Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France

2005

DHAMACA: AN ASPECT ORIENTED LANGUAGE FOR EXPLICIT DISTRIBUTED PROGRAMMING

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Luis Daniel Benavides Navarro

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Mario Südholt (INRIA-Ecole des Mines de Nantes)

Contents

List of Figures

Abstract

The development of distributed applications with current programming languages is a difficult task. Aspect oriented programming (AOP) promises to provide means for encapsulation of the so-called crosscutting concerns. However few approaches with language support for distribution have been developed.

This thesis introduces Dhamaca, an AOP language with explicit support for distribution. The Dhamaca language is designed as an extension of the Java language. It provides constructs for remote pointcuts (in particular *cflow* and *seq*), distributed advice, aspects and neighborhoods.

Some Examples are presented to show how Dhamaca can be used to write distributed applications. In particular it has been shown how a replicated cache implementation is easily realizable compared to implementations using standard Java platforms, such as JBossCache.

A prototype of Dhamaca, developed by extending JAsCo, is also presented

Chapter 1

Introduction

Modularization of concerns in software applications has been in the center of interest of computer science for a long period, and many approaches have been developed to address this interest. Object-oriented programming has been one of the approaches that has received more attention and resources in last years. Research and industrial experience with those technologies have allowed evaluation of advantages and disadvantages of such technologies.

In particular the so-called *crosscutting concerns* have been found to be application features that cannot be modeled in independent abstraction units of the current programming paradigms, and most of the time the code that manages those concerns is found scattered in the rest of the application code. The following fundamental functionalities, among others, have been identified as crosscutting: security [WVD01], concurrency [KG02], transactions [SLB02, KG02] and distribution [SLB02, MN04]. Aspect oriented programing (AOP) [GK97] is a recent programming paradigm that has been conceived to help developers to modularize crosscutting concerns.

During the last years a large number of tools and languages have been developed for aspect oriented programing in sequential applications, however there are only few approaches to address the development of distributed applications. As the interest in distributed applications is growing thanks to popularization of technologies like Internet, ubiquitous computing and mobile devices, the relation of AOP and distributed programming has become an interesting topic in the AO community. This relation is the main interest of this thesis.

Distributed applications are inherently more complex to develop than sequential centralized applications, in particular, because it is harder to deal with separate processes that communicate each other over the network, with out sharing memory, than working with centralized shared memory systems. When a programmer is designing a distributed application she must address how to partition the application in components over the network, how this components communi-

cate with each other, how they synchronize, how the system will act in case of a failure and all this under the constraint of providing a good performance. Those requirements together with the functional requirements are particularly hard to program because current popular programming languages do not provide means to reason separately about distributed concerns, thus obligating the programmer to write programs where distributed concerns and functional concerns are tangled together in programming units. Crosscutting concerns are those that cannot be modularized in different programming units and then they have to be written together generating the tangling of code, a modern version of the "go to" problem [Lop97].

To address this difficulties different programming practices and different programming languages have been proposed. This work is part of approaches that at the language level can solve the complexity of this crosscutting concerns by providing languages with abstraction components to encapsulate distribution concerns. Making programming less error prone by enforcing coding of these issues by language rules[Lop97]. We consider explicit support, in AOP languages, for distributed concerns to help programmers to write programs that are easier understand, maintain and evolve.

To motivate the research I will consider the implementation of distributed replicated caches. In general it is well known that the performance of distributed applications can be improved by getting data closer to the client with out loosing the reliability, stability and consistency of the system. *Replicated caches* that provide a fast store close to client applications are a common solution to speed up the performance of distributed applications. As a typical example the replicated cache implementation presented by Jboss(tm) in [BW05] achieves a good level of encapsulation for distributed concerns but at the cost of making functional code tangled with mechanism used to enforce such separation (spatial crosscutting), and creating also run time code that is not easy to understand and follow (temporal crosscutting). I will show how using Dhamaca[1], the AO language for explicit distributed programming developed in this thesis, enables writing of simple versions of a replicated cache, and how even sophisticated cache policies, that are not easy to implement with, e.g., JbossCache[BW05], can be easily implemented using *Dhamaca*.

The main contributions of this thesis belong in three groups: first, contributions at the language level; second, contributions at the implementation level; and third, contributions at the application level (language's application).

At the language level, based in the concept of remote pointcuts presented by

[1]Dhamaca is a compound name. D is for distribution, and hamaca is the Spanish word for hammock. In hot weather noting better than having a good rest in a hammock: if programmers use dhamaca for distributed programming they will have the time to do so.

Muga, Chiba and Tatsubori [MN04], the proposed solution introduces three main extensions: a distributed sequence pointcut [DFS04, WV05, AAC+05, DFL+05], support for distributed advice and aspects[MN04, PSD+04], and a notion for explicit neighborhoods.

At the implementation level a prototype of Dhamaca has been developed by extending JAsCo [SVJ03]. The implemented version , called DJAsCo, includes the following contributions: distribution of aspects by means of distribution of connectors using Jgroups [Ban05], distributed support for sequences and stateful distributed aspects extending the work presented in [SVJ03, VS], and distributed advice implementation.

At the application level, different examples of practical applications of the language are presented: first an example of implementing a replicated distributed cache as the one presented in [BW05]; then, an example of implementation of sophisticated, and expressive, cache policies that can not be achieved cleanly with JbossCache; also an example of implementing distribution in a non-distributed application using the context modification allowed by the *proceed* construct; and finally an example of testing architectural constrains in applications.

Thesis structure. The remainder of this document is structured as follows. Section 2 presents the related work, by discussing several approaches on the language and implementation levels. Section 3 presents our AO language for explicit distributed programming. Section 4 presents details of the implementation of DJAsCo. And section 5 concludes.

Chapter 2

State of the art

This chapter presents an overview of research work related to distributed AOP languages. First, The problem of distributed crosscutting concerns, along with approaches to address it is introduced. Followed by a discussion about history-based AOP approaches.

2.1 The problem: crosscutting concerns in distributed applications

In order to create distributed applications a programmer has to deal with concerns found on sequential centralized applications (e.g. functionality, security, logging, persistence) plus concerns for distribution. In particular a programmer has to deal with partitioning the functional components through the network, coordination of such components and handling of partial failures. These concerns are considered crosscutting concerns because they cannot be encapsulated and modeled in units of abstractions of popular programming languages like Java, Smalltalk, C or C++; thus, code developed in such languages and related to distribution concerns became tangled with other concerns' code. This section present related work that address problem of code tangling of crosscutting concerns in distributed applications, showing at the end an actual example of the problem.

2.1.1 General research on distributed crosscutting concerns

The phenomenon of tangling of code related with distribution has already been addressed in literature. Lopes showed in [Lop97] how ,using Java as developing language, the simple inclusion of concurrency and distribution generated a very complicated tangled code. Her work also showed how the tangling phenomenon could be reduced. First, by enforcing programming practices generating patterns

and rules that should be followed when programming; and second, by providing programming languages with specific support for implementing such concerns. It was clear that the first approach presented some level of success but with side effects , like generating patterns' specific tangled code that also polluted the functional code making it not easy to understand and again tangled. The second approach proved to produce programs more reliable and easy to maintain.

Concurrency and failures are presented as crosscutting concerns by Kienzle and Guerraoui [KG02], in their work is stated that full aspectization of these considerations is not possible. To show this, an experiment using AspectJ and transactions as the mechanism to handle concurrency and failures is presented. The experiment was divided in three parts. The first part applied transactions to an already existing application using AOP, however, the solution presented was limited, according to the authors, because the incompatibility between linearizability of methods invocations on shared objects [HW90] and the serializability of transactions [Pap79]; it was also limited because impossibility to identify irreversible actions (E.g. I/O actions). The next step in the experiment was to aspectize the transaction interface (begin, commit and rollback), but in the presented solution exception handling is not aspectized in part because authors argue that in some situations the functional and concurrent aspects are connected and they can not be separated completely. The last part of the example tries to aspectize the transactions mechanism, in this case the authors present a syntactical separation, but they argue that this separation requires more expertise and knowledge than the standard way of writing transactions.

Soares et al. [SLB02] reports the experience of converting an specific distributed application to a non distributed version an then incorporating the distributed concern using AspectJ. The exercise shows an interesting example of applicability of aspects for implementing distribution, however also shows how difficult is using AOP for distribution with current approaches, in the sense that the utilization of current AspectJ approaches presents the need of having a high degree of knowledge of the middleware mechanism (RMI).

2.1.2 Motivating example: policies for distributed caching

To motivate our investigation and to show that tangling is an actual issue in programs developed with current programming languages I will analyze the tangling phenomenon for the caching subsystem of a well-known platform for distributed application development, JbossCache [BW05]. Furthermore, this example will be used later to illustrate the approach of my thesis.

Jboss Cache Overview. JbossCache is a cache implementation which data is stored in a tree data structure, it supports transactions, replication, eviction policies and automatic access to backend data stores. In this implementation each

node in the tree has one parent, and can have many children, it also has an associated hash table with pairs key-value. The tree structure was preferred over the hash table structure because a tree can be used as a hash table as well as a tree and it gives support for mapping objects in a more fine grained way.

The implementation of the cache is based on a simple tree class that implements a tree data structure, the name of this class is *org.jboss.cache.TreeCache.* This main class is augmented with code to support a chain of interceptors, separating crosscutting concerns in classes called interceptors. Examples of those classes are *org.jboss.cache.interceptors.Lockinterceptor* and

org.jboss.cache.interceptors.ReplicationInterceptor. Each method invocation to a *TreeCache* object is then processed by all the elements of the interceptor chain.

Even though the design and coding rules are applied with care, the generated *TreeCache* class has tangled code that address distribution, concurrency and failure safety as well as tangled code for the interceptor filter pattern (all within a source file of 1741 LOC, with 126 methods and 27 imports). The source file representing the tree data structure is hard understand and it's original behavior is hidden behind the code that implements the chain of interceptors mechanism. Furthermore the interceptor Filter pattern implemented creates a solution that at runtime generates time tangling making it hard to determine exactly what code and when was ran. This file is a clear example of spatial and time tangling enforced by programming practices (e.g. interceptors chain) and by language mechanisms that, as Java does, provides synchronization mechanism orthogonal to the OO paradigm, as stated in [Lop97].

JbossCache uses a replicated cache policy, where all data of caches is replicated. Solutions to achieve more flexible policies using JBossCache leads to crosscutting code. To achieve such policies, with standard JBossCache, complex cache hierarchies have to be combined with fine grained eviction policies. For example a variation of the JBossCache standard policy where synchronization behavior with those caches that hold a copy of the objects that have been explicitly put in the cache by the local process and only over the specific object, is only achievable in limited versions.

