JAC: An Aspect-Based Distributed Dynamic Framework

Renaud Pawlak*, Laurence Duchien**, Gerard Florin*,
Fabrice Legond-Aubry***, Lionel Seinturier***, Laurent Martelli****

*Lab. CEDRIC-CNAM, 292 rue Saint Martin, 75141 Paris Cedex 03, France
** Laboratoire LIFL, Bâtiment M3, 59655 Villeneuve d'Ascq, France
*** Laboratoire LIP6, 4 place Jussieu, 75252 Paris Cedex 05, France
**** AOPSYS, 5 rue Brown Séquard, 75015 Paris, France

December 5, 2002

Abstract

In this paper, we present the JAC (Java Aspect Components) framework to build aspect-oriented distributed applications in Java. This paper goes from the aspect-oriented programming means to the architectural details of the framework implementation. The two core mechanisms depicted to extend the application semantics in order to add well-separated concerns are dynamic wrappers and metamodel annotations. These two mechanisms are dynamically controlled by aspect components. They provide a configuration interface that allows the programmer to adapt and integrate new concerns in running applications. We also focus on the distributed pointcuts mechanism that is one of the more original feature of our framework. Distributed pointcut (which are also dynamic), enable definitions of crosscutting structures that are not necessarily located on a single host.

1 Introduction

Aspect-Oriented Programming (AOP) [KLM+97] is an emerging programming paradigm and philosophy. Its origin goes back to several approaches focusing on Separation of Concerns (SoC) such as Composition Filters [BA01], Aspectual Components [LLM99], Subject-Oriented Programming [HO93, OKH+95, OKK+96], or Multi-Dimensional Separation of Concerns [OT01b, OT01a]. Structured programming and Object-Oriented Programming have introduced a new approach to design programs and a set of guidelines to make code more readable and reusable. AOP goes beyond these approaches and introduces a new style of programming. It addresses issues that can be solved in other approaches, but in a more elegant way and by greatly improving SoC.

Within the current context of the world-wide globalisation of the IT industry, applications may run in open and distributed environments where the underlying structures may be nomad or hardly reliable. In these environments, several concerns such as fault-tolerance, data consistency, remote version updating, run-time maintenance, dynamic server lookup, or scalability must be handled by distributed applications. Concerning all these issues, needs for dynamic or fast reconfiguration of distributed applications are greatly increasing and AO (Aspect Oriented) approaches can provide some answers. Indeed, means to separate concerns, beyond making the applications easier to develop and maintain, also offer some inherent facilities to add or remove concerns on existing applications.

Object-based SoC techniques such as Composition Filters [BA01], MOPs [Sul01] or techniques based on interceptors [OMG02], provide means to separate distribution-related concerns and most of them provide support for dynamic adaptability on a per-object basis. Distribution support is implemented by the reification of the client-server object interaction or communication [AWBB94, FBLL02] but the global modification of a set of distributed objects to introduce a new global and distributed concern requires extra programming efforts. On the other hand, AO languages such as AspectJ [KHH$^+$01] provide powerful syntax extensions (i.e. aspects, advice and joinpoints) to capture crosscutting concerns that allow the programmer to perform global modifications on object sets in a very straightforward way. Distribution and dynamicity of the concerns can be implemented in such languages but require extra implementation efforts.

Finally, if the language approach improves separation of concerns for centralized code, it is sometimes frustrating for the programmer to be unable to apply these concepts within dynamic and distributed environments in a straightforward way (without any extra-implementation efforts). Similarly, object-based or component based SoC techniques hardly capture global and distributed crosscutting concerns within a single syntactic unit and programming concept as in AO languages. Thus, both approaches provide incomplete tools for runtime manipulations of aspects within distributed environment such as dynamic weaving, unweaving, and reconfiguration.

This article provides an insight on the JAC framework (Java Aspect Components) [PSDF01b, PSDF01a, PMSb] that we are developing for Aspect-Oriented Programming in Java. JAC final goal is to provide a set of concepts to enable distributed and dynamic AOP programming. We use ideas coming from object-based approaches and AO languages to reify the pointcut notion so that it can be manipulated in distributed environments and in a dynamic fashion. By using this notion the programmer is able to use AOP to implement dynamically reconfigurable distributed applications.

Section 2 presents the JAC programming model and depicts the semantics of all the available programming concepts. Section 3 proposes practical programming and configuration samples of aspects, including distributed aspects and dynamic reconfiguration. Section 4 gives an overview of the JAC architecture and on the implementation issues for implementing the notions presented in section 2. Finally we conclude and present some directions for future works.

2 The JAC Programming Model

The goal of this section is to present the programming concepts that JAC provides to allow programmers to build dynamic and distributed aspects. JAC is a framework, not a new language. Manipulation of concepts is achieved by extending the classes of the framework and by using existing JAC methods. There are two levels of aspect-oriented programming with JAC:

- The programming level where you can program totally new aspects. At this level, which is the same as AspectJ, programmers create new aspects, new pointcuts, and new wrappers to implement crosscutting concerns. This level is presented in sections 2.1 and 2.2.
- The configuration level (where one can customize existing aspects to make them work with existing applications). This level is supported by a configuration language with a generic syntax that allows the programmer to call configuration methods on existing aspects. In the JAC philosophy, it is very important to understand that you do not need to program aspects to use AO features. We furnish a set of aspects with easy-to-use configuration methods for that. This level is presented in section 2.3.

2.1 Aspect Components

Aspect Components are the central point of our AO framework. These components are hosted by JAC containers [PDF$^+$00]. Much like containers for EJB components, JAC containers are remotely accessible servers. Built on top of a generic distribution layer, these servers come with two personalities: RMI or CORBA. But contrary to EJB containers that only host business components, JAC containers host both business components (i.e. base objects) and aspect components. Much like the components that can be found in other programming models (see for instance the Aspectual Components model [LLM99]), our aspect components are the implementation units that define extra characteristics which crosscut a set of base-objects. The key point of our approach is that these base objects are not necessarily located in a single container. We thus provide a straightforward means to develop distributed applications based on distributed aspects.

