

# Implementation of a Role Language for Object-Specific Dynamic Separation of Concerns

Kasper B. Graversen<sup>\*</sup>  
IT University of Copenhagen  
Glentevej, Copenhagen  
Denmark  
kbilsted@it-c.dk

Kasper Østerbye  
IT University of Copenhagen  
Glentevej, Copenhagen  
Denmark  
kasper@it-c.dk

## ABSTRACT

This paper presents an implementation of a rich role model named Chameleon, which has close resemblances to aspect-oriented programming. A role is a perspective (a dynamic extension) to an existing object that can both provide new state and behaviour as well as modify existing behaviour. One important contribution of the Chameleon model, is the notion of “constituent methods”, which allows role-behaviour to hook onto behaviour in the intrinsic object. Consequently, this enables one to define *aspects on perspectives of objects, rather than directly on objects themselves*. The Chameleon model has been implemented as an extension to the Java language. A main contribution of this paper is a presentation of how the dynamic extension mechanism of Chameleon is transformed into plain Java, thus highlighting the different semantic aspects of Chameleon in a implementation oriented manner.

**Keywords:** Programming language, implementation.

## 1. INTRODUCTION

Many researchers have argued the advantages of roles in modeling and implementation [3, 5, 6, 10, 13, 17]. Roles allow objects to evolve over time, they enable independent and concurrently existing views (interfaces) of the object, explicating the different contexts of the object, and separating concerns. Generally roles are a natural element of our daily concept forming. Roles in programming languages enable objects to have changing interfaces, as we see it in real life—things change over time, are used differently in different contexts, etc.

These advantages serves as our motivation for examining the implementation of the Chameleon role model (which is elaborated in [7]). A significant contribution of this paper is the presentation of the technical implementation of a role model. An important contribution of the Chameleon model is the notion of constituent methods, methods that in essence serves the same purpose as advices from the AspectJ language [11].

The prototype implementation of Chameleon is designed as an extension to Java, and produces standard Java code. In general, it is possible to transform Chameleon into Java, but some inconveniences have surfaced, both in relation to typing and in object layout. The Chameleon prototype has

been implemented using OpenJava [18], which provides limited possibilities for syntax extension, but extensive high-level support for program transformation.

The paper is structured as follows. First a brief introduction to roles is given, followed by an elaboration of some of the significant implementation considerations. The paper ends with related work and discussion of further issues.

## 2. ROLE DEFINITIONS

In Chameleon, we use certain terminology, which is founded in modelling with roles.

**Intrinsic:** An entity which can play roles. When roles have roles attached, they too are referred to as intrinsic.

**Extrinsic:** The roles played by an intrinsic. Depending on the view, an entity can be described as being both intrinsic and extrinsic.

**Entity:** Denotes either an object or a role.

Roles is a concept to augment statically typed class-based programming languages. By this we mean, that roles should not be used *instead of* objects, but *with* objects—enriching the languages expressiveness. A role is a dynamic extension of a class, meaning an extension that can be attached and detached at run-time. Roles have many of the qualities of both inheritance and aggregation, and thus may be hard to not to misunderstand when encountered for the first time. In [7, chap. 4] a more thorough analysis of roles and their relations to inheritance and aggregation is provided. A role definition contains a set of fields and methods (properties), in many ways similarly to a class definition. But a role do not describe a phenomenon as a class does, rather it describes a perspective of an object. These perspectives are used by other entities of the application to access the object—a special way of knowing and using the object, defining a specific context. Each role constitutes a distinct view on the intrinsic, each with its own set of properties. Generally, the roles are independent of each other, meaning that whether or not an intrinsic is used through a role  $\mathcal{R}$  does not affect its use through another role  $\mathcal{S}$ . This means that in order for a role to have any effect it must be used explicitly (otherwise it would affect other roles, and hence not be independent!).

Different from class-based specialization is, that it takes place on the instance level rather than on the class level. This means that for a subset of objects it is possible to dynamically attach and remove roles, e.g. assign a teacher role

---

<sup>\*</sup>corresponding author

(a role for persons). This reflects the real world many applications try to model. Roles have a specific field **intrinsic** object, which provide access to the intrinsic of the role (similar to **super** for classes).

As roles define perspectives, they cannot be used as independent entities, i.e. they must always be attached some object. A role shares the properties of the object it is a role for, so changing the name of a teacher, changes the name of the person (which contains a name field) and affects all other roles using the persons name. Finally, one can define a role for a role (i.e. specializing the perspective of a teacher to be a “supervisor”)—which similarly operate on subset of teacher roles. Statically typed class-based languages are awfully static, and are not tailored to model the dynamic world we live in. This surprises many, despite critique has been around for a while, e.g. Harrison and Ossher’s critique dating back to 1993 [9]. In the real world nothing is static.