2.2 Aspectizing distribution concerns with AOP languages and frameworks for distribution

Language or framework support for distribution concerns have been addressed in different ways by AOP research (or AOP related research). This section shows different approaches to address crosscutting concerns in the scene of distributed ap-

plications. Remote pointcuts provide an interesting language construct designed to identify join points in programs running in remote hosts. Distribution through automatic partitioning of applications, presents ideas for frameworks that can have explicit configuration of how components are deployed and coordinated in different hosts, even though they do not provide explicit language extension this solution present a mechanism for separating distribution concerns . Finally, aggressive encapsulation of distribution and concurrency section presents fundamental ideas of separation of distribution concerns. These different ideas will provide design guidance to Dhamaca.

2.2.1 Remote pointcuts

Remote pointcuts were introduced into AOP like language/framework constructs to detect join points executed in different execution spaces (I refer to execution spaces to indicate processes that do not share memory but not necessarily run in different physical machines). Work presented in this subsection address related work in AOP area that deal with remote pointcut mechanism as a concept. The approaches presented show differences between supporting the remote pointcuts by extending the API of base language or by augmenting the language.

Remote pointcuts were addressed by Pawlak et al. [PSD+04] and later by Nishizawa et al.[MN04]. The work by Pawlak presents JAC a dynamic AO framework for distribution where the concept of distributed pointcuts is introduced. A remote pointcut construct was developed extending the Java API, the developed construct was a method that could include the specification of a named host, allowing the programmer to explicitly decide in what container a joint point should be detected. As mentioned here JAC is provided as an API extension, and the aspect language and the pointcut language are implemented using OO abstractions provided by Java, thus it does not provide new specific language constructs. When dealing with distribution this work also address the problem of aspect distribution, in the sense of how an aspect is deployed to various containers, the solution shown in the paper is the replication of an aspect manager on each node and the implementation of a consistency protocol between them to communicate the events of deployment and undeployment of an aspect (JAC implements dynamic weaving of aspects at the byte code level).

The work presented by Nishizawa et al.[MN04] address, in a more explicit way, remote pointcuts, presenting DJcutter,an extension of AspectJ, as a language for modularizing distributed concerns. Their work extend a subset of AspectJ's constructs to make them behave as remote pointcuts and it also provides new language constructs to reason about remote join points. A pointcut designator introduced by DJcutter is *hosts(hostid)*, this language element basically match all the join points found in a specific host. The system also make an extension of

cflow. However, such cflow extension is constrained because it requires the communication, of the base application, to be implemented using an specific custom socket implementation. It is important to note that join points' state can be exposed to advice, by means of value binding to pointcut parameters, using *args* and *target*. By default these parameters are sent to the aspect server by copy, but the system can be configured to pass them as remote references. Information of join points can be also gathered using the reflection capabilities of the system. A final feature provided with DJcutter is the possibility of telling an aspect to execute locally, thus the aspect and the advice are distributed but once a host find a join point it executes the advice in the local machine with out sharing information. DJcutter uses an Aspect Server where the advice are run, then you only distribute the remote pointcuts definitions and once they are activated in any host the aspect server will run the advice body. The thread that invokes the pointcut is suspended until the server finishes the execution of the advice. As we mentioned before the weaving is executed at load time on each server.

2.2.2 Related approaches for distribution

Distribution trough automatic partitioning of applications Many research works have been proposed on automatic partition as a mechanism to incorporate distribution concerns to a non distributed application with out modifying the original code. Between those studies, I am interested in work that address Java AOP implementation techniques. This interest is motivated by two reasons: first, they present an example of a mechanism for separating distribution concerns; and second, because they address interesting implementation issues found when dealing with distribution, like the problem of implementing remote references, and how those references have to be treated in different manner depending of nature of the object being referenced.

Aridor et al. [AFT99] apply the concept above by modifying the virtual machine making it aware of the of the cluster to create a distributed virtual machine image. Other approaches like Addistant[TSCI01] and J-Orchestra[TS02] implement the automatic partitioning by modifying the bytecode of the application to implement the distributed virtual machine image. These approaches document the problem of how remote references to objects are different depending of the nature of the object. In particular, they present different proposals of how unmodifiable code (e.g. binary optimized code found in some core classes of Java) is treated. Thus, it is not possible to modify this binary optimized code to introduce remote proxy indirections, so the automatic partitioning of code, including classes with that kind of code, must be addressed with special care.

As stated above, one of the interesting implementation issues introduced by the mentioned related work is how remote references are treated, depending of

where the remote reference is introduced and what kind of objects are referenced. A full description of all the approaches is out of the scope of this document, because of that I am going to detail how proxies were used in [TSCI01], this selection is made because I consider their approach more general an detailed. The model used is to implement proxies for remote references, so when a remote reference is needed a proxy is used and the proxy hides all the complexity of the communication. Based in this model we find that different types of proxies can be used depending of the code that wants to be modified (in the case of byte code modification). The first case, is a proxy that replaces the original object so all the references will be through the proxy, and it must be use when all the references must be remote. The second case, is the one where a proxy is generated and the code of the objects that reference the master object are modified only if they are suppose to hold a remote reference of the original object. The third case, is the subclass approach, where a subclass of the master object is created and depending if the reference to the object is local or remote the system will use the original object or the subclass version. The fourth case, is when no remote reference of an object must exist in this case any attempt to have a remote reference will cause creation of a new copy of the object, that will be only referenced locally.[1]

The work presented in [TS02] introduce an interesting implementation to manage unmodifiable code, basically this works shows that when two unmodifiable objects reference each other is impossible to introduce indirect references, but a modifiable object that references an unmodifiable object can be change and be placed in different host. In order to achieve a clean implementation the proxies must be aware of when a reference is passing from a modifiable scope to an unmodifiable one so the right transformation can be applied to parameters. This rewrite particularity give the possibility of having object mobility, meaning that objects can move at runtime from one server to other in a particular deployment topology.

From this series of works it is important to extract some notes in a distributed thread model that is addressed in [AFT99]. This interest is driven by the necessity of finding optimized solutions to the cflow construct proposed in [MN04]. The approach presented requires modification of the base code to introduce a custom implementation of socket classes. Such implementation basically propose migration of the thread stack information in each remote call. A clear improvement, at least in network load, will be to evaluate the cflow information in a distributed way. In other words maintaining a distributed stack like the one proposed in [AFT99], and accessing it only when necessary. Note that the information in the stack is not copied, instead of that each node holds a part of the full stack information.

[1] Remember that the distribution is not explicit in the program, in general the distribution will be configured externally.

Code mobility I want to broaden this discussion by including some material about code mobility, because this concept is used in a limited form in the context of distributed advice in Dhamaca. Many languages with support for code mobility have been proposed , such languages can be classified into those that provide *strong mobility* and those that provide *weak mobility* as presented in [CGPV97]. Strong mobility refers to the capacity of moving execution units with their code and context between execution spaces. Weak mobility refers to allow a execution unit in a execution space to be bound dynamically to code coming from a different execution space. In Dhamaca's distributed advice we will use a version of weak mobility. Some recent work related to mobility is studying how prototype-based object models can help create better languages for mobile devices [JD05b, JD05a]. This approach is of particular interest because it can simplify the concepts needed to think about parameter passing (i.e. the relation class object is not a problem any more in prototype based languages).

2.2.3 Encapsulation of distribution and concurrency

Syntactical separation of concerns is one of the points addressed by AOP approaches. To achieve such separation in distributed applications different AOP approaches have been proposed. In this section I now discuss approaches which treat concurrency as a particularly important concern of distributed systems.

One of the first AOP approaches that address distribution concerns was D [LK97], this language basically introduce a framework of three languages designed to provide encapsulation of distribution concurrency and functional concerns. The functional language JCore is a traditional OO language for writing objects, that in this case is a subset of Java language. COOL the coordination language have directives to describe relations between methods, such relation can tell if a method is auto-exclusive (only one thread at the time) or if two methods or more are mutually exclusive. RIDL, the remote interface aspect language, allows the programmer to specify how information is sent and what information is sent when remote method invocation happens between different execution spaces. It provides control over the copying semantics, allowing the programmer to restrict the amount of the object graph that has to be copied when a parameter is pass by copy.

Other example of work developed with the concept of aggressive encapsulation of distributions concerns, concurrency in this case, is the one presented in *Sequential Object Monitors* (SOM)[CMT04] as an alternative to Java Monitors. One of the criteria used in the design of SOM was modularity, meaning that synchronization code should be specified separately from application code, in order to achieve a clean separation of concerns, and making it easy to "plug" synchronization onto existing, not thread-safe, classes. Based on this criteria, the authors

present a sequential Object Monitor, that is a standard object to which a low-cost (thread-less) scheduler is attached.

2.3 Expressiveness through history based pointcuts

AspectJ's pointcut language provides means to take actions based in the current state of the computation, with the notably exception of *cflow* construct that allows the programmer to take actions based on the relation of join points found in the same control flow. Proposal to extend this conception allowing the application behavior to be modified depending on the events found in the history of the computation have been proposed in [RD01, AAC+05, WV04, RD02, DFL+05]. Such approaches provide definitions for atomics elements that can be detected in the computation trace (e.g. events), they also provide a language to express relations between those atomic elements and to extract values form such expressions, and finally the provided a mechanism to attach behavior to such expressions in order to be executed when the specific event or pattern of events is detected. Dhamaca will be consider constructs to support explicitly event based AOP in distributed environments. This section presents a discussion of different semantics introduced in the context of non-distributed event based AOP approaches. Support for those semantics is considered and addressed in dhmaca's design.

Pointcut languages for history-based approaches. Different comparison can be made from the different pointcut languages proposed to detect event patterns in different approaches. Expressiveness is one of such characteristics, based on this it could be said that some approaches present pointcut languages based on regular grammars (e.g. regular expressions) as presented in [AAC+05]; others present pointcut languages based on context free grammars [WV04], making them more expressive (e.g. support for nested event patterns); and finally other approaches presents pointcut languages that are Turing complete (e.g. work by Douence et al. [DFS04, DT04]) making them even more expressive. Other, more informal, comparison that can be made, is based in how aspect language syntax is modified. Having some approaches that extend the pointcut language to support constructs to declare event patterns as presented in [WV05]; other approach is to have composition of aspects as that presented in [RD02, DFS04]; and finally a new language abstraction (e.g. aspect, class) that encapsulates the pattern expression as provided by [AAC+05, WV04]. All these approaches present different ways to express the relation between events, and each of them present advantages to express different features in an easier way, by example with the relation between aspects you can avoid the complexity of thinking in the instantiation problem in sequences, because the sequence is something external to the aspect. Support for event pattern detection presents an implicit mechanism to allow the flow of information (e.g.

past event execution).

