in AspectJ and Hyper/J. However, other concepts and mechanisms were introduced that are usually not referred to in the context of aspect-oriented programming but are quite similar to aspect-oriented introductions.

5.1 Generic Types

Parametric introductions as implemented in Sally on top of the programming language Java are mainly used to pass types to introductions. Thus, they look quite similar to *generic types* [2] or *parametric types* [19] in Java that also permit types to be passed as parameters. The most obvious difference between both approaches is that an introduction has a direct impact on the target class, i.e., it directly extends the existing target class by additional members while generic types are new types that need to be instantiated and do not influence the existing type structure. Thus, generic types are preplanned while an introduction is an unanticipated type evolution.

5.2 Generic Functions

If a method is introduced to several classes it can be invoked on instances of all of them. So, the body of the method is the same and only the type of its receivers changes. Hence, introduced methods are similar to *generic functions* as provided by Common Lisp [24]. A generic function is a function whose behavior depends on the types of the arguments supplied to it. It contains a number of methods to which the calls may be delegated. The difference between a generic function and a parametric introduction is that within parametric introductions the code to be introduced varies due to the weave time parameters. Using generic functions the programmer has to specify in what way the method body varies in dependence of, e.g., the passed parameters.

5.3 Roles

Roles [23] are temporary views on objects. A role's properties can be regarded as subjective, extrinsic properties of the object the role is assigned to. During its lifetime an object is able to adopt and abandon roles. Thus, an environment of an object can access not only its intrinsic, but also its extrinsic properties. Because of this characteristic roles provide a mechanism that can be compared to introductions (see [12] for a comprehensive discussion of aspects and roles). There are numerous different implementations of the role concept that make use of the composition mechanisms of the underlying programming language. For example, [17] proposes an implementation based on object-based inheritance, [6] uses the Smalltalk-specific handling of incoming messages, [20] a language mechanism called *per-object mixins* and [12] *dynamic proxies*. The major difference between roles and aspect-oriented introductions is that a role works on a single object, i.e., a role does not extend the interface of a class, but the interface of the object they are assigned to. Nevertheless, [20] provides a mechanism called *per-class mixin* that permits to add roles to classes. Thus, the interface of the class is extended. Nevertheless, this mechanism does not permit to declare any variability within the role that is "vitalized" when the role is assigned to the target class. Furthermore, since this concept is provided by an untyped programming language it hard is to compare it to aspect-oriented introductions based on typed programming languages.

5.4 Aspect-oriented logic meta programming

[25] proposes *aspect-oriented logical meta programming* as a mechanism for modularizing concerns. Aspect-oriented logic meta programming means to write logical programs that reason about aspect declarations. Aspect declarations can be accessed and declared by logical rules. So, the weaver is constructed in a logical programming language that provides a number of rules for generating the woven code. The logical programming language proposed in [25] for weaving is called *TyRuBa* (in fact, the here proposed mechanism is implemented using TyRuBa).

In TyRuBa *quoted code blocks* can be declared that permit to use pieces of Java code as terms in logical programs. These code pieces may contain logical variables that are substituted during weaving. In fact, this mechanism, in conjunction with the weaver, provides parametric introductions. The difference between both approaches is that the proposed implementation in Sally is an extension of the programming language Java while TyRuBa handles logical programming separately from object-oriented programming. The weaver implementation in TyRuBa is not connected to the object-oriented programming language but generates object-oriented code. That means, before weaving, no checks are performed, neither on the involved classes nor on the involved introductions. So it is not even determined if quoted code blocks contain Java code at all. This makes software development in TyRuBa error prone. In Sally all classes and introductions are parsed before weaving and type-checking is performed on the involved classes (of course, with the exception of parameterized introductions).

6. CONCLUSION

In this paper we identified introduction as an important mechanism to modularize static crosscutting code in aspect-oriented programming languages. We showed common examples of static crosscutting code in which the introductions of AspectJ and Hyper/J fail to modularize those examples.

As a solution, we proposed parametric introductions, i.e., introductions that receive parameters during weave-time, and indicated how they are realized in the general-purpose aspect language Sally. Moreover, we introduced a mechanism for reusing introductions by using the pointcut language of the aspect language for static crosscutting code. We demonstrated how parametric introductions solve the inadequacies of Hyper/J and AspectJ by applying parametric introductions to the examples where the introductions of AspectJ and Hyper/J failed.

Mechanisms like AspectJ and Sally, which do not bind `this` at introduction definition time, have to offer mechanisms for checking the validity on introductions that are not bound to some target. Otherwise, aspect-oriented development based on introductions will be quite error prone. A discussion of what kind of checking is possible in introductions and the impact of each kind of introductions on type checking will be examined by us in the next future.