**Example 1:** Having a person object, and wanting this person to become a teacher, the person object must be destroyed and all references to the person must be redirected to refer to the freshly instantiated teacher object, and the state of the person object must be transferred to the teacher object. Such an operation is not possible in most languages, and implementing a scheme manually could raise the complexity of the implementation and may be a costly operation[5]. Roles, on the other hand, are dynamic—it is always possible to attach a role needed for the situation. □

Summary of roles:

Roles are dynamic like aggregation, but shares properties and types with its intrinsic like inheritance. With roles new possibilities arise. Most significantly:

- Independent views of an object can be implemented without using multiple inheritance, the object is thereby not blurred with the mix of inherited properties, as these are clearly separated in roles.
- Accessing an object is restricted to the role which the object is referenced through (multiple views on objects).
- One can apply many roles of the same type to an object, e.g. modelling a person being a teacher at several different places.
- Roles can be used instead of building a hierarchy of very similar classes.

Evidently classes and roles serves different purposes: Class hierarchies are classifications of entities into disjoint entity types, whereas role hierarchies prescribe disjoint behavioural contexts for entities (a hierarchy for a type of classes). A Class hierarchy can co-exist with several role hierarchies, where every class may serve as the root for a role hierarchy [6].

In Chameleon, roles are first class entities, meaning they can be instantiated, referenced to, and used similarly to objects.

## 2.1 Java Derivatives

As Chameleon is implemented as a language extension to Java, it was important to us that the extension was as seamless possible to the programmer. By this we mean, that the

style of the language must be maintained in the extended language. Due to this criterion, the following properties of roles derive from the Java language. A role has:

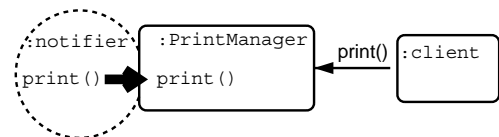
- At least the same types as its intrinsic.
- Access to all but private properties of its intrinsic.
- Late binding of self (all methods virtual).
- Static typing, meaning that if there are no type errors during compilation, there are no type errors during execution. Only on objects that are extended with a given role, is it allowed to invoke that role’s methods.

## 2.2 Constituent methods

Many role- and delegation-languages generally fit the above descriptions e.g. [2, 6, 12, 17]. In such systems, it is difficult to functionally decompose an intrinsic using roles, since in order for a role to function, it must be explicitly referenced. This is particularly problematic when several dynamic roles must execute when invoking a method on the intrinsic, as the roles must be piggy-backed, which makes it hard to add and remove them.

We extend the notion of roles with “constituent methods” originating from ideas in [14]. Constituent methods are different than normal methods in many ways. First, they cannot be invoked explicitly. Constituent methods hook on to methods in its intrinsic, and are automatically executed before, after or instead of the original methods, and have the possibility to change input arguments and return value. With this extension roles are no longer independant of eachother.

Constituent methods are comparable to advices in AspectJ, where pointcuts in chameleon are defined by which methods the constituent method hooks onto. Using constituent methods, an object can be extended without use of role references, and hence represents true separation of concerns. A usage is depicted on figure 1, where a print manager is equipped with a role that when printer problems arise notifies the system administrator.



**Figure 1: A print manager (object) with a role (with a constituent method) which automatically notifies the sysadm., when the printer malfunctions. Clients (objects) are unaware of the roles’ presence.**

When more than one constituent method hook onto a method, priorities accompanying the constituent methods ensures indetermistic behaviour. However, unlike static aspect languages, the priorities are modifiable per object at runtime, enabling the order of execution to conform to the objects context.

## 3. ROLE ATTACHABILITIES

A role can be a role for any static specialization of its intrinsic. This is depicted on the left side of figure 2,

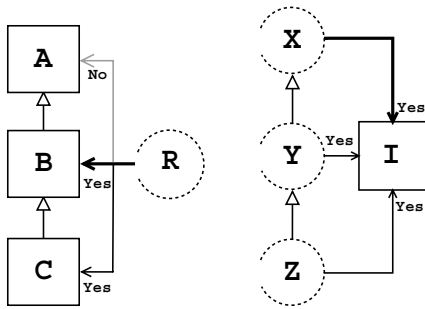


Figure 2: Role  $\mathcal{R}$  for intrinsic  $\mathcal{B}$  can also be attached to  $\mathcal{C}$  but not  $\mathcal{A}$ . On the right side: When  $\mathcal{X}$  is a role for  $\mathcal{I}$ , then also specializations of  $\mathcal{X}$  are roles for  $\mathcal{I}$ .

which shows that role  $\mathcal{R}$  for  $\mathcal{B}$  can also be attached to any subtype of  $\mathcal{B}$ , but not to supertypes of  $\mathcal{B}$ .