Variable binding. Other way to pass information is to define a mechanism for explicitly sending information at the pointcut level. Douence et al. make a formal presentation of an inter-crosscut pattern variables mechanism in [DFS04], with this approach it is possible to bound values to variables, in order to be used in the definition of future pointcuts. Allowing them to express dynamic pointcut expressions where pointcuts are modified depending in the variables bound by past events. This mechanism is implemented in *tracematches*[AAC+05], extending Douence et al. work, by letting binding mechanism to determine the presence of "multiple parallel object specific traces" inside a tracematch construct. Those parallel per object specific traces are matched individually depending on the values bounded to variables (e.g. an event match the trace if it bounds the same set of values to the same set of variables). This mechanism address in implicit way the problem of sequence construct instantiation. Note that this definition can give to expected/unexpected behavior by branching.

Sequences and event patterns. As mentioned before these languages modify the behavior of a program by allowing the programmer to reason about the history trace of events in the computation, one of the common constructors encountered is the sequence of events that basically define a sequential pattern. Such constructors have been extended introducing more sophisticated means like detection of events regular expressions [WV04, AAC+05]; or introducing constructs to compose aspects, such constructs allows the programmer to create parallel execution, recursive composition and selection of aspects as proposed in [RD02, DFL+05]. One interesting issue that appears when sequences are introduced is the concept of when and how instantiation happens, as mentioned before one a way to do it is to define instantiation in terms of variable binding as presented in[AAC+05]; other way is to make instantiation explicit, in such a case you can have such instantiation by attaching an instance to an aspect instance[AAC+05, WV05] or by providing external constructs that relate aspects (Aspects and sequences at the same level)[DFL+05]. The semantics of the sequence instantiation introduced in JAsCo stateful aspects is simply an single instance attached to an aspect instance that is enough to create a language extension that allows to create finite state machines. This subject will be discussed in detail in section 4.1.

The advice language. Finally each language provide mechanism to extend behavior in the interesting points, once they are detected. Such mechanism can modify global values and local values and they also can use the values that are bound by the pointcuts in the advice signature variables.

2.4 Summary and conclusions

In this section I present the problem of tangling of crosscutting concerns in distributed applications. Showing, that writing distributed applications is tough, and is even more difficult is to maintain and debug distributed applications created with popular programming languages. An actual example of tangled code in distributed applications is showed based in JbossCache source code. It is also showed that the problem of code tangling can be lessened by improving programming practices or improving language support for encapsulating distribution concerns.

Improving language support for encapsulating distribution concerns in AOP presents an open space for research. This due to the fact that most of the current AOP research approaches are targeted to sequential applications and not to distributed applications. Dhamaca is an AOP language with explicit support for distribution concerns an approach that takes advantage of this open space for research.

Dhamaca design is driven by ideas provided in different approaches that were discussed in this section. From the information analyzed the design directives that were extracted for my designs are:

- Dhamaca will have support for remote pointcuts. Presenting explicit constructs at the pointcut language level.
- It will also present support for detecting patterns of events in the distributed computation trace. Presenting difference semantics for the event pattern detection constructs.
- It will present constructs to have distributed advice support. Allowing the programmer to reason explicitly about in what execution space an advice should run.
- It will promote encapsulation of distribution concerns in aspects.

Chapter 3

Explicit programming with the Dhamaca language

This chapter presents Dhamaca, an AOP language with explicit support for distribution. I present also a motivating example to introduce the language, followed by a definition of the syntax and informal semantics, and finally a detailed presentation of diverse application examples.

3.1 A motivating example: cache replication

A common practice in enterprise application design is to separate the application architecture in multiple tiers, examples of such architecture are enterprise transactional web sites, such systems can have a servers that provides presentation services to clients and communicate with applications servers that provides business services that at the same time are communicated with robust backends [Jac03]. This kind of architecture is used to allow insertion of firewalls between layers (i.e to increase security of data expose to clients) and to create clusters. Clusters are created to improve reliability and performance of the whole application. This is achieved by means of distributing client requests to different members of the cluster, and assuring that if a cluster's node crash, clients' requests will be handled by the remaining active members of the cluster. To improve performance in distributed applications it is also commonly used to realize caches [GJL05, Bor01], i.e., data stores which are close and therefore rapidly accessible from clients. It is also known that the inclusion of synchronization between those caches, one in each different node in the cluster, can be useful in enterprise applications to improve reliability[BW05, Bor01]. Figure 3.1 shows an instantiation of an architecture described above, where the presentation layer is supported by a cluster of servers that interact with another cluster of servers in the application layer that

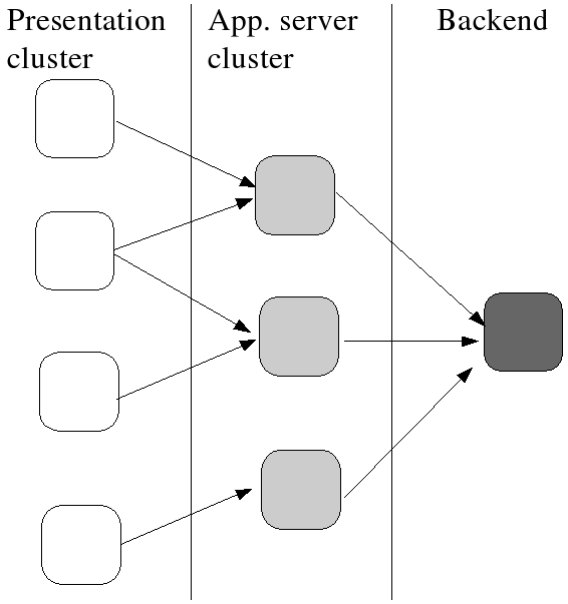


Figure 3.1: Multitier cluster architecture

finally interacts with a robust backend. The arrows show a typical scenario of communication but not the only one. As mentioned above over such architectures is possible to dramatically increase performance by introducing caches. The reliability of such applications can be improved by the introduction of replication between those caches [BW05, Bor01].

As analyzed before in section 2.1.2 on page 12, the implementation of a replicated cache solution is per se an example of a distributed application, that suffers the problem of spatial tangling when implemented and the problem of time tangling when deployed in a cluster environment as the one described above. This problem is even worse when the deployment of the cache implementation is considered in different layers of the application (e.g. between the presentation servers and application server; and between application server and backed servers).

To introduce Dhamaca we will consider an example where we want to implement a general solution for a replicated cache similar to the ones proposed in [BW05, Bor01]. Listing 3.1presents a simple version of an cache class implementing a simple caching mechanism using a *Hashtable*. It exposes two methods, *get* to get information from the cache and *put* that puts information in the cache. This is a plain version that does not includes any replication code. In order to extend the behavior of this solution to a replicated one, we introduce a replication aspect as presented in Listing 3.2, this simple aspect introduces replication to the solution. Line number 2 present the header of the aspect that indicates that the

```
1
2  import java.util.*;
3
4  Class Cache{
5
6         private static Cache _instance= new Cache();
7
8         public Hashtable cache = new HashTable();
9
10        public Cache(){}
11
12        public static Cache getInstance(){
13
14                return _instance;
15
16        }
17
18        public Object get(String s){
19
20                User u = cache.get("/user/"+ s);
21
22                }
23
24                return u;
25
26        }
27
28        public void put(String userId, User u){
29
30                cache.put("/user/"+ s, u);
31
32        }
33
34 }
```

Listing 3.1: Cache with simple cache implementation example.

```
1
2  global aspect CacheReplication{
3
4     pointcut cachePcut(Object key, Object o): call(* Cache.put(Object,Object))
5
6          && args(key,o) && !host(localhost) && !on(jphost);
7
8     before(Object key, Object o): cachePcut(key,o){
9
10      Cache.getInstance.cache.put(key, o);
11
12    }
13
14 }
```

Listing 3.2: Replicated cache aspect example.

aspect will be distributed globally and it will have an singleton instance on each host, thus controlling instantiation and distributed deployment conditions. Lines 4 to 8 show the pointcut language, the *call* pointcut picks out the join points in the *Cache* class when the put method is called; the *args* pointcut binds the values of the arguments passed in the method invocation to the arguments of the pointcut; the *!host(localhost)* pointcut matches the join points that originated in hosts different from the one where the aspect is deployed; the *!on(jphost)* pointcut matches all the joinpoints that are intercepted by an aspect that is not deployed in the host where the join point was executed, so this joinpoint controls whether an advice is executed or not depending on the host where the advice is deployed. Note that the *host* pointcut let the programmer to express restrictions on the host where the join point is originated; the *on* construct let the programmer to express restrictions on the host where the advice is executed; and finally there are two constants introduced to facilitate writing of aspects with out knowing the distribution technicals details like the ip address and port of the node, these constants are *localhost* that is the host where the aspect is deployed and *jphost* that represents the host where the join point, under evaluation, occurred. Lines 8-12 shows a simple advice code that simply put in the cache the element, performing the actual replication.

As a summary, the aspect is distributed to all the host participating in the solution, and it is instantiated creating a singleton instance on each host. Then when an invocation of the *put* method in a Cache class occurs, the appropriate join point is distributed to all the participating hosts where it is evaluated individually by each aspect instance. In such an evaluation the join point is matched in all the aspects that are not deployed in the host where the join point was originated.

The above example presents the implementation, using Dhamaca, of an actual

distributed application. This implementation ,however , represent a little set of the distribution policies that need to be implemented in a distributed cache solution. To achieve more expressive policies in a cache solution it is necessary to have more expressive constructs to reason about distribution concerns. As shown in section 3.5.1, to achieve more sophisticated policies it is necessary to support sequence like constructs (e.g. like those introduced in 2.3).

Finally, the example above present improvements over previous work, like the one presented by Chiba et al.[MN04], by adding the concept of explicit control over distributed advice execution, also presenting the possibility of having multiple parallel advice execution in specific hosts. Another feature introduced by Dhamaca is the possibility to reason about the deployment behavior of aspects, meaning that the programmer can control where aspects are deployed.