2.1.1 Programming aspect components

In JAC, any programmer can define a new aspect component by subclassing the *jac.core.AspectComponent* class. This class provides default primitives to implement technical requirements that influence a set of functional objects in a systematic way, which corresponds to the definition of an aspect in the AOP guidelines [KLM$^+$97]. More specifically, an aspect component programmer can influence the base program in three different ways:

- By extending the base classes' semantics through the definition of some structural meta-informations. In JAC, a runtime meta-model called RTTI (for Run-Time Type Information) is defined for each application. Each item of the RTTI is an element of the base program (e.g. classes, methods, fields, collections). It can be tagged by any aspect component with some meta-data in order to extend its semantics.
- By implementing an internally defined MOP interface (*jac.core.BaseProgramListener*) that allows the aspect components to react on some events occurring within the system (e.g. a system shutdown, an instance garbaging, an application's launching, and so on).
- By constructing pointcuts, i.e. by adding extra treatments before/after/around sets of base-method executions. Contrary to the RTTI annotations that allow structural changes on a per-class basis, pointcuts control the dynamic part of the application.

```
01> public class MyAspect extends AspectComponent {
02>   public MyAspect() {
03>     ClassItem cl=ClassRepository.get().getClass(AClass.class);
04>     cl.getMethod("m():void")
          .setAttribute("synchronized",new Boolean(true));
05>     pointcut("ALL","ALL","ALL",
                   MyWrapper.class,"wrappingMethod",true);
06>   }
07>   public void onExit() {
08>     [...]
09> } }
```

Figure 1: An Aspect Component definition example.

Figure 1 shows a typical definition of an aspect component. As said before, any aspect component extends the *AspectComponent* class (line 1). At construction-time (lines 2-6), the programmer can modify the base program semantics by tagging the classes with attributes or by constructing pointcuts. As shown in lines 3-4, adding a meta-data on a class is achieved through the *ClassRepository* class that is the entry point of the RTTI. Here, we look up a method called *m* and we attach to it an attribute (i.e. a meta-data) *synchronized* set to *true*. All the meta-data that is defined by the aspect components can be retrieved at any time and exploited within the aspect components' deeper implementation to perform extra treatments (here, the entrance of method *m* should be controlled by a monitor). The actual global modification of the base-program code is performed by the pointcut definition of line 6 (pointcut is an inherited method). In JAC, a pointcut installs extra code around sets of methods. The extra code is defined by the

MyWrapper.wrappingMethod method's body. The introductions locations are all the methods (third *ALL*) of all the instances (first *ALL*) of all the classes (second *ALL*). The last parameter (boolean value true) will be explained later in section 2.2.2.

2.1.2 Pointcuts Semantics

A pointcut belongs to an aspect component and can be parameterized to automatically wrap or perform aspect-actions on a set of base-objects according to pointcut expressions. A pointcut expression is a regular expression that can include other keywords or operators (see tables 2 and 3). A given object/class/method/host is extended by the pointcut if:

- the first expression (the object pointcut expression) matches the name of the object (any object hosted by a JAC container is potentially named),
- the second expression (the class pointcut expression) matches the class,
- the third expression (the method pointcut expression) matches the method prototype (for instance, to match a method "void get(int i)", then the expression should be "get(int):void" or any regular/pointcut expression that matches this string),
- the last expression (the host pointcut expression) matches the container name where the owner object is located (this is an optional expression; default is ".*" so that the aspect is applied, by default, on all the hosts of the topology).

Concerning the pointcut expressions, keywords are provided to define method sets. Beyond simplifying pointcut writing, keywords let the pointcut definition be independent from method names and from specific naming conventions. Table 2 shows some of the available keywords when writing a method pointcut expression (other slightly more specific keywords are available in the actual implementation: *GETTERS*, *ADDERS*, and so on). These keywords greatly simplify the expressions so that the use of regular expressions is most of the time not required. Furthermore, pointcut sub-expressions can be composed using the operators given in table 3.

Keyword	Semantics
ALL	all the methods of the matching objects
MODIFIERS[(<name>)]	all the methods that modify the states of the objects if no name is given, else the field
ACCESSORS[(<name>)]	all the methods that access the states of the objects if no name is given, else the field

Table 2: Some keywords examples within the pointcut expressions

Operator	Semantics	Example
||	logical or	**.*:int || MODIFIERS** // matches all the methods that return an integer + all the methods that modify the object state (method expression)
&&	logical and	**MODIFIERS && ACCESSORS** // matches only the methods that modify and also access the object state (method expression)
!	logical not	**package.* && !package.A** // matches all the classes that belong to *package* except *A* (class expression)

Table 3: Allowed operators within the pointcut expressions

At this development stage, pointcut expressions are not rigorously defined in a grammar, this should be achieved in a close future in order to provide enhanced validity checks.

2.2 Dynamic wrappers

One of the key feature of the JAC framework is dynamic wrappers [PDF+00], also called generic advice in relation to the aspect-oriented terminology. A dynamic wrapper is defined as a regular stand-alone object (i.e. it contains fields that form its state and methods that define its functionalities). However, a dynamic wrapper can implement several methods that have special semantics. Generally speaking, a dynamic wrapper is an object that defines behaviours to extend the behaviour of regular objects (we say that the base object is wrapped by a wrapper).

2.2.1 Programming dynamic wrappers

Dynamic wrappers can define three kinds of methods in addition to regular methods that define the functional interface of the wrapper itself:

- *wrapping methods*: they can perform treatments before and after the regular objects methods they are applied to (same as the *around* advice in AspectJ).
- *role methods*: that can extend regular objects interfaces (similarly to the *introduce* statement in AspectJ).
- *exception handlers*: that can handle exceptions that are raised by server objects in the object the wrapper is applied to.

The following code shows a wrapper that contains the three kinds of methods.

```
01> public class MyWrapper extends Wrapper {
02>   public MyWrapper(AspectComponent ac) {super(ac);}
03>   public Object aWrappingMethod(Interaction interaction) {
04>     Object ret;
05>     [...] // before code
06>     proceed(interaction);
07>     [...] // after code
08>     return ret;
09>   }
10>   public Object aRoleMethod(Wrappee wrappee,... /* params */) {
11>     [...]
12>   }
13>   public void anExceptionHandler(Interaction interaction,
14>                                  AnException e) {
15>     [...]
16> } }
```

In JAC, the common joinpoint is the interaction which represents an arriving call to an object. When an object is wrapped by a wrapper (this object is then called a wrappee), some methods can be wrapped by wrapping methods. In this case, an arriving call is intercepted by the wrapping methods that receive the current interaction as a unique parameter (line 3). The interaction contains the wrappee (*Wrappee interaction.wrappee*), the currently called method (*AbstractMethodItem interaction.method*), and the parameters passed to this method (*Object[] interaction.args*). Note that *this* references the wrapper whilst *interaction.wrappee* references the object that is being wrapped. Line 6 asks for the actual realization of the interaction using the *proceed* method defined by the *Wrapper* class. During an interaction, an exception can be raised and caught by wrapper-defined exception handlers (line 13 defines an exception handler for exceptions instance of the *AnException* class). Role methods (line 10) do not take the full interaction but only the wrappee.