Any static specialization of a role  $\mathcal{X}$ , is a role for the same intrinsic as  $\mathcal{X}$ . This is depicted on the right on figure 2, and shows a static specialization of a role  $\mathcal{X}$  (that is a role for  $\mathcal{I}$ ), into the role  $\mathcal{Y}$ , which is further specialized into the role  $\mathcal{Z}$ .

### 3.1 Dual inheritance

When specializing an intrinsic by both static and dynamic inheritance, we define the dynamic specialization to be stronger, since it is applied after the static specialization (similarly a static specialization for a static specialization prevails).

Let an intrinsic  $\mathcal{I}$  define a method `foo`. A static specialization  $\mathcal{S}$  redefines `foo` to calling `bar` defined in  $\mathcal{S}$ . A role  $\mathcal{R}$  defined for  $\mathcal{I}$  defines a method `bar`. We imagine, that  $\mathcal{S}$  and  $\mathcal{R}$  are developed independently of each other. In a running environment  $\mathcal{R}$  is attached  $\mathcal{S}$  (see figure 3). Calling `foo` on the role will execute `foo` in  $\mathcal{S}$  and `bar` in  $\mathcal{R}$  due to the late binding of self.

There is a problem however,  $\mathcal{S}$  and  $\mathcal{R}$  were designed on the basis of  $\mathcal{I}$ , which did not define a `bar`, thus their `bar` may be completely unrelated. This situation we call accidental overwriting. Forgetting that “the role always prevails when attached” can yield unexpected executions. However, this is no different than languages with polymorphism and dynamic binding (like Java). Given the method definition `foo(InputStream in){ in.read(); }` it is indeterminable what code is executed when executing `in.read()`, since anything having the type `InputStream` can be passed to on—e.g. `PipedInputStream` which has a different definition of `read` than `InputStream`.

Had  $\mathcal{I}$ , on the other hand, defined a `bar`, the situation would have been vastly different, as  $\mathcal{S}$ ,  $\mathcal{R}$  would specialize the method and thus not be unrelated.

## 4. ROLEMANAGER

We have now presented the rudimentaries of our role model. The rest of the paper continues on a technical level. Here we elaborate on the execution model. The next section deals with how classes and roles are transformed to Java.

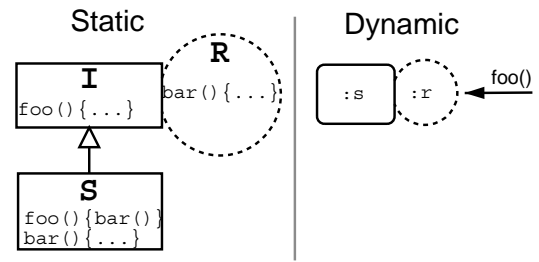


Figure 3: A problematic situation for independently introduced `bar` methods

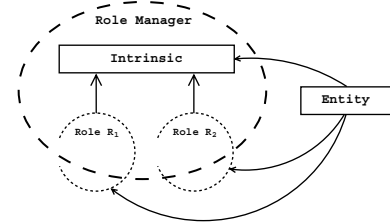


Figure 4: An intrinsic embraced by its role manager. The role manager intercepts every access to the intrinsic. Accessing properties of roles are not intercepted by the intrinsic's role manager.

The introduction of roles in the language requires a form of role management—that is, house keeping and related functionality to be at disposal. It should be possible to inquire an object of the roles it currently plays, it should be possible to attach and remove roles from it. It should be possible to set up constraints prohibiting two roles to co-exist on an object. Finally, method dispatching must take place in a controlled fashion, enabling constituent methods to be called before, after or instead of the original method. We use a meta-object we call “role manager” to be responsible for this. Each object in the system has exactly one role manager. Figure 4 shows a situation of an object with roles. The role manager can be thought of as a membrane embracing an object and controlling access to it. This model resembles the internal structure of Composition Filters[1].

When roles have roles attached they become intrinsic for those roles. Thus, all roles can potentially become an intrinsic, hence a role manager is attached all role instances. Attaching a role on role  $\mathcal{R}$ , the role will be managed by  $\mathcal{R}$ 's role manager. The run time situation is thus more precisely depicted on figure 5.