3.2 Syntax and informal semantics

This section introduces the informal semantics of the language. Dhamaca is an extension of the Java language that uses some of the constructs an ideas introduced by AspectJ[GKG01], extending such approaches by adding a remote join point model[MN04, PSD+04] and sequences to provide expressive constructs in order to have explicit distribution support. It also introduce an explicit notion of neighborhood.

3.2.1 Join points, neighborhoods and nodes

Dhamaca's join point model consist of method calls only. The semantic of method calls are modified by the introduction of the concept of groups of nodes. A node is any runtime instance of Dhamaca processes (programs written in Dhamaca). More than one node can be running in a machine. A group of such nodes will be called a neighborhood, in fact in a deployed distributed application it can be many neighborhoods. Note that a node can also belong to many neighborhoods. These groups are created by runtime instances of Dhamaca processes (nodes), such instances can join and leave the groups at any time.

Dhamaca's joinpoints are augmented with the information of node and neighborhoods to which the node belongs, and those augmented joinpoints are distributed as messages to all the members of application when they occur. This constitute the distributed joinpoint model of Dhamaca, and will be the base for building remote pointcuts.

3.2.2 Pointcuts

This section introduces the pointcut language definition, as mentioned before the constructs provided by dhamaca are remote pointcuts. Remote pointcuts are language constructs for identifying joinpoints in the execution of program in a remote node. Figure 3.2 shows the pointcut language.

$$
\begin{array}{lcl}
P & ::= & PPrim \mid PSeq \mid PDist \mid PNeigh \\
 & \mid & P \,||\, P \mid P \,\&\&\, P \mid !P \\
 & & \\
PPrim & ::= & call(Signature) \mid target(TypeIdExpr) \mid args(\{TypeIdExpr\}) \\
 & \mid & cflow(P) \\
 & & \\
typeIdExpr & ::= & JavaType \mid Identifier \\
 & & \\
PDist & ::= & host(HExpr) \\
 & \mid & on(HExpr) \\
 & & \\
PSeq & ::= & seq(\{PSeqStep\}) \\
 & \mid & step(PSeq, stepId) \\
 & & \\
PNeigh & ::= & neighborhood(NExpr) \\
 & \mid & onneighborhood(NExpr) \\
 & & \\
HExpr & ::= & localhost \mid jphost \mid \text{"}ipAddr : port\text{"} \mid Identifier \\
 & & \\
PSeqStep & ::= & [Identifier :] P \,[>> NextStateExpr] \,[RegExprCard] \\
 & & \\
RegExprCard & ::= & + \mid * \mid \{n, m\} \\
 & & \\
NextStateExpr & ::= & StepId \\
 & \mid & StepId || NextStateExpr \\
 & & \\
StepId & ::= & Identifier | Int \\
 & & \\
Identifier & ::= & //Java\ valid\ identifier \\
JavaType & ::= & //Java\ valid\ type\ expression \\
Int, ipAdd, port & ::= & //integer\ expression \\
NExpr & ::= & //A\ valid\ Java\ string\ expression
\end{array}
$$

Figure 3.2: Pointcut language

Pointcuts are constructed from primitive pointcuts *PPrim*, sequence pointcut *PSeq*, distribution pointcut *PDist*, neighborhood pointcut *PNeigh*, or any logical composition of a pointcut expression using *||(or)*, *&&(and)* and *!(negation)* operators.

Primitive pointcuts *PPrim*, are those introduced in AspectJ and also present in Dhamaca: *call*, *target*, *args* and *cflow.* These constructs present the same behavior as the constructs of the same name found in AspectJ[GKG01].

Pointcut *call(signature)* matches calls to methods with matching signatures *signatures*, where *signature* is a method signature pattern as those defined in AspectJ. It is important to remember that each pointcut construct will detect join points in any host unless something different is specified explicitly (e.g., using the distribution pointcuts).

Pointcut *target(T)* matches all the join points where a method is called on an object of type *T*, where *T* is a type as introduced in Java language; it is also possible to write the constructor using the syntax *target(t)*, with *t* being a valid identifier of a parameter defined in the pointcut signature(see aspect language definition), and it will match the join points where a method is called on objects of the same type as the parameter t, with the additional behavior that the target object will be bound to the parameter t.

Pointcut *args(T,T,T,...)* will match all the method calls with the same arguments as those presented in the constructor; whit *T* being a type as defined in Java language by *T*; if the programmer uses parameters identifiers (e.g. *t)* in addition the parameters values of the method call will be bound to the parameters defined in the pointcut signature.

Next line introduces *cflow(P)*, this construct have the same behavior as that defined in[MN04] where the cflow information is maintained across different remote method call invocation in different participant hosts. This constructor identifies all the join points that occur between the start and end of the method specified by the constructor, this definition includes the join points that are executed in remote hosts.

The constructor *host(HExpr)* matches all the join points that appear in an specific host, represented by *hostExpr.* The host expression passed as parameter can be: an string with the ip address and the port representing a node; any of two provided constants *localhost* and *jphost*; or an identifier of a string parameter defined in the signature of the pointcut. *localhost* is the representation of the node where the aspect is deployed. *jphost* represents the node where the join point under evaluation was generated. When an identifier is passed as a parameter an string representing the node (i.e., String of the form “ipAddress:port”) is bound to the specified parameter defined in the pointcut signature(see aspect language definition).

The term *seq({PSeqStep})* matches sequences of pointcuts. The events defined

in the sequence are matched individually once they happen. This basic behavior can be modified by the construct *step(pSeq, stepId)* that will only match the step identified by *stepId* in the sequence. Note that to match the joinpoint that completes the full sequence (i.e., match the sequence when it has finished) the programmer can use a construct like *step(seq(s1:pSeq1, s2:pSeq2, send:pSeq3), send)*. An aspect can has as many sequences definition as desired, and each join point will be capable of changing all the sequences states when evaluated inside an aspect. The sequence expression can be also defined using regular expression notations (see non-terminal PSeqExpr).

The last defined constructs are the ones related to the neighborhood concept, these have a similar behavior as the *hosts* construct. *neighborhood(NExpr)* match all the join points that are originated in a host that belongs to the specified neighborhood (set of nodes). Likewise the construct *onneighborhood(NExpr)* match all the join points when the aspect evaluating a join point belongs to the specified neighborhood. *nExpr* is a string construct as defined by Java language that defines the name of the neighborhood. The names of the neighborhood are created dynamically in the advice, when nodes ask to be included in a neighborhood that does not exist yet, it will be created with the demanding node as the only element. Also this neighborhood will exist until the last element is removed. A node can belong to many neighborhoods, and many neighborhood can be defined.

3.2.3 Aspect and advice language

The aspect language is presented in Figure 3.3.

The non-terminal *AspDecl* presents the basic syntax for an aspect. The *global|local* modifier indicates if the aspect will be deployed locally or globally, in other words if it will be distributed to be loaded in other nodes or if is going to be loaded only locally. Note that the default behavior of an aspect is ti be distributed globally and to create one singleton instance on each node where is deployed. An aspect is marked by the *aspect* keyword, followed by the aspect's class and a list of element declarations, i.e., variable, pointcut or advice declarations. The aspect declaration can be modified by "per" clauses that modify the instantiation behavior, a deep explanation of that feature can be found in section 3.4.

The non-terminal *PcutDecl* presents the syntax used to write pointcuts definitions, in this case the keyword *pointcut* is used to designate a pointcut construct, an *Identifier* string represent the name of the aspect, and the *ParamPattern* is the signature of the pointcut. This signature contains the variables that will be bound with values extracted from the joinpoint and that can be used in the advice. The aspect can have as many pointcut definitions as needed by the programmer. The modifier *triggering* is used to indicate that the pointcut can crate instances of the aspect depending of the set of bindings provided by that specific pointcut, this

$$
\begin{array}{lll}
AspDecl & ::= & [LocalityExpr]\ aspect\ IdAsp\ [instExpr]'\{'\{elemDecl\}'\}' \\
LocalityExpr & ::= & global \mid local \\
InstExpr & ::= & perthread \mid perobject \mid perclass \mid perbinding \\
ElemDecl & ::= & Vardecl \mid PcutAdvDef \\
PcutAdvDef & ::= & [PcutDecl]\ AdviceDecl \\
PcutDecl & ::= & [triggering]\ pointcut\ Identifier(paramPattern) \\
Vardecl & ::= & javaVearDecl \\
ParamPattern & ::= & (javaType\ Identifier)? \\
 & \mid & javaType\ Identifier\ (,\ JavaType\ Identifier)* \\
AdviceDecl & ::= & AdvicePrim(ParamPattern)\ :\ PcutInvo\ [:\ on(HExpr)]'\{'\{Body\}'\}' \\
PcutInvo & ::= & Identifier(ParamPattern) \mid P \\
AdvicePrim & ::= & before \mid around \mid after \\
Body & ::= & JavaInstr \\
 & \mid & proceed(identifierPattern) \\
 & \mid & addneighborhood(NExpr) \\
 & \mid & removeneighborhood(NExpr) \\
IdentifierPattern & ::= & (identifier)? \mid identifier\ (,\ identifier)? \\
HExpr & ::= & localhost \mid jphost \mid "ipAddr : port" \mid Identifier \\
JavaInstr & ::= & //Java\ valid\ instruction \\
Identifier & ::= & //Java\ valid\ identifier \\
JavaType & ::= & //Java\ valid\ type\ expression \\
Int, ipAdd, port & ::= & //integer\ expression \\
NExpr & ::= & //A\ valid\ Java\ string\ expression
\end{array}
$$

Figure 3.3: Aspect language

option is activated only if the pointcut is used as a step of a sequence definition, this concept will be explained section 3.4.

Advice *AdvicePrim(paramPattern) : PcutInvo[:on(HExpr)]{{Body}}* is declared using AspectJ-like weaving modifiers *AdvicePrim* (before or after or around). The advice's *ParamPattern* determines the parameters that will be used in the advice and it has to be the same as the named pointcut used in the definition. Then the programmer can compose a pointcut using named pointcuts and/or the pointcut language. Advice bodies are constructed as follows: *JavaInstr* is a primitive of the Java language or a valid expression to express programs in the language. The *proceed*() construct is available when the *around* advice is used, this instruction will execute the advised join point with the parameters given in the proceed declaration, this construct is of particular importance because the execution context can be changed by modifying the parameters passed to the *proceed* construct. The advice language is augmented with the constructs *addneighborhood* and *removeneighborhood*, with these constructs the programmer can tell at runtime if a node joins or leave a specific neighborhood. I will show how this construct can be exploited in the section 3.5.