The following code shows the implementation of a synchronization wrapping method. It can be used by the pointcut defined in the *mutexclusive* configuration method of the Cool-like aspect (see section 2.3). As one can see, a simple implementation consists in using the *synchronized* keyword of Java on the wrapping method. By this technique, the monitor of the wrapper is used to make all the methods that are wrapped *by the same instance* of the *CoordinatorWrapper* mutually exclusive.

```
public class CoordinatorWrapper extends Wrapper {
  public CoordinatorWrapper(AspectComponent ac) {super(ac);}
  public synchronized Object mutexclusive(Interaction i) {
    return proceed();
  }
}
```

2.2.2 Shared state vs. local state

A wrapper can wrap a unique given object. In this case its state is local to the wrappee: it is a one-to-one relation). But a same wrapper can also wrap several objects. In this case its state is shared by the wrappees: it is a one-to-many relation. Using a one-to-one or a one-to-many relation does not have the same effect on the aspect implementation. For instance, the *CoordinatorWrapper* defined in the previous section should make all the methods that are mutually exclusive share the same wrapper (otherwise, the *synchronized* statement does not work). On the contrary, a wrapper that increments a counter to know how many times a given method has been called should be used with a one-to-one relation. Thus, each method has its own counter.

As showed in section 2.1, the pointcuts of the aspect components handle the wrappers. When creating a new pointcut, one can pass a wrapper instance at the pointcut construction or let the pointcut deal with the wrapper construction. In this case, one just passes the wrapper class name and specifies the *one2one* flag:

- if *one2one* = *false* (default), then the wrapper is constructed only once and shared between all the wrappees – see the left part of figure 2,
- if *one2one* = *true*, then the wrapper is constructed as many times as the pointcut is applied (there is one wrapper per base method) – see the right part of figure 2.

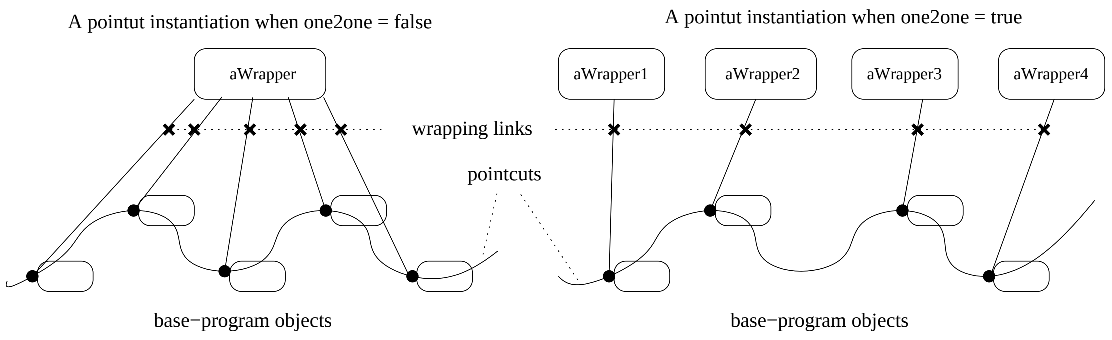

Figure 2: Shared or local states depending on the pointcut one2one flag.

Figure 2 shows the creation of two pointcuts reflecting the two possible uses of the *one2one* flag. On the left, a single wrapper is shared by several objects. On the right, each object is wrapped by one wrapper of the pointcut.

When a dynamic wrapper wraps a base object, the reference of the base object is not changed. As a consequence, their is no need to change the server's reference for a client object (this is called *non-invasive change* which it is easier to implement). A base object can be wrapped by as many wrappers as needed.

2.2.3 Composing dynamic wrappers

There are several advantages about using a standalone object to define an advice (in contrast with AspectJ that defines the advice within the aspects). First, advice can be more easily re-used when programming new aspects (that can be defined independently from the aspects that use them). Furthermore, they can be added or removed at runtime. Since they are neither part of the base object's class, nor the aspect's class, the aspect can instantiate new wrappers or destroy unused one on demand. This allows JAC to perform dynamic weaving. Last but not least, a wrapper is the composition unit when composing several aspects together (when several wrapping methods wrap the same wrappee method). JAC provides a composition aspect that can be configured by the programmer to activate some composition rules. These composition rules specify in which order the wrappers must be applied to the base objects. They can also define some dependencies or incompatibilities between the wrappers.

By using these composition rules, the JAC system can define, for each method and wrapper set, a correct wrapping order that fulfills the final global semantics of the application. This final ordering is called a *wrapping chain*. When the *proceed* method is called in a wrapping method implementation, the system seamlessly calls the next wrapping method if the wrapping chain is not ending. Finally, within the last wrapping method of the chain, the *proceed* method invokes the wrappee method.

Constructing the wrapping chain is beyond the scope of this paper and the interested reader can refer to [PDF99, PSDF01b] for further details.

2.3 DSL definitions

Besides providing a way to program new aspect components, JAC gives users the ability to define their own DSL (Domain-Specific Language) simply by providing a set of configuration methods. As its name states, a DSL provides a set of functions that are very specific to a given scope which corresponds to a particular domain of interest or goal. Within the AOP context, each DSL corresponds to a given concern implemented by an aspect.

```
public class MyAspect extends AspectComponent {
  public void synchro(ClassItem cl,String method) {
    cl.getMethod(method).setAttribute("synchronized",new Boolean(true));
    pointcut("ALL","ALL","ALL",MyWrapper.class,"wrappingMethod",true);
} }
```

Figure 3: An Aspect Component with a configuration method example.

Figure 3 shows the aspect component defined in figure 1 where the modifications performed by the aspect are not implemented at construction time anymore but within a configuration method called *synchro*. This method can be invoked from a configuration file. For instance, the following configuration file makes methods *m1* and *m2* from class *C* synchronized.

```
synchro C m1;
synchro C m2;
```

This file is loaded just after the creation of the aspect and both given calls to method *synchro* are performed. The configuration syntax also enables blocking keywords to factorize sets of configuration method parameters and/or invocations. The aspect should then implement an aspect component method called *getBlockingKeywords()*. Using this feature, it is really easy to rapidly develop DSL. For instance, figure 4 shows how to rapidly specify a Cool-like DSL. Cool is a declarative language dedicated to synchronization and is fully presented in [LK97].

```
public class CoolAC extends AspectComponent {
  public void defcoordinator(String name,ClassItem cl) {...}
  public void selfexclusive(String coordName,String[] methods) {...}
  public void mutexclusive(String coordName,String[] methods) {...}
  public List getBlockingKeywords() {
    return new Vector(new Object[] {
      new BlockingKeyword(
        "coordinator",
        "defcoordinator(String,ClassItem):void")
      }
    );
} }
```

Figure 4: An Aspect Component for a Cool-like DSL.