For the role manager to accomplish its tasks, it must be in control of the method dispatching. This is done by transforming all methods to consult their role manager as the first thing they do—thereby handing over the control. On figure 6 a simplified model is depicted: A call to method `foo` is “intercepted” by the role manager. `foo` is compiler generated and acts as a shell for the original `foo` method (which has been renamed to `org_foo`). The “shell method” invokes the role manager, which then freely can decide if any constituent method, e.g. `bar` is invoked (step 3+4) and even if the original method (5+6) should be invoked. If `bar` is an “around method” the steps 5+6 may not be performed,

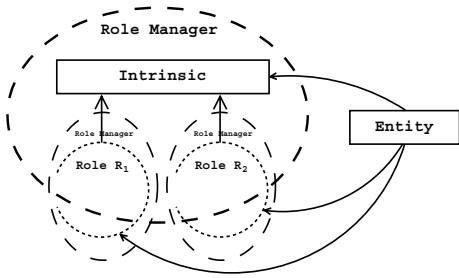


Figure 5: Since every role potentially can become an intrinsic, all roles are equipped with a role manager.

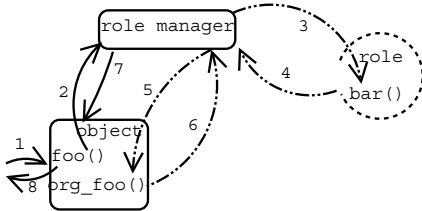


Figure 6: The role managers’ “interception” of a method call `foo` on an object. The steps 3+4 and 5+6 are optional and can take place in the reverse order.

if `bar` is an “after method” rather than a “before method”, steps 3+4 are executed before 5+6.

## 5. IMPLEMENTATION STRATEGIES

We have chosen to extend a language rather than creating a new language. This has several advantages: It is vastly easier to implement, existing code can easily be converted, existing developer software can be used and people generally are more willing to use an extension to a known language, rather than a completely new language. We chose to extend the Java language due to its wide use and since we knew that language best, hence it would be easier to capture the spirit of the language, and carry that on to the extension.

Simply put, the task was to implement object-based inheritance in a class-based inheritance language. In this paper we have chosen to focus on three central topics: representation of intrinsic and roles, and late binding of `self`. We describe how the source code is transformed into plain Java, which then is compiled using any standard Java compiler. Since roles can be intrinsic for other roles, all transformations applied to intrinsics are, with few exceptions, applied to roles as well (still we feel a separation is healthy to maintain).

Our approach to describing the transformations is to list some requirements and explain how they are accommodated.

### 5.1 Transformations of intrinsics

*We want roles and constituent methods on field access.* We do this by generating accessor methods for all fields in the intrinsic. Then all “direct access” to fields are transformed to using these accessor methods, e.g. `i=3` becomes `seti(3)`. Further, all side-effecting operators (`++`, `--`, `+=`, `...`), are transformed to explicit “operator method calls” (generated methods). `i++` becomes `postInci()`. The rest of the opera-

tors are untouched. Array access is similar to field access, except the number of possibilities of access ( $6+2 \cdot \text{dimension}$ ), and hence is solved by creating appropriately many accessor methods each handling a specific form of access. Since all field access has been converted to explicit method calls, they are interceptable the same way methods are (see fig. 6).

*We want late binding of self.* Basically we implement a delegation scheme as presented in Lieberman [15]. Delegation means, that when a message receiver cannot execute a received message, it passes it on to its parent (equivalent to a super), but retains the original receiver as `self`.

During code transformation, the code is transformed to implementing late binding of `self`, as well as redirecting all calls to the role manager (as described in section 4). Transformation of all methods is done by the following steps, but is exemplified with the method `x`:

1. Generate an overloaded method `x` taking extra arguments `self` and a boolean. The method delegates the control of execution to the role manager. If `x` returns a value, return the result of computation from the role manager. The boolean denotes whether the call came from outside or inside the entity (if the later, delegation is temporarily disabled).
2. Rename the original method to “`org_x`”. This protects it from being invoked by other entities in the application (except the role manager).
3. Generate another `x` with identical signature to `org_x`. This method is the receiver of all method invocations. Its sole purpose is to call the forwarding method (generated in step 1) by invoking it with `this` as the first argument, thereby passing on `self`, which is retained through the rest of the calls.
4. Expand all “implicit method calls”—all calls not on an explicit reference or on the `intrinsic` reference. The expansion adds a `self` reference and a boolean field as arguments. Further the calls are changed into invoking the method on the `self` reference passed to the method. The `self` reference is passed due to delegation, and the boolean field denotes whether the call is a call to the intrinsic. Consider a role invoking a method on its intrinsic, the intrinsics role manager needs to know that the call is intended for its intrinsics’ method, hence the invocation should not be made on the `self` reference also passed on to it. If we did not have the boolean and both intrinsic and role defines the method `foo`, and the role calls `intrinsic.foo()` it would always be the roles `foo` which will be invoked.
5. Explicit method calls are not expanded, since if it is a call on a Chameleon-compiled entity, they will be caught by the method generated in step 3 (on that entity). If the call is on a non Chameleon-compiled entity (e.g. an API call) no harm is done.