Note that the advice constructor can be modified by a *on* expression, the differences of this constructor and the one presented in the pointcut language are: first, this one generates a synchronous call in the specified host to a first class advice object that is executed in the context of the remote node, so it has access to the static information on that host; second, it can use the values binded by the pointcut in the variables of the pointcut definition, thus it can dynamically decide where to execute the call. This concept will be covered in example of section 3.5.2 on page 36.

3.3 Discussion of sequence variants

Sequence relationships, in particular regular ones, have received a lot of interest during last years in AOP[RD02, WV05, DFS04, DFL$^+$05, AAC$^+$05, WV04]. This sequence relationships are particular useful in a distributed setting, i.e., in order to express all sorts of communication protocols[DFL$^+$05, WV04]. One of the examples proposed at the end of this chapter is intended to express a cache with sophisticated cache policies. Such example shows how sequence can be used to build a protocol that will neighborhood communication between different nodes (see section 3.5.1 for details).

dhamaca provides explicit support for sequences, this support is provided by the pointcut language. The sequence can have different semantics that can be controlled specifically by the pointcut language:

- The first case of sequence is a sequence pointcut that is triggered at each

change of the state in the sequence, this is the default behavior and to achieve it the *seq* constructor has to be used.

- The second case is where particular steps of a sequence can be advised. In this case a step expression is used to modify the sequence default behavior, this constructor receives a sequence and a *stepId* value that represents explicitly what step of the sequence should be detected, the resulting pointcut can be advised accordingly.

- The third case of sequence is a sequence that is advised when all events of the sequence have occurred, this behavior achieved using a constructor like *step(seq(s1:pSeq1, s2:pSeq2, send:pSeq3), send)* constructor. This is just using the step constructor to advice the last event of the sequence.

- A fourth kind of sequence can be created by specifying at each step what will be the next step or combination of steps in the sequence, thus it will be possible to have a definition for stateful aspects as presented in [DFS04, WV05]. Note that the special construct *s1||s2*, where *s1* and *s2* are ids of steps in the sequence, has the meaning of leaving the sequence in any of those states, and the one selected will be the one who's event arrives first.

- finally a fifth kind of sequence will be the one crated in side an aspect that is modified by the keyword *perbinding*, such an aspect will be instantiated when the first event of sequence appears, binding a new set of values to the variables defined in the pointcut signature. Thus, a new aspect instance will be created each time that a triggering pointcut defined as an step of a sequence is found, only if there is not an instance of such aspect with the same set of values bounded to the signature variables (see section 3.5.1).

Listing 3.3 shows some specific examples to clarify the semantics of the sequence construct.The example defines a simple three step sequence that detects when the update of a user has been done inside a transaction. Line number 14 shows a pointcut that define the intersection between a sequence pointcut and a primitive call pointcut, the resultant behavior is that the intersection of the two set of events defined by each pointcut is detected. In this case the second step in the sequence will trigger the advice execution. Line number 16 shows how the step constructor can be used to express the same behavior, extracting the second step in the sequence. Lines 18 and 20 show how it can be achieved in different ways the behavior of triggering an advice only when the full sequence has been completed.

The language supports regular expressions notation as syntactic sugar for sequence expressions, note that in the example the second step in the sequence definition is modified by a *, this will mean that the second step can be matched many

```
1
2  pointcut transacStart(): call(* transacManager.startTransaction(..));
3
4  pointcut userUpdate(): call(**.obasco.User.set*(..));
5
6  pointcut transacEnd(): call(*transacManager.endTransaction(..));
7
8  pointcut userPersisted(): call(*persistanceManager.persistUser(..))
9
10 pointcut seqUserUpdate(): seq(transacStart(), userUpdate()*,
11
12                                      transacEnd(), seqEnd: userPersisted());
13
14 before(): seqUserUpdate() && userUpdate(){//advice of the second jp
   of the sequence}
15
16 before(): step(seqUserUpdate(),2){//advice of the second jp of the
   sequence}
17
18 before(): seqUserUpdate() && userPersisted(){//advice of the last jp
   of the sequence}
19
20 before(): step(seqUserUpdate(), seqEnd){//advice the sequence when
   completed}
```

Listing 3.3: Sequence pointcut examples.

times before the occurrence of a end of transaction event. The modifiers supported are: the * modifier means that 0 or more join point will be detected, the + modifier will indicate that at least one join point must be detected and {n,m} will indicate that a join point will be detected a minimum of n times and a maximum of m times. if The * modifier is used in the last pointcut of a sequence, the sequence will not be consumed and it will remain active in the last state until the end of the program.

3.4 Discussion of Aspects instantiation and deployment

One problem that arise when you are designing Aspect Languages, is how an aspect is instantiated and what does the instantiation means. When the problem is considered in a distributed environment the question of how an aspect is deployed had to be considered too. For our language the instantiation won't be related with the scope of lookup of the pointcuts, in our case the scope and the instantiation will be handled in a different way. This separation will give the programmer a better flexibility in the implementation of programs using the aspect language.

The default behavior for instantiation and deployment, when no modifiers are used in aspect declaration, is: the aspect will be deployed to all the nodes and there a singleton instance of the aspect will be created.

Deployment behavior can only be modified to restrict the distribution of the aspect, thus with the modifier *local* in the aspect declaration, the aspect will be deployed only in the node where it physically deployed. Note that Dhamaca has a dynamic behavior, meaning that aspects can be loaded and unloaded at runtime.

Instantiation. Instantiation is controlled by the "per" clauses, these clauses are *perthread, perobject, perclass,* and *perbinding*. The *perthread* clause creates an aspect instance for each thread in the application, and the aspect will detect events only in that specific thread. The *perobject* clause will create an instance of aspect for each target object, and the aspect will detect only events of that object. The *perclass* clause will create an instance of an aspect per each class and will detect events in objects of that class, it is important to note that this aspect will have a global reach, meaning that they will detect events of objects of that class in any node of the distributed application. Finally the *perbinding* clause will instantiate an aspect for each set of values binded to variables of the signature of a sequence pointcut, thus a new instance of the aspect will be created for each new set of values bound to variables by a *triggering* pointcut defined in a sequence. This last concept will by clarified in the examples section.

3.5 Applications of distributed aspects

3.5.1 Sophisticated cache policies

Modern web applications deal with sessions of different users, these sessions can hold references to objects needed for the application to perform some tasks. Web applications are also a good example of applications that tend to run in clusters. Nowadays it is common to find web applications that serve different enterprises (for example the enterprise zones in one bank's web sites, i.e., sites of all subsidiaries, or applications deployed in Application Service Provider web sites); let us assume that such an application has a large object representing the enterprise, and that object is referenced by all the sessions of users that belong to a given enterprise. If we want to have this object in a replicated cache in the cluster, common cache architectures (cf. the description of the JbossCache in section 2.1.2) make it necessary to have all the enterprise objects replicated in all the caches independently from whether they are used in the specific cluster node or not. This is a waste of memory, so it is useful to have a more sophisticated cache policy, one that only locally adds an object but replicates when an object is updated or removed. Furthermore, it is important to note that the replicated behavior is only desirable in the nodes that hold a copy of the modified object. Solutions to this problem produce tangled code because crosscutting between the functional code and the distribution code(spatial tangling); they also create a complicated tangled trace because entropy introduce by the uncertainty of the node behavior (e.g. Which node is going to ask for what object at what time).

The example in listing 3.4 shows a replicated cache policy as described above implemented with Dhamaca. The joinpoints of interest in the problem are the local addition of a value to the local cache (*addInstr*) and the updation of non-local caches (*updateIstr*). The first pointcut *addInstr* is defined as a triggering event, meaning that this event will generate an instance of the aspect if there is no instance in the current host with the same bound values. The event is also defined as local, because of that the triggering condition will only apply when a local event occurs. The next pointcut defined in the example is *updateIstr*, this pointcut represents events that are calls to the put method in the cache on other machines. The third pointcut defines the sequence that will be attached to the aspect, the sequence is defined by a regular expression consisting in the occurrence of the event *addInstr* followed by multiple occurrences of the event *updateIstr,* each occurrence is advised. After the pointcut definitions we have two advice definitions. The first advice definition is triggered when the first event occurs, adding the current host to a *neighborhood*. The last advice is used to listen events in the neighborhood and accordingly modifies the cache contents. Note that the name of the neighborhood is given by the name of an identifier of the data that is stored, thus all the

```
1
2  aspect AdaptiveCachePolicy perbinding{
3
4
5
6  triggering pointcut addInstr(String h, String fqn, Cache c):
7
8     host(h) && host(localhost) && call(* Cache.put(String Object))
9
10    && args(fqn, Object) && target(c);
11
12
13
14 pointcut updateInstr(String fqn, Object o):
15
16    call(* Cache.put(String, Object)) &&
17
18    args(fqn, Object) && !host(localhost) && onneighborhood(fqn);
19
20
21
22 pointcut cachePolicy(Host h, String fqn, Object o, Cache c):
23
24    seq(s1: addInstr(h, fqn, c), s2: update(fqn, o)*);
25
26
27
28  after(String h, String fqn, Object o, Cache c):
29
30    step(cachePolicy(h, fqn, o, c), s1) {
31
32      addneighborhood(fqn);
33
34   }
35
36   after(String h, String fqn, Object o, Cache c):
37
38    cachePolicy(h, fqn, o, c){
39
40      c.cache.put(fqn, o);
41
42   }
43
44 }
```

Listing 3.4: Sophysticated cache policy example.

caches that include this data in their caches will join a neighborhood where data of that explicit object is synchronized.

This simple aspect modifies the cache behavior presented in section 3.1 as described previously: objects are included locally in the cache and the cache works as a replicated one in case of further updates of the object. Note that now the objects are only replicated on the nodes that have asked for them explicitly (avoiding the problem of waste of memory), and the cache synchronization is triggered by a remote joinpoint that activates the put instruction in the second advice of the aspect.

3.5.2 Simplifying RMI-based distributed applications

Distribution considered as a crosscutting concern has been analyzed among others by Soares et al. [SLB02] by means of an aspect-engineered version of a health care watcher system. The system was a manually-implemented tangled distributed system, that was converted to a non-distributed version and then distribution was added using AO techniques. In order to implement distribution, the authors use aspects to automatically add RMI code in the non distributed version of the software. A a particular characteristic of these application is that three aspects had to be considered: first an aspect to handle client distribution concern ; second an aspect to handle server distribution concerns; third an aspect to handle serialization issues. In the cited paper, they present a solution to change the context of the target object by a remote object in the advice using the *proceed*() construct, however the proposal is not functional mainly because of implementation details of RMI (The distributed object is a remote interface and not the actual object). So the authors finally had to advice each method of the interface to call a method of the remote object. Using our approach the distributed version can implemented using only one aspect, that handles all the client/server concerns and also the distributed execution. The aspect is presented in Listing 3.5 .