A typical configuration of this aspect component can be the following (this example is a JAC rewriting of an example taken to present Cool [LK97]).

```
coordinator Blcoord : BookLocator {
  // register and unregister methods from the class BookLocator
  // are synchronized (in the Java terminology)
  selfexclusive {register,unregister};
  // a set of methods from the class BookLocator
  // that cannot be called simultaneously
  mutexclusive {register,unregister,locate};
}
```

Note that this configuration is equivalent to the following one (which is however less straightforward because of the *Blcoord* parameter redundancy).

```
defcoordinator Blcoord BookLocator;
selfexclusive Blcoord {register,unregister};
mutexclusive Blcoord {register,unregister,locate};
```

3 Programming with JAC

The goal of this section is to provide a deeper insight on the JAC programming model by showing concrete programming examples. In section 3.1, we show a wrapper that implements constraint checking. Section 3.2 presents a full application constructed by re-using existing aspect components. Finally, section 3.3 explains how to perform distributed and dynamic AOP in JAC.

3.1 A simple aspect example

Figure 5 shows a core business class that performs simple calculi.

Figure 6 shows how to implement some constraint checking in a very simple way. This example gives some enlightenment on how to use the interaction.

- line 3: the pointcut denotes a set of points of the application where the aspect will change something (note that the wrapper is stateless so the *one2one* flag is set to false),

```
public class Calculus {
  double value=0;
  public void add(double toadd) { value+=toadd; }
  public void sub(double tosub) { value-=tosub; }
}
```

Figure 5: A simple functional class.

```
01> public class TestAC extends AspectComponent {
02>   public TestAC() {
03>     pointcut("Calculus","sub.*",LimiterWrapper.class,"limit",false);
04>   }
05>   public class LimiterWrapper extends Wrapper {
06>     public void limit(Interaction i) {
07>       if( ((Double)getFieldValue(i.wrappee,"value")).doubleValue()
08>           - ((Double)i.args[0]).doubleValue() < 0 )
09>         throw new Exception("<0 forbidden");
10>       proceed(i);
11> } } }
```

Figure 6: A very simple aspect component implementation.

- line 5-6: the wrapper wraps the points defined by the pointcuts with new code. Here, the code is the one defined by the wrapping method *limit*,
- line 7: the actual test. Within a wrapper, we have access to the current interaction information. Here, we know that *i.args[0]* is the parameter of the method that is associated with the pointcut and that *interaction.wrappee* is the calculus object. So we can test them and throw an exception if the *value* field is negative,
- line 10: the test failed, so we can run the *sub* method (thanks to *proceed(i)*).

3.2 Programming an entire application

Besides providing a way to program some customized aspects, JAC comes with a library of ready to use aspects. As a consequence, programming an aspect-oriented application is just a matter of choosing the right aspects and configuring them so that they behave properly for the targeted program.

Each existing aspect corresponds to an aspect component (see section 2.1). Each aspect component is documented so that any final programmer is able to use it just by taking a look at its API (i.e. the set of available configuration methods). Table 4 gives a list of some of the more useful aspect components provided by JAC at the time we write this paper.

Later we will show (for the impatient reader, see figure 13, page 15) the global control flow between all the components of JAC framework for a given application. It goes from the configuration files evaluation process to the actual extension of the base program through the aspects components. For the programmer that just uses existing aspects and configures them by writing configuration files, all the underlying mechanisms being transparent.

To begin with the details of this entire process, we take the example of the *Calculus* class defined in section 3.1 and we add support for 3 aspects: GUI, persistence and authentication. All the used commands are configuration methods defined in the corresponding aspect component interfaces.

aspect name	used to...
tracing	switch on/off some verbose mode on programs, it can also count the methods invocations that occur in the program.
persistence	define which objects of the application will be stored in a database using JDBC
authentication	allow the programmer to restrict the access of some methods to some users
user	handle users and their profiles (often co-used with authentication)
session	make some useful contextual data persistent so that it will remain at the same value for a set of interactions that belong to the same user
gui	define some presentation information
deployment	define how the objects of the application are deployed on the remote JAC servers defined by the topology when JAC runs in a distributed mode
consistency	introduce some protocols to make several remote objects (of the same name) consistent
broadcasting	introduce some protocols to perform broadcasting on remote objects
load-balancing	introduce some protocols to perform load-balancing on remote servers

Table 4: Some useful re-usable aspect components

```
class Calculus {
  // make JAC generate all the resolvable parameter names
  // for the methods that follow the naming conventions
  // (i.e. setter, adders, or removers)
  generateDefaultParameterNames;
  method add {
    // add does not follow the naming conventions, we force
    // the name that the GUI will use
    setParameterNames {"Value to add"};
    // set a default value when add is called
    setDefaultValues {1};
  }
  // same for sub
  method sub {
    setParameterNames {"Value to sub"};
    setDefaultValues {1};
  }
}
```

Figure 7: The gui.acc configuration file.

Figure 7 shows the configuration file for the GUI. The JAC GUI is generic and introspects the application objects to provide default views on them. However, with simple reflection, the GUI is not able to fetch all the information that is needed when constructing the views. For this reason, the GUI aspect furnishes a set of configuration methods that extends the application classes semantics with useful data for the GUI. Typical examples of added information are the parameter names of the methods which are not available with the *java.lang.reflect* package. Hence, the *generateParameterNames* and the *setParameterNames* allow the programmer to indicate these names. As shown in figure 10, these indications are provided by RTTI attributes.

Other types of configuration can be made with the GUI such as the default parameter values when a method is called. Here, the pop-up window that is opened by the GUI when the *add* method is called will propose the ”1” default value to the user, as defined by the *setDefaultValues* configuration method. These are very simple configurations. For more details on all the available configuration methods (such as providing specialized editors or viewers for the fields, changing the default placements in the views, and so on), one can refer to the JAC programming API at [PMSa].

```
// define the users table with their passwords
addTrustedUser "renaud" "renaud";
addTrustedUser "jac" "jac";
// define the methods where authentication must perform
control Calcul {
  restrict add;
  restrict sub;
}
```

Figure 8: The authentication.acc configuration file.

Figure 8 shows the configuration file of the authentication aspect. The authentication provided by JAC is quite simple and would necessitate some enhancements to be fully usable. Still, it allows the programmer to define some users with their passwords and to restrict some methods. The right to call a method is granted only if the *(username, password)* couple matches one defined in the trusted users' list. Again, the RTTI is used by this aspect to denote the restricted methods, as shown in figure 10. This aspect also installs some wrappers on the restricted methods and interacts with the GUI to open a pop-up dialog that requires the user to fill the authentication information.

```
// define Calcul as a persistent root
configureClass Calcul root;
// calcul0 is a name of an object that should never
// change and that should be bound to the corresponding
// stored object
registerStatic calcul0;
// the storage configuration
configureStorage jac.aspects.persistence.PostgresStorage {
    // the database name, the user, and the password
    "calcul", "jac", ""
};
```

Figure 9: The persistence.acc configuration file.