The transformations ensure two things: Control is handed over to the role manager, so that constituent methods are executed (cf. step 1) and new method calls will start out by remembering the `self` reference (cf. step 3) on which they make all invocations (cf. step 4). Further, step 3 ensures “backward compatibility” to other entities not compiled with our compiler (since name and arguments are retained).

In the running system there is now both a *self* pointer and a *this* pointer. *self* refers to the original message receiver, and *this* refers to the current object executing.

On figure 7 a diagram shows an execution of the code below. The intrinsics `foo` and the roles `bar` methods are executed (the gray regions). The rest of the diagram is automatic forwarding.

```
class I {
  void foo(int i) { bar(i);}
  void bar(int j) {...}
}
role R roleifies I {
  void bar(int k) {...}
}
```

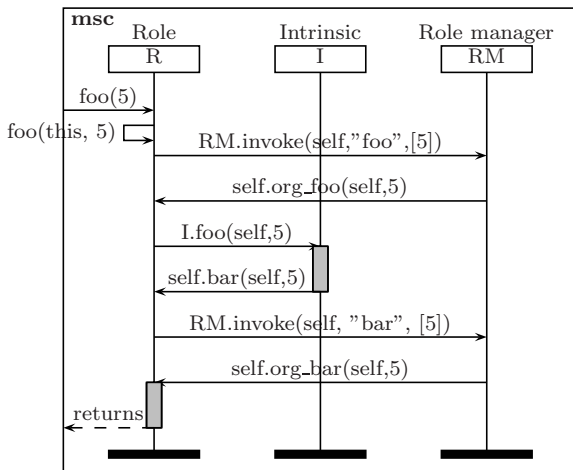


Figure 7: Simplified diagram of a method execution.

## 5.2 Transformations of roles

In addition to the above transformations, roles are further transformed during compilation.

We want the role to have the types as its intrinsic. We let the role extend its intrinsic. This approach has two drawbacks. First, all fields of the intrinsic are redundantly inherited, as state must be shared between role and intrinsic. Remember that when dealing with roles there are always two instances in play—the role and its intrinsic. Secondly, since Java has single inheritance, a role cannot both specialize a role and change intrinsic, as this requires inheriting from two entities—the current Chameleon implementation cannot implement situations such as figure 8. The advantage of the approach is that roles has access to protected properties of its intrinsic, which are inaccessible had interfaces been used to getting the proper types.

We want constituent methods to function on calls on roles. Methods in the intrinsic not redefined by its roles are inherited by the roles due to the static inheritance introduced above. In such situations we are dealing with two method instances, one in the role, and one in the intrinsic. This is a problem, since constituent methods only hooks onto the “original method instance” (using the intrinsic role manager). To make constituent methods work on method calls

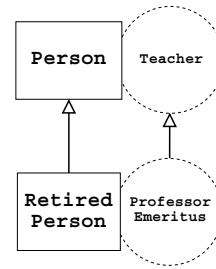


Figure 8: A situation the role implementation cannot represent (due to Java’s single inheritance). Interestingly the figure further shows, that in a role hierarchy, the intrinsic need not be the same.

on the role, we pass on the request to the original method. So for all methods in the intrinsic not redefined in the role, forward methods are created in the role, which simply call the corresponding method in the intrinsic. Redefining a method `foo` in the role that do not call the intrinsic `foo` to be executed when `foo`-calls are made on the role reference. This is a natural and similar behaviour as static aspects, and conceptually correct.

We want fields to be shared among the intrinsic and its attached roles. Fields in Java are non-virtual. Redefining a field shadows the super class’ field. In Chameleon, non-shadowed fields are shared with its roles. This is easily accomplished since all field access has been changed to method calls. For all fields in the intrinsic not redefined in the role, generate *forward accessor methods* rather than regular accessor methods. Forward accessors merely forward the access to the intrinsic instance.

## 6. CONSTITUENT METHODS AND DELEGATION

In [7, chap. 8] we found, that delegation [15] fails on both the technical and conceptual plane during the execution of constituent methods —*self* sometimes must refer to a completely different entity, than prescribed by the rules of delegation. First an example showing the technical inconsistency:

**Example 2:** Let  $\mathcal{I}$  be an intrinsic with the method  $\mathcal{M}$ . Let  $\mathcal{I}$  have the role  $\mathcal{R}_1$  which has a constituent method  $\mathcal{C}$  hooking on to  $\mathcal{M}$ . The constituent method  $\mathcal{C}$  invokes the method  $\mathcal{O}$ . Let another role  $\mathcal{R}_2$  be attached to  $\mathcal{I}$ , and let a method invocation call  $\mathcal{M}$  on this role, as illustrated on figure 9.  $\square$

According to delegation rules, *self* refers to  $\mathcal{R}_2$  when  $\mathcal{C}$  is invoked. This leads to an invocation of  $\mathcal{O}$  on  $\mathcal{R}_2$  which is impossible. If the call was made directly on  $\mathcal{I}$ ,  $\mathcal{C}$  would equally fail in invoking  $\mathcal{O}$ .