The example shows the use of the *on()* construct at the advice level, and shows the interesting application of the *proceed()* construct, by changing the context of the execution by providing a new target object in the remote host. Note that at this level the behavior of *on* at the advice level is a synchronous behavior, the advice is treated as an independent object that can be sent and executed in different context. In fact such an advice can be send through the network and it can access the static context of the remote application, in this particular example configuring a full remote method invocation (thus using dhmaca's support for weak mobility).

Note that the advice signature is modified by the *on("hostServer")* construct, this will cause *facadeCalls* joinpoint to trigger advice synchronous execution on the *hostServer.* Once in the *hostServer,* the message is bound to an advice with the same code as the original advice, there the *proceed* instruction will be executed

```
1
2  pointcut facadeCalls(HWFacade f):
3
4     target(f) && call(* *(..)) &&
5
6     !call(static * *(..)) && this(HttpServlet);
7
8  Object around(HWFacade f) throws /*..*/ :
9
10    facadeCalls(f) :on("hostServer"){
11
12    return proceed(HWFacade.getInstance());
13
14 }
```

Listing 3.5: Distribution aspect Example.

with a new parameter that replaces the target of the method invocation, by giving the remote instance of the facade.

3.5.3 Test of architectural constraints

The work presented by Chiba et al. [MN04], uses a test framework to show the applications of remote pointcuts. The example presented here, extends the idea of applying remote pointcuts in test frameworks, to detect possible architectural violations in a deployed system. Let us analyze the case where we have three servers: h0 holds a middleware application, h1 holds a web application, and h2 holds a backend application. There are two rules, the first one is that all interaction between h1 and h2 should be done through h0 and that all data base updates in h2 triggered from h1 should be included in transactions. The pointcuts to handle these rules are showed in Listing 3.6. The *pathPC* pointcut is the one that specifies the allowed path, the *transactionSeq* pointcut is the one that restricts the execution of operations inside a transaction, and finally the *archRule* pointcut detects the calls to the *set* methods on *ReqServer* classes in the host *h0* that violate the rules specified by the above pointcuts.

Note that this example shows an extension over the test framework presented in [MN04]. Here updates are checked to be made inside a transaction using sequences, thus avoiding the need of having control flow relation between calls. This kind of test are difficult to support without support for distributed sequences.

```
1
2  pointcut pathPC() :
3
4    cflow(call(* *.*(..)) && host(h1))
5
6    && cflow(call(* *.*(..)) && host(h0)) &&
7
8    call(* reqserver.set*(..)) && host(h2);
9
10
11
12 pointcut transactionSeq():
13
14   seq(call(* obasco.txmanager.start()),
15
16         *call(* *.*(..)) && host(h0),
17
18         call(* obasco.txmanager.end());
19
20
21
22 pointcut archRule():
23
24   call(* reqserver.set*(..)) && host(h2) &&
25
26   (!pathPC() || !transactionSeq())
```

Listing 3.6: Architectural constrains.

3.5.4 Conclusion

This section has shown the advantages of Dhamaca by providing concrete examples. Explicit neighborhood support in combination with remote pointcuts and distributed sequence support have been used to express context-dependant dynamic protocols and sub-protocols inside a distributed application, e.g., for sophisticated cache policies. It also has been shown how distribution can be achieved easily by using explicit distributed advice determination and avoiding common problems of remote interfaces. Finally, test frameworks can be extended to detect complex event patterns in distributed computation traces to check architectural constraints within distributed systems.

Chapter 4

DJAsCo: a prototype implementation

In this chapter I present DJAsCo, the implementation of the main features proposed by Dhamaca. It has been implemented on top of JAsCo [VS] [SVJ03], a dynamic AOP approach originally designed to combine ideas of aspect-oriented and component-oriented programming. This chapter will present a brief introduction to the JAsCo language, after that I will present the details of the proposed extension to be implemented and finally you will find the implementation details of the first DJAsCo version.

4.1 Introduction to JAsCo

JAsCo is a dynamic AOP extension of Java that was conceived originally to reconcile aspects and components. To achieve that it introduces two additional entities to the language: *aspect beans* and *connectors*. The main idea is that a programmer can produce independent aspect beans that can be bound at deployment time to actual components using connectors. Aspect beans are an extension of Java beans and are specified in an independent way, with out relating them to any specific component specification. These aspect beans contains one or more hooks that can be advised using before,around and after like constructors like in AspectJ. Figure 4.1 shows a simple logging aspect that prints the joinpoint info of the method that is being call. Basically in the aspect a hook called printTrace is defined such hook will be bind to a method (or set of methods) and will execute the *before* advice before the execution of the method. The advice just print the object that represent the joinpoint that is being advised.

Binding of concrete components methods to hooks in the aspect bean is done in the connector, the connector is used to deploy one or more instances of aspect

```
1
2  class PrintTrace {
3
4    hook PrintTrace {
5
6      PrintTrace(method(..args)) {
7
8          execution(method);
9
10       }
11
12       before() {
13
14          System.out.println("Aspect's before advice: " +
15
16                             "jp: " +thisJoinPointObject);
17
18          }
19
20       }
21
22 }
```

Listing 4.1: Aspect bean example

beans, binding the actual components' methods to the corresponding hooks. Notice that aspect instantiation is expressed explicitly in the language. Figure 4.2 shows a connector that create an instance of the aspect bean and binds all methods to the argument of the hook, in this case the aspect will trace all the method executions in all the classes of the component or base program.
Note that in the connector the programmer can use wild cards to attach different methods to the corresponding hooks, explicit precedence can be expressed in the connector and even advice granularity can be expressed in this way.

An interesting feature found in JAsCo is support for stateful aspects as described in the formal model presented by Südholt et al. [DFS04]. This feature allows the language to support protocol history conditions to trigger the advice execution in aspects, thus a programmer can specify a sequence of events and she can advice them individually or defining a common advice to all the transitions. In the language construct proposed, a finite state machine can be defined by specifying the states and the sate transition rules. note that events are joinpoints that match a defined pointcut in the construct. Figure[1] 4.3shows an stateful aspect where three methods are defined in the signature of the hook: *starmethod, run-*

[1]This example was extracted from[WV05].

```
1
2  static connector PrintConnector {
3
4    test.PrintTrace.PrintTrace hook0 =
5
6     new test.PrintTrace.PrintTrace(* *(*));
7
8         hook0.before();
9
10 }
```

Listing 4.2: Connector bean example

```
1
2  class statefulBean {
3
4     hook StatefulHook {
5
6         StatefulHook(startmethod(..args1),
7
8                           runningmethod(..args2),
9
10                          stopmethod(..args3)) {
11
12            start>p1;
13
14            p1: execution(startmethod) > p3||p2;
15
16            p3: execution(stopmethod) > p1;
17
18            p2: execution(runningmethod) > p3||p2;
19
20        }
21
22        before p2 (){
23
24             //do the logging:
25
26             System.out.println("executing: P2 ");
27
28        }
29
30     }
31
32 }
```

Listing 4.3: Stateful bean example

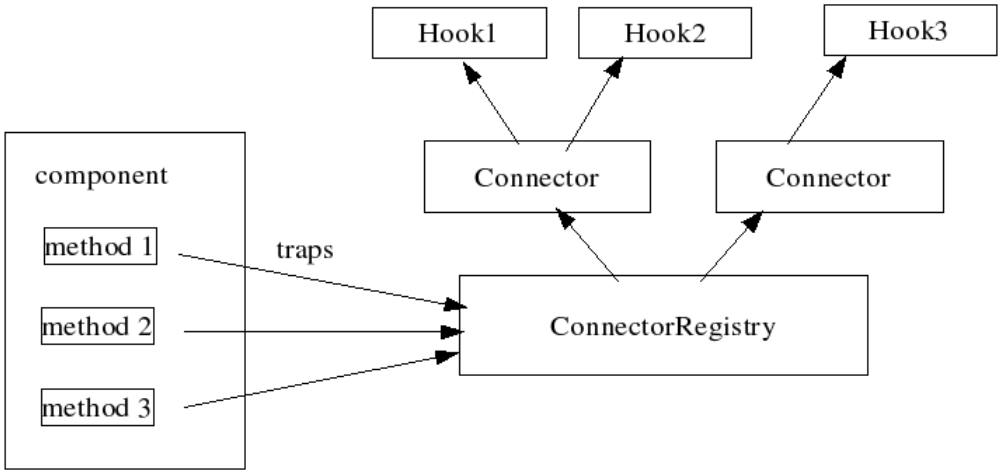


Figure 4.1: JAsCo run-time Architecture

ningmethod and *stopmethod*. Then, with such methods, a protocol is defined. In the first line a *start* construct indicates the first sate of the stateful aspect. In the first named state *p1* the program specify that when a execution of the startmethod event is triggered the program will change its sate to a kind of dual state where the application will be in parallel in state *p3* and *p2*, and the next state will depend of what change condition is satisfied first, by example if the first event to happen is a call to the stopmethod the state of the aspect will change to *p1*; on the other hand if the next event is a call to *runningmethod*, the program will change the state as defined in p2, in this case it will go to the same state as before (*p2||p3)*.

4.2 JAsCo architecture and its efficient execution

This chapter introduce some of the architecture features found in JAsCo. The characteristics presented are: JAsCo runtime architecture, JAsCo dynamic features, and JAsCo's Hotswap and Jutta optimizations.

The JAsCo team propose the runtime architecture shown in figure 4.1 , in the figure there is an already instrumented component with traps in its methods (JAsCo performs dynamic trapping of methods driven by connector definitions). The central connector registry serves as the main addressing element in the JAsCo runtime environment, in the registry all the connector and hooks are registered and whenever a trapped method call occurs is reached in the component, such an event is notified to the connector registry who looks for all the interested connectors and hooks calling the respective methods.

JAsCo is a dynamic AOP language, as such aspects can be deployed and undeployed at runtime. The basic mechanism to support that is the centralized control gave by the connector registry, this part of the system is monitoring the classpath of the application and it detects each time that a new connector is present activating it and its hooks. Likewise the system also detect when a connector is took off form the classpath to achieve undeployment.