The persistence aspect provided by JAC enables the programmer to make any object or class persistent. A sample use is given in figure 9. A root object is an object that is a persistence root, i.e. all the referenced objects are also persistent. A static name is a name that never changes and that is stored within the persistent storage (here the *PostgresStorage*).

Once all the aspects have been applied and configured, the calculus instance is wrapped by a wrapping chain that extends its behaviour. For instance, the authentication wrapper requires the username and the password before each call of the *add* or *sub* methods. Moreover, the *Calculus* class is extended as shown in figure 10. These extensions are made through the internal structural metamodel of JAC defined in the *jac.core.rtti* package. Each meta item such as a class, a field, or a method can be manipulated and annotated with attributes that refine their semantics. This mechanism can be compared to the annotation or the stereotype mechanism defined in UML.

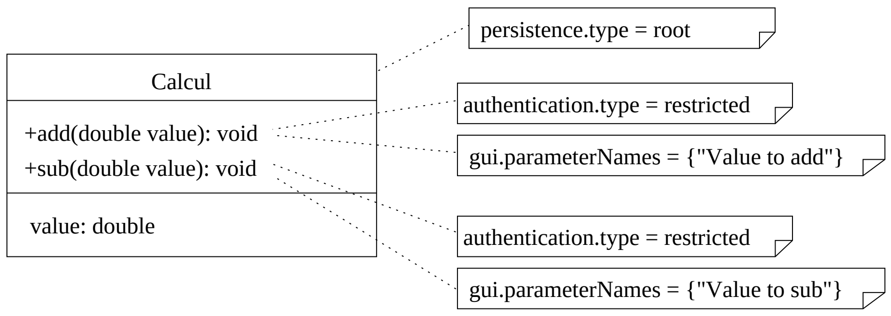
Calcul
+add(double value): void
+sub(double value): void
value: double
persistence.type = root
authentication.type = restricted
gui.parameterNames = {"Value to add"}
authentication.type = restricted
gui.parameterNames = {"Value to sub"}

Figure 10: The *Calculus class extended semantics with the three over-depicted aspects.*

3.3 Creating distributed applications and aspects

Several aspects do not need to be aware of the distribution concern in their definition since they are inherently symmetric regarding distribution (e.g. a tracing aspect). In these cases, the automatic distribution of pointcuts shown in figure 15 is sufficient. However, this is not always the case and some aspects may willingly depend on and even modify the distributed semantics of the application. In the following section, we present a load-balancing aspect that implements a simple load-balancing concern relying on the distribution scheme of the application it is woven to.

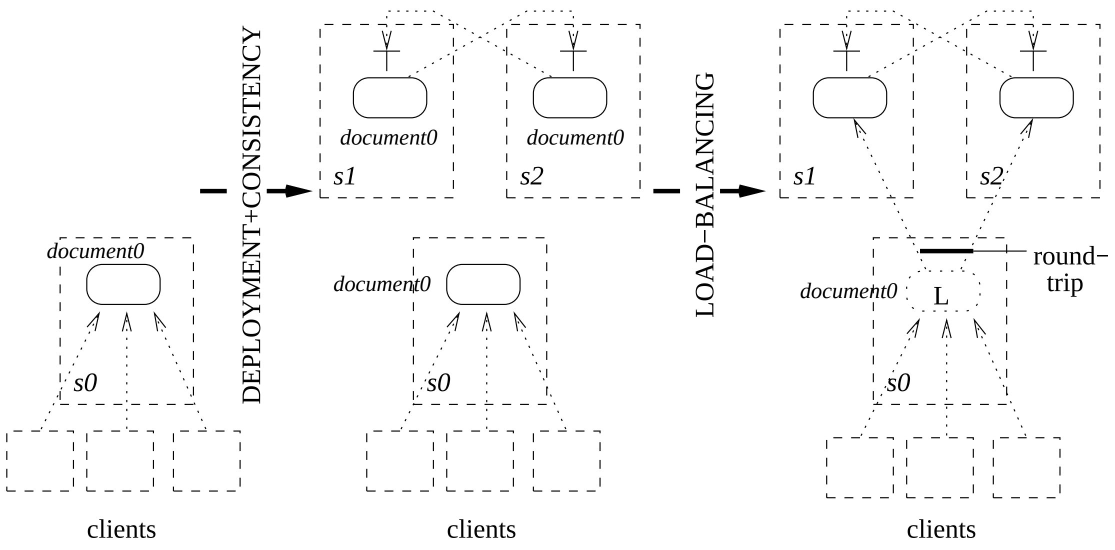

Figure 11: The document server application with a distribution aspect and a load-balancing aspect.

Let us take a client/server application. It consists of a document server object that serves documents to some clients that share this server. The document server is segmented with pages that are indexed from 1 to N and implements a method *int[] searchFor(String s)* returning the indexes of the pages where one or several occurrences of the string *s* were found.

The *searchFor* method can easily overload the server host when the number of pages is high, and thus, it can be very interesting to load-balance this server. As shown in figure 11, this is simple to achieve with the existing aspects of JAC and we there show how to perform a load-balancing server with 2 hosts (the principles would be the same with n hosts).

If one considers an initial centralized server located on *s0* (left part of figure 11), then the idea is to replicate it on sites *s1* and *s2*. These replicas must be in strong consistency so that the set of pages is always the same if we use one replica or another. Note that the initial server is not part of the consistency. By using the JAC furnished aspects, this is done by configuring the deployment and the consistency aspects.

```
// deployment.acc
replicate "document0" ".*[1-2]"
```

Then, to make the two replicated servers consistent:

```
// consistency.acc
addStrongPushConsistency "document0" ".*[1-2]"
```

Note that the second parameter of these rules is a host pointcut expression (see section 2.1.2) that matches all the hosts between *s1* and *s2*. The resulting distributed program is shown in the center part of figure 11. To perform a simple load-balancing protocol on the document server (called *document0*), we can then install a special invocation semantics on the document on *s0* (right part of figure 11). Instead of performing local calls or remote calls to only one remote document, *document0:s0* sequentially calls all the remote documents replicated on the other hosts (with a round-trip algorithm) (see the L stub of figure 11).