It could be defined that when executing constituent methods, *self* refers to the role in which the constituent method is defined. Such a definition, however, changes the semantics of the language. When executing methods in roles, *self* always refers to the role itself, or to a role for it (if the ini-

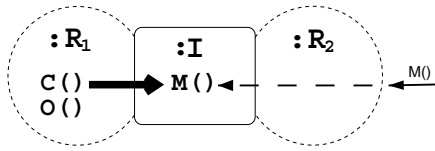


Figure 9: The intrinsic  $\mathcal{I}$  with the roles  $\mathcal{R}_1$  and  $\mathcal{R}_2$  attached and a call  $M()$  on the role  $\mathcal{R}_2$ . The method  $\mathcal{C}$  in  $\mathcal{R}_1$  is a constituent method for  $\mathcal{M}$ .

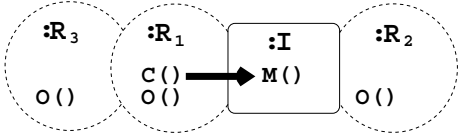


Figure 10: The intrinsic  $\mathcal{I}$  with the roles  $\mathcal{R}_1$ ,  $\mathcal{R}_2$ ,  $\mathcal{R}_3$  attached. Depending on from where  $\mathcal{M}$  is invoked,  $\mathcal{O}$  in either  $\mathcal{R}_1$  or  $\mathcal{R}_3$  is executed.

tial call came from such). With the new definition, this is no longer the case, when the initial call come from a role for the role.

### 6.1 Latest possible binding of self

We have invented a scheme we name *latest possible binding of self*, which is an extension to the delegation scheme. During execution of constituent methods, *self* refers to the most specific entity possible. *self* within a constituent method is set according to the following rules, which take outset in the example 2 and is illustrated on figure 9.

- When the call to method  $\mathcal{M}$  is on  $\mathcal{R}_1$  or a role for  $\mathcal{R}_1$ , *self* is set to that particular role.
- Otherwise *self* is set to the role defining the constituent method ( $\mathcal{R}_1$ ).

Taking outset in figure 10, we see a system of an intrinsic with three roles attached. As in example 2, the method  $\mathcal{C}$  is a constituent method for  $\mathcal{M}$ , which invokes the method  $\mathcal{O}$  (which is our example for all implicit method calls in constituent methods).

Depending on from where a method invocation message to  $\mathcal{M}$  comes, different  $\mathcal{O}$ 's are executed. If  $\mathcal{M}$  is invoked on  $\mathcal{R}_3$  then  $\mathcal{R}_3$ 's  $\mathcal{O}$  is executed. In all other cases ( $\mathcal{R}_1$ ,  $\mathcal{I}$ ,  $\mathcal{R}_2$ ) it is  $\mathcal{R}_1$ 's  $\mathcal{O}$  which is executed.

This definition is as close as possible to the definition of late binding of self. But it also means, that creating a role for a role containing a constituent method, one must be careful when completely redefining the meaning of a method—however, this is no different than when using static inheritance. The rule, although it at first may sound strange, is conceptual founded: Assume  $\mathcal{O}$  within  $\mathcal{R}_1$  writes some information to the screen. Assume then, that  $\mathcal{R}_3$  is a “muting role” hence it re-defines  $\mathcal{O}$  not to print on the screen. Invoking  $\mathcal{M}$  on  $\mathcal{R}_3$  conceptually has the effect, that  $\mathcal{M}$  is

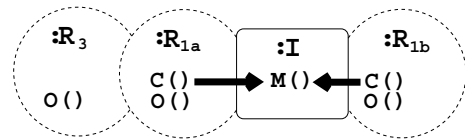


Figure 11: Intrinsic  $\mathcal{I}$  with the roles  $\mathcal{R}_{1a}$ ,  $\mathcal{R}_{1b}$ ,  $\mathcal{R}_3$ . Depending on from where  $\mathcal{M}$  is invoked and who is executing either the  $\mathcal{O}$  in  $\mathcal{R}_{1b}$  or  $\mathcal{R}_3$  is executed.

invoked with nothing printed on the screen. Whereas running it from anywhere else, information should be printed on the screen.