This architecture facilitates the dynamic behavior of JAsCo, because the connector registry is always looking for the active connector each time that a trap is reached, however instrumenting all the methods of all the classes and processing all the traps in the connector registry gives a big overhead to the application, in order to solve this problem two main points have been attacked: the instrumentation phase and the aspect interpretation phase. For the instrumentation phase the JAsCo's developers propose to use Hotswap by means of the instrumentation api introduced in Java 1.5, with this optimization JAsCo achieves that only the methods that are of interest for some connector are instrumented. Thus when a connector is loaded the JAsCo Hotswap mechanism instruments traps only in the methods in which the connector is interested. The other proposed optimization is the introduction of Jutta, a just in time compiler for aspects, basically the Jutta systems creates and caches a highly optimized code fragment for a given joinpoint, this code executes the advice in the order specified in the connector. This cached code is sensitive to loading and unloading of connectors, meaning that the dynamic properties are kept and when a new connector is loaded or unloaded the respective cached code is generated or invalidated.

4.3 Language and features implemented by the prototype

This section describes the language of the prototype which is a sublanguage of the one proposed in the previous chapter. The selected set of features is large enough to address all the main features proposed in Dhamaca.

The proposed set of extensions to implement in the prototype are shown in figure 4.2. The *call* constructor extends the already supported JAsCo version by giving it a distributed wide scope. T *host* construct that can receive a constant string *h* indicating that joinpoints occurred in the host represented by *h* will be matched by this pointcut construct. Next line presents the *cflow* construct, that can receives a pointcut as parameter. The *seq* construct introduces basically the possibility to express sequences of pointcuts that will be match in aspects. The pointcut *on* is also supported such construct receives a constant string, as a parameter, indicating that that joinpoints will be matched and advised by the aspect

only if the aspect is deployed in the host represented by *h*. And finally we support to compose pointcuts using ||(or) , &&(and) and !(not).

$$
\begin{array}{rcl}
p & ::= & call(s) \\
 & | & host(h) \\
 & | & cflow(p) \\
 & | & seq(p,p,...) \\
 & | & on(h) \\
 & | & p||p|p\&\&p|!p
\end{array}
$$

Figure 4.2: Restricted pointcut language

4.4 Architecture of DJAsCo

The basic concept behind DJAsCo is to express communication at the language level meaning that the infrastructure to achieve distribution is provided by the AOP language. The features supported are:

- Explicit support for remote pointcuts as defined in the language section. JAsCo instances exchange messages to address communication between them.
- Reliable communication between nodes that participate in the distributed application.
- The programmer does not have to be worried about communication stuff like fragmentation or encryption. Most communications problems are not seen by the programmer.
- Detection of joining members, leaving members and crashed members. The runtime architecture detects automatically what nodes had been activated (A dJAsCo process has been started) or what nodes have left the deployment architecture.
- Extensible communication mechanism. The communication schema defined is extensible enough to support future features of the language, and provides mechanisms to address synchronous communication (through code mobility) and neighborhood support.
- Not centralized coordination. The architecture do not depends of one specific node in the deployment topology.

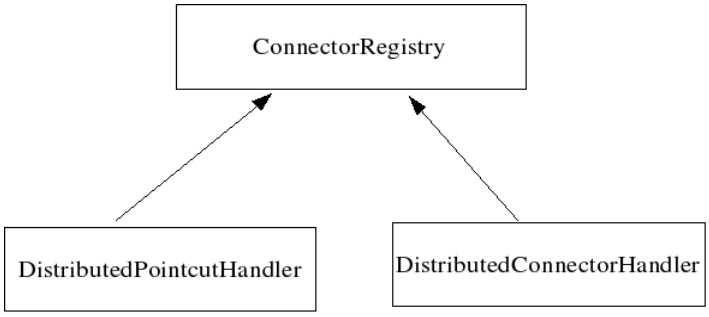


Figure 4.3: DJAsCo run-time architecture

- Running JAsCo in a local mode or running DJAsCo (in distribution mode) should be configurable.
- Support for distribution deployment of aspects, to achieve distributed wide dynamic AOP features. The dynamic features of JAsCo are also extended to DJAsCo.

To manage communication support between the runtime instances of JAsCo I am going to use Jgroups[Ban05], Jgroups api is designed to support reliable multicast communication. Jgroups provides an abstraction for groups of nodes. The API is very simple but powerful, first you have to create a channel, then you join a group and then you start receiving and sending messages. Such features makes Jgroups a perfect match for DJAsCo implementation. This features facilitate the implementation of the communication mechanism between the different JAsCo instances.

The provided implementation separates communication in two dimensions, in one dimension DJAsCo will handle the distribution of remote pointcuts and in the other dimension the runtime environment will handle the the protocol for loading and unloading of connectors to achieve distributed dynamic behavior, meaning that activations or deactivation of aspects can have a distributed effect(we will cover this subject later).

The main modifications made on JAsCo were basically two: first, inserting code for intercepting the events of joinpoint detection and loading and unloading of connectors; second making the related objects suitable for transmission (that in Java terms is making them serializable) and creating extended versions to store distribution specific information like the original host. In figure 4.3 is shown the runtime architecture of JAsCo, basically the ConnectorRegistry behavior is augmented by adding two elements that are listening to events encountered in the connector registry, the first element intercepts the detection of traps in the

connector to create objects representing remote pointcuts that are shared with all the other members of the group. Note that when a DJAsCo instance is started it joins a default group where it start exchanging messages. The other component creates another communicating group where the connectors events like loading and unloading are shared to achieve distributed wide dynamic AOP.

Figure 4.4 shows the class diagram with the core extensions made to JAsCo in order to add distributed support. As shown in the picture the connector registry has a default trap executor, that executor was extended to create a distributed version called *DistributedTrapExecutor*. This class is used whenever JAsCo is started in DJAsCo mode. The class is responsible for delegating the distribution of *DistributedJascoMethod* to the *DistributedPointcutHandler* instance. This last class is a extend creates a singleton object that implements the *RequestHandler* interface from Jgroups, hence it is the responsible of creating the joinpoint default neighborhood.

The connector framework is made by implementing a *ConnectorRegistryListener* interface, that listener is called *DistributedConnectorListener*, and its main function is to delegate distribution of connectors to the *DistributedConnectorHandler*. This last class creates a singleton object that implements the *RequestHandler* interface from Jgroups, hence it is responsible of connector distribution.

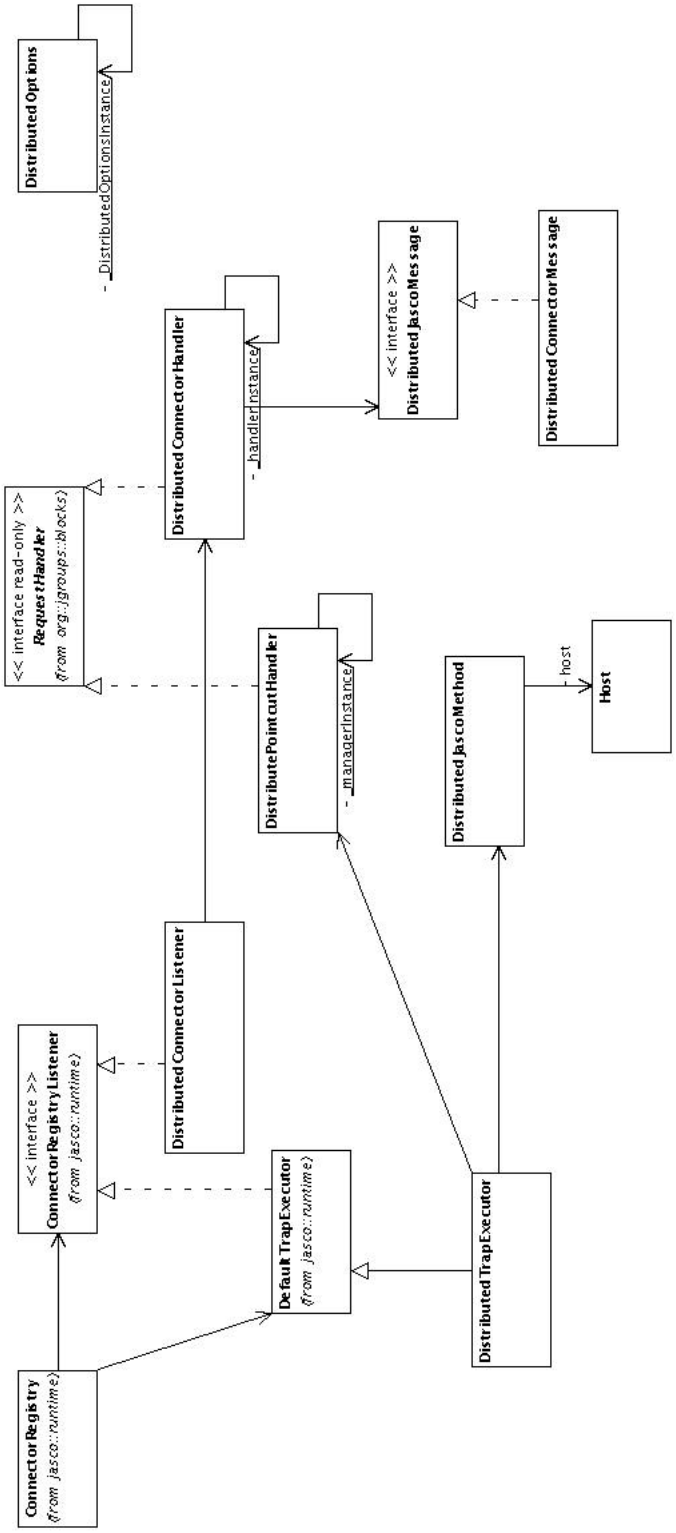
The source code of the *DistributedPointcutHandler* and *DistributedConnectorHandler* is given in appendix A and B.

4.5 Implementing remote pointcut support

As mentioned before a default group of servers is created for exchanging of messages with the information of the event on a joinpoint. Each time the runtime of DJAsCo is started it joins to a group where it is allowed to exchange messages with a distributed joinpoint information, such messages are send to every node in the group except the one were the joinpoint originated from. Once this join point arrives it will be treated as a local joinpoint, it will processed by JAsCo's connector registry. Look that with this basic extension all the pointcuts constructs achieve a distributed wide scope, this is because, even tough the host information is present in the joinpoint, the distributed information is used only by specialized pointcut constructs that will be introduced later. Lets explain this with a simple trace example presented in listing 4.4, this code is the same as shown in the introduction to JAsCo language but in this case the pointcuts constructs are remote pointcuts, meaning that they will check also for joinpoints executed in other nodes different from the one where they are deployed.