Since our pointcuts have distributed capabilities, it is easy to define a load-balancing aspect that performs these requirements. Figure 12 shows the load-balancing aspect that we program especially for this example.

```
public class LoadBalancingAC extends AspectComponent {
  LoadBalancingAC() {
    // note that this pointcut uses an host expression to specify
    // on which host the load-balancing is performed
    pointcut("document0", "Document", ".*",
             LoadBalancingWrapper.class, "loadBalance",
             "s0");
  }
  class LoadBalancingWrapper extends Wrapper {
    int count = 0;
    // this wrapping method actually implements the aspect-
    // method and the round-trip algorithm on the replicas
    public Object loadBalance() {
      // if the replication group is empty we perform a local call
      if( replicaRefs.length == 0 ) return proceed();
      // this test is part of the round-trip
      if( count >= replicaRefs.length ) {
        count = 0;
      }
      // performs the remote call on the currently selected replica
      // and increments the count to round-trip
      return replicaRefs[count++].invoke(method(),args());
    }
}
```

Figure 12: A simple load-balancing aspect.

As one can see, implementing such an aspect is quite easy with our AO model. Of course, this implementation is not generic and is dedicated to this particular example. However, we can easily add some configuration methods that allow the user to parameterize this aspect for his/her own application.

Several other useful aspects can be programmed using the same kind of technique such as fault-tolerance, caching (with a cache proxy), or broadcasting. Several sample implementations are provided in the *jac.aspects.distribution* package of JAC [PMSa].

3.4 Implementing dynamic adaptability with aspects

In section 3.3, we have shown how to program a distributed aspect-oriented application. In this section, we want to explain how dynamic adaptability can be performed.

If we take again the document server example, one can see that a strong consistency protocol has been chosen (in the consistency aspect). However, if the data consistency is not very important, i.e. working documents can be unconsistent, a weak consistency protocol can be used.

```
// consistency-alt.acc
addWeakPullConsistency "document0" ".*[1-2]"
```

By using the *consistency-alt* configuration, the network traffic will be considerably lowered when users change the document's data. However, when the available bandwidth is high, we prefer to use the strong consistency. Thus, a specific agent can be used to swap configurations depending on the available bandwidth. The following code uses the JAC API to implement such an agent.

```
public class ConsistencyPolicySwapper {
  [...] // constants definition, watching thread construction
  int policy=STRONG;
  public void watch() {
    if(Network.availableBandwidth()>50 && policy==WEAK) {
      setPolicy("consistency.acc");
    } else if(Network.availableBandwidth()<=50 && policy==STRONG) {
      setPolicy("consistency-alt.acc");
    }
  }
  void setPolicy(String accName) {
    try {
      Application app=ApplicationRepository.get().getApplication("document");
      // all the client calls wait!
      lock(app);
      // remove the consistency aspect for the document application
      ApplicationRepository.get().unextend("document","consistency");
      // set the new configuration file
      app.getAcConfiguration().setURL(new URL(accName));
      // weave again the consistency
      ApplicationRepository.get().extend("document","consistency");
      // notify the clients
      unlock(app);
    } catch(Exception e) {
      [...]
  }
  [...] // lock and unlock implementation (transactional features)
}
```

4 The JAC Architecture

In sections 2 and 3, we have presented the programming model of JAC and introduced aspect components, pointcuts, and dynamic wrappers semantics. The goal of section 4 is to show how these different elements interact within the JAC architecture as well as to present some of the implementation issues encountered.

4.1 Overview

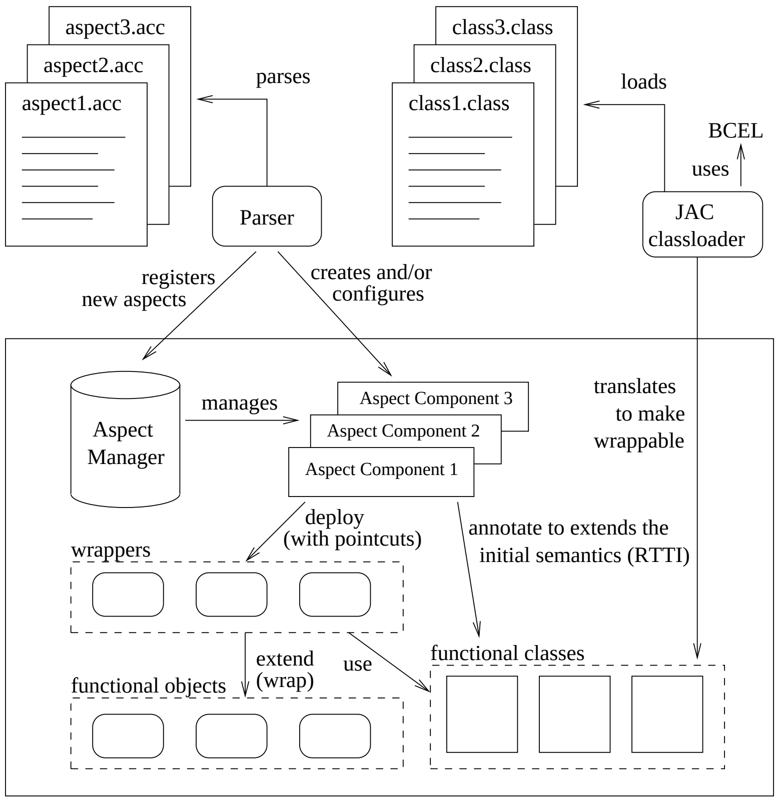

Figure 13: The JAC architecture: global flow of control of a JAC application.

Figure 13 shows how the different JAC system objects interact with the application objects to implement the aspects semantics as depicted in section 2. On the right side of the figure, one can see that the functional classes are modified at the bytecode-level in order to make their instances wrappable. The bytecode translation mechanism is implemented with BCEL [Pro] and is further explained in section 4.2. Once the classes are translated, they are ready to be wrapped by the aspect components. When an aspect is woven to a given application, the JAC system first reads the available aspect component configuration (*.acc files on the upper left of the figure). The parser then invokes a set of configuration methods on the newly instantiated aspect component. These invocations trigger the creation of pointcuts and the tagging of the functional classes with some meta-data (through the RTTI).

When a new instance is created, the AC-Manager (Aspect-Component-Manager) automatically notifies all the registered aspects so that the pointcuts wrap its methods according to the aspect configuration. The important point here is that any aspect component can be woven or unwoven at any time, moreover, its configuration file can be parsed again while the application is running. When an aspect is unwoven or when a new configuration is read, all the added meta-data and the pointcuts are destroyed by the system. This is possible thanks to the wrapping mechanism implementation for dynamic wrapping/unwrapping. The next section shows how the wrapping mechanism is implemented.

4.2 Dynamic wrapping implementation

As said in section 2.2, JAC wrappers can be added or removed at runtime. The goal of this section is to explain and evaluate the underlying mechanisms that are used to implement this feature.