When there are several roles of the same type on the object, things are slightly more complicated. We then have to check both the origin of the call, and which constituent method is currently executing. On figure 11 we see such a situation.  $\mathcal{M}$  is invoked on  $\mathcal{R}_3$ , for the example  $\mathcal{R}_{1b}$ 's  $\mathcal{C}$  is executing before  $\mathcal{R}_{1a}$ : It should execute its own  $\mathcal{O}$ , rather than  $\mathcal{R}_3$ 's which, however, must be invoked when  $\mathcal{R}_{1a}$ 's  $\mathcal{C}$  is executing!

Using the following algorithm, we detect what  $\mathcal{O}$  is to be executed. We examine the origin of the call (*self*) and the currently executing constituent method (*this*).

```

if(self == this || self.intrinsic* == this)
    self.o();
else this.o();

```

where \* denotes a recursive search. Line 1+2 checks if the origin of the call is either: ‘the currently executing constituent method’, or ‘a role for the currently executing constituent method’. If so, call its method. Line 3: If not, call the method in the currently executing constituent method.

If applying an empty role  $\mathcal{R}_4$  on top of  $\mathcal{R}_3$  and calling  $\mathcal{M}$  on this role instead, the algorithm still works, since  $\mathcal{O}$  is “inherited” from  $\mathcal{R}_3$ , hence the invocation on *self* is sound.

## 7. ROLE MODIFIERS

We have now shown the foundation of our role model. Building on top of this is model shows very powerful. We now describe one out of many role modifiers defined in Chameleon. The extension, however, shows how easy it is to extend the role concept to also cover the static aspect domain.

### 7.1 Life roles

In relation to aspects, the biggest problem with roles is their dynamic nature. Even though we have advocated dynamic systems, researchers have presented many relevant yet static aspects. Using roles with constituent methods as static aspects turn out to be problematic, as every time an object of a given type is instantiated, it must be ensured the right set of roles are attached (which there is no compiler support for). Secondly, the code is cluttered with unrelated role instantiations and attachments. In static aspect languages ‘aspects are just there’. What is needed is to be able to declare a

role such that whenever its intrinsic is instantiated, the role is instantiated and attached the intrinsic instance. We call such roles “life roles”, since they are sort of roles for the intrinsics’ life time. To declare a role a life role, simply add a `liferole` modifier to the role declaration.

Although this seems a minor adjustment to the role model, it rises possibilities in the aspect domain. Since roles can be roles for roles, we say we can apply aspects *on a perspective of an object, rather than on the object itself*. When a role is removed from an intrinsic, so are the roles’ roles. This means, that *aspects become dynamic—only existing when needed*. A constituent method do not define an aspect, as aspects by definition are cross-cutting. However, constituent methods in unison certainly can define aspects. Whether aspect logic is defined in one separate block (an aspect) or in several separate blocks (roles), makes no difference. This is similarly argued in [10].

The implementation of class roles is straight forward. We simply insert code into the constructor of the intrinsic instantiating and attaching the role instance to itself.

## 8. RELATED WORK

As was clear from the discussion in section 3 and 6, there are advanced rules for the self binding in a role language, many of these design deliberations are naturally strongly related to similar issues regarding multiple-inheritance in class based, as well as, delegation based systems. In [4], Carré and Geib presented a thorough discussion on problems of multiple inheritance. Their notion of point of view corresponds to a large extend to our notion of “latest possible binding of self”, in combination with the fact that one accesses a role, and not single combined object (as is the case with multiple inheritance). This is also similar to the 1987 proposal for multiple inheritance in Self [19]; the similarity arise from the fact, that there are strong similarities between delegation and roles.

### JAC

JAC [16] is a framework for distributed applications which supports dynamic wrappers. JAC is centred around wrappers containing wrapper- and role-methods, which are comparable to roles containing constituent methods and normal methods. But the dynamic wrappers are wrappers, not seamless extensions of an object. There is no late binding between an object and its wrappers. Wrappers do not inherit the objects methods, so these must be invoked on an explicit reference. Role methods must be invoked in a cumbersome way using an explicit call such as `o.invokeRoleMethod("myRoleMethod", args)`. Which lack strong typing, and forces the programmer to first marshal the arguments and box simple types before invocation.

However, since the wrappers are just a chunk of code, wrappers can also be used in pointcuts, that is, specifications of certain points in the execution of an application. Our role model lacks a ‘pointcut mechanism’, as roles are bound to the objects that they are roles for. A similar functionality can be implemented in our role model by the use of roles and constituent methods, where the constituent methods invoke the same method in a helper-object containing the functionality as the wrapper.

### Lava

Lava [12] is a delegation language rather than a role language. There is a high resemblance between the two concepts (delegation is typically used in role languages). Still, the focus is different. Kniesel’s interests seems to be unanticipated dynamic adaptation of applications, that is, changing the behaviour of active unloadable components by adding more components to them. Although many similarities, Lava lacks support for constituent methods, thereby misses the possibility to create separated aspects.