Furthermore, the limits of reusability of introductions have to be examined to provide an appropriate language support. Such limits are either name collisions of introduced members that are not handled by any implementation (as far as we known). Also, the *fragile base class problem* (cf. [18]), which means that the reusability of an extension not only depends on the extension itself but also on the base classes, has to be considered.

Parametric introductions are a powerful mechanism that increases the modularization of static crosscutting code. The proposed usage of connecting introductions and pointcut language in conjunction with an inheritance relationship between classes containing introductions increase the reusability of introductions.

7. ACKNOWLEDGMENTS

We would like to thank Arno Schmidmeier for several discussions about this work and the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] AspectJ Team: The AspectJ Programming Guide, http://aspectj.org/doc/dist/progguide/.

[2] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P., Making the future safe for the past: Adding Genericity to the Java Programming Language, OOPSLA 98, Vancouver, October 1998.

[3] Cannon, H., Flavors: A non-hierarchical approach to object-oriented programming, Symbolics Inc., 1982

[4] Clifton, C., Leavens, G., Chambers, C., Millstein, T., MultiJava, Modular open classes and symmetric multiple dispatch for Java, In Proc. of OOPSLA 2000, pp. 130–146

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[6] Gottlob, G., Schrefl, M., Röck, B., Extending Object-Oriented Systems with Roles, ACM Transactions on Information Systems, Vol. 14, No. 3, July 1996.

[7] Grand, M: Patterns in Java, Vol. 1, John Wiley & Sons, 1998

[8] Gybels, K: Using a logic language to express cross-cutting through dynamic joinpoints, Second Workshop on Aspect-Oriented Software Development of the GI, Bonn, February 21-22, 2002

[9] Hanenberg, S., Costanza, P., Connecting Aspects in AspectJ: Strategies vs. Patterns, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD, Enschede, April, 2002

[10] Hanenberg, S., Unland, R., Using and Reusing Aspects in AspectJ. Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA, 2001

[11] Hanenberg, S., Unland, R., A Proposal For Classifying Tangled Code, Second Workshop on Aspect-Oriented Software Development of the GI, Bonn, February 21-22, 2002,

[12] Hanenberg, S., Unland, R., Roles and Aspects: Similarities, Differences, and Synergetic Potential, 8th International Conference on Object-Oriented Information Systems (OOIS) LNCS 2425, Springer-Verlag, 2002, pp. 507-521.

[13] Hannemann, J., Kiczales, G., Design Pattern Implementations in Java and AspectJ, Proc. of OOPSLA 2002, pp. 161-173

[14] Harrison, W., Ossher, H., Subject-Oriented Programming (A Critique of Pure Objects), Proc. of OOPSLA 1993, pp. 411-428

[15] IBM alphaworks, Hyper/J Homepage, http://www.alphaworks.ibm.com/tech/Hyper/J, last access: February 2001.

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J., Aspect-Oriented Programming. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag, pp. 220-242, 1997

[17] Kniesel, G., Objects Don't Migrate - Perspectives on Objects with Roles, Technical Report IAI-TR-96-11, University of Bonn, April 1996.

[18] Mikhajlov, L., Sekerinski, E., A Study of the Fragile Base Class Problem, ECOOP '98, LNCS 1445, Springer-Verlag, pp. 355-382.

[19] Myers, A., Bank, J., Liskov, B., Parameterized types for Java, Symposium on Principles of Programming Languages, pp. 132–145, ACM, 1997.

[20] Neumann, G., Zdun, U., Enhancing object-based system composition through per-object mixins. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Takamatsu, Japan, December 1999.

[21] Ossher, H., Kaplan, M., Katz, A., Harrison, W., Kruskal, V., Specifying Subject-Oriented Composition, TAPOS - Theory and Practice of Object Systems, volume 2, number 3, 1996, Wiley & Sons, pp. 179-202

[22] Ossher, H., Tarr, P., Using multidimensional separation of concerns to (re)shape evolving software. Communication of the ACM, 44(10), 2001, pp. 43-50.

[23] Pernici, B., Objects with Roles, in: F.H. Lochovsky, R.B. Allen (Eds.): Proceedings of the Conference on Office Information Systems, SIGOIS Bulletin, vol. 11, no. 2/3, ACM Press, New York, 1990, pp. 205-215.

[24] Steele, G: Common Lisp: the Language, 2nd Edition, Digital Press. 1990.

[25] De Volder, K., D'Hondt, T., Aspect-Oriented Logic Meta Programming, Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99. LNCS 1616, Springer-Verlag, 1999, pp. 250-272.