Contrary to regular wrappers that delegate to the wrappee and that implement the same interface as the wrappee [BFJR98, Kni99, BW00] most of the time, dynamic wrappers rely on a Meta-Object Protocol (MOP) [Mae87, KdRB91, Zim96, Sul01] that uses reflection for its implementation. The JAC MOP implementation uses a load-time transformative technique [Chi00] that inserts hooks towards the wrappers. These hooks are reflective invocations so that the actual wrapping method can be resolved at runtime. Let us take a simple *C* class example.

```
public class C {
  public int m1(int i,String s) {
    // m1 body
  }
}
```

Our load-time transformation translates the class C into the following:

```
01> public class C implements Wrappee {
02>   static AbstractMethodItem _JAC_method_0;
03>   static {
04>     _JAC_method_0=ClassRepository.get().getClass("C")
05>       .getMethod("m1(int,String):int"); }
06>   List _JAC_wc_method0;
07>   C() {
08>     _JAC_wc_method_0=Wrapping.getWrappingChain(this,_JAC_method_0);
09>   }
10>   // renamed method
11>   private int _org_m1(int i,String s) {
12>     // m1 body
13>   }
14>   // added stub method
15>   public int m1(int i,String s) {
16>     return ((Integer)Wrapping.nextWrapper(new Interaction(
17>       _JAC_wc_method_0,this,_JAC_method_0,
18>       new Object[]{new Integer(i),s},0)).intValue();
19>   }
20> }
```

The main translation is to rename the original method into an hidden one (here *m1* is translated into *_org_m1* (line 11)) so that the original method is not called directly. Instead, the original method is replaced by a stub method (same prototype, line 15) that wraps the original method by using *Wrapper.nextWrapper(Interaction)* (line 16). The fields *static AbstractMethodItem _JAC_method_0* (line 2) and *Object[][] _JAC_wc_method0* (line 6) are added on optimization purpose. Indeed, the method item (corresponding to a *java.lang.reflect.Method* but in the JAC RTTI) and the wrapping chain (i.e. all the wrappers that currently wrap the wrappee) are quite slow to resolve (lines 4-5 and line 8) so that it is crucial to cache the result in directly accessible fields.

The initial interaction is created in the stub method (lines 16-18). It takes the wrapping chain that is internally used, the wrappee's reference, the current method, the parameters (translated into an array of objects), and an initial rank in the wrapping chain (0). The following code shows a simplified version of the *Wrapping.nextWrapper* method (in real implementation, it also handles statics and constructors).

```
public static Object nextWrapper(Interaction interaction) {
  try {
    // if the current rank is smaller than the wrapping chain's size
    // some wrappers still remain to call
    if (interaction.wrappingChain.size>interaction.rank) {
      // invoke the wrapper of current rank ([0]->wrapper,
      // [1]->wrapping method)
      return ((Method)interaction.wrappingChain.get(interaction.rank)[1])
        .invoke(interaction.wrappingChain.get(interaction.rank)[0],
                new Object[]{interaction});
    } else {
      // invoke the original method
      return ((MethodItem)interaction.method).getOrgMethod()
        .invoke(interaction.wrappee,interaction.args);
    }
  } catch (InvocationTargetException e) {
    [...] // handle exception with exception handlers
  }
}
```

Finally, the *proceed(interaction)* method defined in section 2.2.3 is simply equivalent to *Wrapping.nextWrapper(interaction.rank+=1)*. Moreover, wrapping or unwrapping a given object at runtime is really easy since it consists of adding or removing an element in the method's wrapping chain (that can be retrieved with *Wrapping.getWrappingChain(wrappee,method)*.

4.3 Distribution

Figure 14: The distribution mechanism in JAC: focus on a single object.

JAC fully handles the distribution of aspects. Our motivation for this feature is that, when programming distributed applications, some aspects properties may be required on several containers where the distributed application is running so that the aspect modifications crosscut objects that are not necessarily located on one single container. As simple examples, when applying a tracing aspect to a distributed application, most of the programmers would expect all the calls on all the objects to be traced, whether they are local or remote. In client/server schemes, an authentication aspect should be able to add the authentication concern, on all the clients and all the server containers. Finally, several aspects such as data consistency or load-balancing are inherently distributed and need to be aware of distribution information.

As shown in figure 14, the core distribution mechanism in JAC is based on the application of two kinds of aspect component.

- A deployment aspect component that is used to create a distributed application from a centralized application (step one). The deployment aspect defines a pointcut that wraps the constructors to apply a deployment process (using RMI).
- A set of distributed aspects that implement distributed protocols (step two) which allow the different objects of the application to collaborate between the different containers where the application has been deployed. Since the pointcuts have distributed semantics, it is really simple to define distributed protocols once the application has been deployed.

Figure 15: The distribution of aspects mechanism.

Using aspects authorizes to modularize the deployment issue and the different protocols that are used when programming a distributed application. The protocols implemented by the aspects are of various schemes (consistency protocols, multicasting, load-balancing, fault-tolerance, and so on). Another advantage in using aspects is that the distributed protocols can be added or removed at runtime if the underlying AO middleware handles it (this is the case with JAC). This feature allows the programmer to adapt easily its distributed application to various technical requirements.

The basic mechanism to distribute aspects consists of applying the same technique to the aspect manager of the JAC system. Several distribution strategies are allowed for JAC since the distribution is parameterized into two core system aspects. One of the simpler technique is to replicate the AC-Manager (see section 4.1) and to install a consistency protocol (top part of figure 15) so that the weaving/unweaving of an aspect on one site triggers the weaving of the same aspect on other sites (middle part of figure 15). This way, the semantics of the pointcuts are no more local but distributed as shown on the bottom part of figure 15.

4.4 Performance measurements

The critical point of the JAC framework in terms of performances is the dynamic wrappers invocation mechanism. Since this invocation relies on reflection in order to achieve dynamic adding or aspect removing, the performance overhead of JAC mainly comes from the reflective call's overhead (itself mainly coming from the array of objects construction to pass the parameters).

Type of calls	Number of calls	Total time	Time per call	Overhead
(A) regular object calls	6,000,000	55 ms	~9.16 ns	-
(B) reflective calls	60,000	47 ms	~0.78 μs	(A)x 85
(C) JAC objects calls (0 wrapper)	60,000	61 ms	~1 μs	(A)x 111 or (B)x 1.29
JAC (1 wrapper)	60,000	85 ms	~1.41 μs	(C)+41%
JAC (2 wrappers)	60,000	110 ms	~1.83 μs	(C)+83%
JAC (3 wrappers)	60,000	130 ms	~2.16 μs	(C)+116%

Table 6: Comparative performance measurements for Java and JAC.

Table 6 shows the performances of empty method calls on regular objects and on JAC wrappable objects. These tests are performed with a bench program that calls several methods with different prototypes and that is available in the JAC distribution [PMSb]. The bench program was run under Linux with a Pentium III 600 MHz with 256KB of cache and with the SUN's Java HotSpot Client VM version 1.4.