### Gottlob et al.

The role model in [6] is close to ours. The main differences are, that roles are divided into roles and qualified roles, where the later has an id, which the first hasn’t. It seem qualified roles are mainly used when several roles of the same type is applied an object, where the id is used to easily distinguish the roles. The second main difference is, that roles can be subclassed, that is a class’ super can be a role. In our role model we clearly separate classes and roles by declaring only roles can extend classes, not the other way around. We can easily implement this feature, but we have not yet assessed its usage or conceptual founding.

Gottlob et al’s role model is implemented in Smalltalk, which is dynamically typed. For that reason, we assume, there has not been put emphasis on concepts we have emphasized, such as: static typing, the roles’ type, what roles can access in the object, and finally late binding of *self*—all of which are missing in their role model. Further, their role model does not include some of the “advanced concepts” such as constituent methods, life roles, and other extensions.

## 9. DISCUSSION

Chameleon is currently implemented by a transformation into Java. From a research perspective, it has allowed us to develop a prototype in a reasonable fast manner, but it has its drawbacks. One problem is the use of inheritance, since a role extending another role cannot change intrinsic, which however likely or unlikely a situation represents a specialization that conceptually makes sense. In addition, it means that many instance variables are duplicated quite unnecessarily. The present implementation of the role-manager uses reflection, which has severe performance penalties. We believe that it is indeed possible to generate a role-manager that do not use reflection, because it is given by the type of intrinsic which methods can be augmented with constituent methods. The design with the role manager is interesting, as it provides a separation of what is described using new role specific syntax, and meta-level implementation. The idea is that one can write specialized role-managers without changing the surface syntax.

As mentioned in the introduction, we see roles as an interesting accompaniment to aspect-oriented programming, in that roles from its outset is founded in analysis of the problem domain (e.g. the work of Kristensen [13, 14]), where the motivation for aspects has risen from the solution domain, to some extend being a advanced mechanism for physical structuring of source text. One example where we believe that the problem domain foundation of roles has shown its strength is the fact that it has been clear from the beginning that one could have roles on roles (on roles etc). Another

situation is that the role model allows us to put two roles of the same type onto the same object. In AspectJ it does not make sense to apply the same aspect twice (As soon as an aspect is parameterized in some way, we expect this need will arise).

There are several directions for future work. First, it would be nice to restructure the implementation, so that we are not bound by transforming into Java. Secondly, there are ample opportunities for optimizations, especially it is interesting to look into the role-manger, and constituent methods—applying run-time code generation to optimize the invocation of constituent methods would reduce the penalty for invocation, but make it more costly to attach and detach roles with constituent methods. Finally, there are still some details of the role mechanisms that has to be worked out, e.g. the above modularization mechanism that allow a number of roles to describe a cross-cutting concern better (and to establish a better conceptual foundation in problem domain terms what a cross-cutting concern really is).

## 10. REFERENCES

- [1] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *ECOOP*, volume 615, pages 372–395. Springer-Verlag, 1992.
- [2] A. Albano, G. Antognoni, and G. Ghelli. View operations on objects with roles for a statically typed database language. *Knowledge and Data Engineering*, 12(4):548–567, 2000.
- [3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs.*, 1993.
- [4] B. Carré and J.-M. Geib. The point of view notion for multiple inheritance. In *ECOOP/OOPSLA*, pages 312–321. ACM Press, 1990.
- [5] W. W. Chu and G. Zhang. Associations and roles in object-oriented modeling. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 257–270, 1997.
- [6] G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.
- [7] K. B. Graversen and J. Beyer. Chameleon, August 2002. Masters thesis. IT-University of Copenhagen.
- [8] K. B. Graversen and K. Østerbye. Aspect modelling as role modelling, 2002. OOPSLA’02 Workshop on “Tool Support for Aspect Oriented Software Development”.
- [9] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). pages 411–428. OOPSLA, 1993.
- [10] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. *Proceedings of Net.ObjectDays*, 2002.
- [11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
- [12] G. Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP ’99*, volume 1628, pages 351–366. Springer-Verlag, 1999.
- [13] B. B. Kristensen. Object-oriented modeling with roles. In J. Murphy and B. Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag, 1996.
- [14] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.
- [15] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA*, volume 21, pages 214–223, 1986.
- [16] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns*. Springer, September 2001.
- [17] D. Riehle and T. Gross. Role model based framework design and integration. In *OOPSLA ’98*, pages 117–133, 1998.
- [18] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Lecture Notes in Computer Science 1826, Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.
- [19] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA ’87*, pages 227–242, 1987.