One can see that a call on a JAC wrappable object is comparable to a reflective call on a regular Java object (with an overhead of 29%). Each time a wrapper is added, an overhead of about 40% of the initial time is added (note that the bench adds empty wrappers that just call *proceed* in their implementations).

Finally, the price to pay for adaptability is quite high (as for reflection) compared to compiled approaches such as AspectJ. However, with real-word aspects and especially when the application is distributed, this cost becomes negligible. For the moment, the JAC approach is thus more suited for middle grained wrappable objects (only business objects are made wrappable in real-word applications, technical components that need performances are not aspectized) and for distributed and adaptable programming.

5 Related works

The JAC approach is widely inspired from the AOP guidelines [KLM+97] and the AspectJ [KHH+01] programming concepts. In fact, JAC can be regarded as a research and implementation effort to apply an AspectJ-like model to dynamic distributed programming. It also intends to provide a powerful configuration level to maximize aspect re-use.

Several projects such as JMangler [KCA01] or Javassist [Chi00] use bytecode-level weaving techniques to achieve AOP without needing the source code . However, they are not specifically dedicated to dynamic adaptation or distributed AOP and thus require some extra implementation efforts.

A relatively close approach is the The Lasagne project [TVJ+01] that provides dynamic weaving/unweaving of aspects that makes them close to the JAC implementation. The dynamicity is achieved by policy selection on the client. The distribution mechanism is provided on regular ORB which makes difficult the actual distribution of the aspects. In fact, the main difference comes from the distributed pointcut notion which is, as far as we know, original to the JAC framework.

Generally speaking, several works handle separation of concerns in object-oriented or component-based environments at a client-server interaction level [AWBB94, HNP98, CBE00, FBLL02]. Most of them provide dynamic adaptability features. However, we think that actual AOP in distributed environments should provide a distributed-pointcut or a similar notion in order to be really effective.

6 Conclusion

This paper introduces JAC, a framework for aspect-oriented programming in Java. We give a synthesis of all the global design that enables dynamic weaving, distribution of aspects, and aspect re-use through configuration. We also gave several indications on how to use the framework and focussed on some implementation points.

The key concept introduced by JAC is the notion of aspect component. An aspect component is the software entity that captures a crosscutting concern. The pointcut mechanism associated with an aspect component allows, like in other approaches such as AspectJ, to express the elements of the base objects where the aspect is to be woven. The originality of our approach is that this mechanism can be applied to distributed objects: i.e. a given crosscut can modify the semantics of objects physically located on distributed hosts. To achieve this, JAC comes with a container mechanism. Our containers host both business objects and aspect component instance. They are remotely accessible using either CORBA or RMI software buses.

Moreover, aspect components can be dynamically (un)woven to the application. This dynamicity enables application adaptability which can be extremely useful within distributed and changing environments.

Besides aspect components, JAC also provides a powerful mechanism for configuring existing or developed aspects. Based on methods defined in aspect components, customized configuration files can be provided for each aspect involved in a given application. The idea is to let the person in charge of software integration express in a declarative way the steps required to configure an application. This mechanism can be related to a Domain Specific Language (DSL) for aspect configuration.

Concerning future works, we are currently investigating the means to overcome the reflection overhead needed by dynamic (un)weaving of aspects (see section 4.2). Tracks can be found in partial evaluation techniques but we will certainly need a modified VM if we want to keep dynamic adaptability properties.

In the close future, we also intend to provide better support for aspect configuration which is, to us, one of the central point of AOP. The challenge here is to re-use existing aspects and, this way, answer the hard problem of component integration. Several configuration languages will be supported in a close future and their grammar should be better defined. Moreover, the pointcut grammar should also be more rigorously defined in order to support compilation and validity checks.

References

[AWBB94] M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting object interactions using composition filters. *Lecture Notes in Computer Science*, 791:152–184, 1994.

[BA01] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[BFJR98] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *Prodeedings of ECOOP'98*, 1998.

[BW00] M. Buchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 201–225. Springer, June 2000.

[CBE00] Constantinos A. Constantinides, Atef Bader, and Tzilla Elrad. Separation of concerns in concurrent software systems. In *ECOOP workshop on Aspect-Oriented Programming*, 2000.

[Chi00] S. Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, June 2000.

[FBLL02] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Comm. ACM*, 45(1):116–122, January 2002.

[HNP98] D. Holmes, J. Noble, and J. Potter. Towards reusable synchronisation for object-oriented. In *ECOOP workshop on Aspect-Oriented Programming*, 1998.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA'93, volume 28 of SIGPLAN Notices*, pages 411–428, octobre 1993.

[KCA01] G. Kniesel, P. Costanza, and M. Austermann. JMangler - A Framework for Load-Time transformation of Java class files. In *IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2001.

[KdRB91] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KHH$^+$01] 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.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[Kni99] G. Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of the ECOOP'99*, volume 1628 of *Lecture Notes in Computer Science*, 1999.

[LK97] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical report, Xerox Palo Alto Research Center, 1997.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, avril 1999.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 147–155. ACM Press, December 1987.

[OKH$^+$95] H. Ossher, K. Kaplan, W. Harrison, A. Matz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA'95*, volume 30 of *Sigplan Notices*, pages 235–250. ACM Press, 1995.

[OKK$^+$96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3), 1996.

[OMG02] OMG. *Common Object Request Broker Architecture 2.6*, February 2002. `http://www.omg.org`.

[OT01a] H. Ossher and P. Tarr. *Multi-dimensional separation of concerns and the hyperspace approach*, chapter Software Architectures and Component Technology: The State of the Art in Research and Practice. In L. Bergmans and M. Aksit, kluwer academic publishers edition, 2001.

[OT01b] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[PDF99] R. Pawlak, L. Duchien, and G. Florin. An automatic aspect weaver with a reflective programming language. In *Proceedings of Reflection'99*, July 1999.

[PDF$^+$00] R. Pawlak, L. Duchien, G. Florin, L. Martelli, and L. Seinturier. Distributed separation with aspect components. In *Proceedings of TOOLS Europe 2000*, June 2000.

[PMSa] R. Pawlak, L. Martelli, and L. Seinturier. The JAC API. `http://jac.aopsys.com/doc/javadoc/`.

[PMSb] R. Pawlak, L. Martelli, and L. Seinturier. The JAC project home page. `http://jac.aopsys.com`.

[Pro] The Jakarta Project. Bcel. `http://jakarta.apache.org/bcel/`.

[PSDF01a] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic wrappers: handling the composition issue with jac. In *Proceedings of TOOLS USA 2001*, 2001.

[PSDF01b] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Proceedings of Reflection 2001*, LNCS 2192, pages 1–21, May 2001.

[Sul01] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[TVJ$^+$01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, Joergensen, and N. Bo. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of ICSE'01*, 2001.

[Zim96] C. Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.