If we consider that the connector deploying this aspect is the one presented in listing 4.5, then all the joinpoints executed in any machine involved in the dis-

Figure 4.4: DJAsCo class diagram



```
1
2  class PrintTrace {
3
4    hook PrintTrace {
5
6      PrintTrace(method(..args)) {
7
8          execution(method);
9
10       }
11
12       before() {
13
14          System.out.println("Aspect's before advice: " +
15
16                             "jp: " +thisJoinPointObject);
17
18          }
19
20       }
21
22 }
```

Listing 4.4: Distributed aspect bean example

```
1
2  static connector PrintConnector {
3
4    test.PrintTrace.PrintTrace hook0 =
5
6     new test.PrintTrace.PrintTrace(* *(*));
7
8          hook0.before();
9
10 }
```

Listing 4.5: Distributed connector

```
1
2  class PrintTrace {
3
4    hook PrintTrace {
5
6      PrintTrace(method(..args)) {
7
8          execution(method) && joinpointhost("192.168.17.248:10895");
9
10       }
11
12       before() {
13
14          System.out.println("Aspect's before advice: " +
15
16                             "jp: " +thisJoinPointObject);
17
18          }
19
20       }
21
22 }
```

Listing 4.6: joinpointhost ussage

tributed application will be logged by this aspect (Note that this aspect will be deployed and instrumented in all other machines so this aspect will start logging in all the active nodes, I will cover this feature later in the section regarding distributed deployment of aspects).

Until now we have incorporated the remote joinpoint support to the framework but we have not introduced yet any remote pointcut construct to deal with these distributed joinpoints. So now I will show how the *host* construct of the language can be used. Such construct will match only joinpoints that are issued in an specific node of the topology. Listing 4.6shows the trace aspect version with the new pointcut construct *joinpointhost* in line 8. The introduction of this new element restrict our logger aspect and now only the joinpoints that are executed in the specified node are detected and advised.

As another example, consider an application of the negation construct introduced in section 4.3 on page 45, any pointcut can be operated with *!* operator, so the an expression as *!joinpointhost("192.168.17.248:2345")* is feasible. This kind of pointcut will be matched only if the joinpoint's node is different from that specified in the parameter of the constructor.

```
1
2  class PrintTrace {
3
4    hook PrintTrace {
5
6      PrintTrace(method(..args)) {
7
8          execution(method) &&
9
10               executionhost("192.168.17.248:10895");
11
12       }
13
14       before() {
15
16          System.out.println("Aspect's before advice: " +
17
18                             "jp: " +thisJoinPointObject);
19
20          }
21
22       }
23
24 }
```

Listing 4.7: executionhost ussage

4.6 Distributed advice support

One of the main contributions of our language is the support of distributed advice, to achieve this a new construct, called *executionhost*, was introduced this construct will match an joinpoint if the aspect is deployed in the node specified by the parameter of the construct. Listing 4.7shows the trace aspect version with construct *executionhost* in line 10, this line change the behavior of the example, now the aspect deployed in the specified node will be the only one to log the trace of the distributed application. It is important to note that the domain of events is again all the joinpoints in the distributed application and here the restriction applies to the node where the advice is executed.

The *executionhost* construct can be negated using *!*, so an expression of the form *!executionhost("192.168.17.248:2345")* is feasible, and will produce that all aspects, deployed in nodes different from the one specified in the construct, to be executed.

Now that we have more that one construct that is distributed aware in the pointcut language of DJAsCo we can operate the using && and ||. Listing 4.8shows the

```
1
2  class PrintTrace {
3
4    hook PrintTrace {
5
6      PrintTrace(method(..args)) {
7
8              execution(method) && joinpointhost("192.168.17.248:10895") &&
9
10                        executionhost("192.168.17.248:10895");
11
12       }
13
14       before() {
15
16          System.out.println("Aspect's before advice: " +
17
18                           "jp: " +thisJoinPointObject);
19
20          }
21
22       }
23
24 }
```

Listing 4.8: operation between pointcuts

operation between *executionhost* pointcut and *joinpointhost* pointcut in line 10, this aspect will match all the joinpoints originated in "192.168.17.248:10895" and will execute the advice only if the aspect is deployed in "192.168.17.248:10895". I this case it will only match local joinpoints in the node "192.168.17.248:10895".

4.7 Distributed deployment of aspects

Dynamic loading and unloading of connectors is achieved by intercepting the original loading events from JAsCo's connector registry, once a connector is loaded in a node (i.e., it is copied in the class path of the application) the connector's class file bytecode is read and packed in a message, then that message is distributed to other machines. When a distributed connector message arrives, the node creates the class from the byte code and the connector is loaded. As mentioned at the beginning of this chapter, one requirement of DJAsCo was preservation of the dynamic features provided by JAsCo augmenting them with a distributed behavior, and this is how it has been done. In that way during the architecture design I decide to create a separate connector's default group to share connector information between the different runtime instances with out interfering with the distributed joinpoint communication. The element in charge of manging the distribution of aspects is the *DistributedConnectorHandler* found in figure 4.3, this element will be listening when its local connector registry loads or unload a connector, it will also receive connectors loaded in other runtime instances.

As mentioned above the first intended behavior for the *DistributedConnectorHandler* is to send messages with information about the occurrence of connector's loading events, once such events happen this element is responsible of assembling the corresponding message and sending it to the group of nodes. It is also responsible of receiving the distributed messages and taking the corresponding actions. Note that the communication with the local *Connector Registry* is done using the file system, in this way the events of loading and unloading the connector are mapped to writing and un-writing explicitly the connector file using Javassist[Chi04].

It is important to mention that DJAsCo uses the optimization enhancements introduced by JAsCo Hotswap and Jutta as discussed at the beginning of this chapter. In such a case note that even tough all the connectors are distributed it does not mean that all the methods in the base program are instrumented, the Hotswap facilities instrument only the methods that will be of interest for any aspect deployed in the distributed system. Likewise Jutta will be applied whenever such optimization is feasible.

4.8 Distributed cflow, a custom socket approach

The main goal of a distributed cflow implementation is to have allow the programmer to write aspects of the form

```
cflow(call(* *.A(..)) &&
  host("192.168.17.5:2345")) &&
    host("192.168.17.9:4444")
```

which means that all the joinpoints, found in node *"192.168.17.9:4444"* that are in the control flow of a remote call originated in the control flow of an invocation of method *A* in the host *"192.168.17.5:2345"*, will be matched. Such behavior presents an inconvenience in the implementation, the inconvenience is that the middleware mechanism for the remote call must transmit the stack information of the caller process. This is not supported in common middleware implementations like Java's RMI. This inconvenience feature can be solved restricting the scope of the distributed cflow support as showed below.

To have a distributed version of cflow we will restrict the scope of this feature, it will be a middleware dependent implementation, in this case it will be an RMI dependent implementation. The basic idea is that the system will provide a custom socket implementation [SM], such implementation has to be used by the distributed application in order to enable DJAsCo to detect cflow distributed constructs. The custom socket implementation will encapsulated in the transmitted stream the stack information that will be stored in a *ThreadLocal* variable in the remotely invoked service. DJAsCo provides the classes that implement the custom sockets factories that can be used to achieve the behavior described above:

- JAsCo.util.distribution.JAsCoClientSocketFactory
- JAsCo.util.distribution.JAsCoServerSocketFactory

Using this classes the server component in an RMI application should be modified in the way that is showed in listing 4.9. Note that only the server part, or in more general way, the code that publish RMI services should be modified. The client will use the correct custom client socket factory that will be downloaded to establish the communication with the published service, that is why only the "server" component need to be modified.

Custom socket implementation is an RMI provided feature. Chiba et al. [MN04] proposed to use that feature to implement distributed cflow, their proposal is the one described here.

```
1
2  try {
3
4     if (System.getSecurityManager() == null) {
5
6           System.setSecurityManager(new RMISecurityManager());
7
8     }
9
10    String name = "Server";
11
12    System.out.println("Starting Server sercice in:" + name);
13
14    RMIClientSocketFactory csf = new JascoClientSocketFactory();
15
16    RMIServerSocketFactory ssf = new JascoServerSocketFactory();
17
18    ServerInterface stub = (ServerInterface) new ServerWrapperImpl(0,
19
20                                    csf, ssf);
21
22    LocateRegistry.createRegistry(1099);
23
24    Registry registry = LocateRegistry.getRegistry(1099);
25
26    registry.rebind(name, stub);
27
28    System.out.println("Service started");
29
30 } catch (Exception e) {
31
32           e.printStackTrace();
33
34 }
```

Listing 4.9: Modified RMI server to suport custom jasco socket implementation

Towards an optimized cflow implementation. The proposed cflow implementation has three major drawbacks: first, it is dependent of the middleware mechanism (RMI); second, that the stack information is sent in each remote invocation even if there is no aspect using the cflow construct; and third, it is necessary to modify the server part of the distributed application. A solution to address all this issues does not exist at the moment but here I present two proposal that were discussed during the development of this thesis.

The first proposal is to introduce a distributed stack as in [AFT99], this solution presents an advantage in the sense that the stack information is not transmitted so the messages send are shorter and network resources are saved. However the computation of cflow checks have to use network communications and still the middleware has to be modified in order to distribute the original thread id.

The second proposal is to change all the application method signatures to include the stack information. In this way is possible to have a middleware independent application, however this will imply an overhead during the instrumentation of all the methods of the base application. This idea is a distributed extension of cflow-passing style proposed by Pengcheng Wu [Wu05]. Note that extending the method invocation information with thread id information can generate even a more efficient mechanism for the distributed cflow.

4.9 Distributed sequence support

JAsCo provides a construct that support stateful aspects, with the new support for remote pointcuts introduced in DJAsCo now is possible to have a sequence distributed construct that uses the default behavior gave by JAsCo's stateful aspects. It is important to note that in this implementation the joinpoints are the base events, thus changing of state in the defined finite state machine is triggered by pointcut matching. Listing 4.10present a distributed enable stateful aspect, such an aspect defines three methods *startmethod*, *executionmethod* and the *stopmethod* with this method arguments the code shows a simple logging aspect that logs all the occurrences of the *executionmethod* in any host between a call of the *startmethod* in the host "192.168.17.248:9876" and a call to the *stopmethod* in any host. This kind of syntax provides a sequence construct and provides an example of how advising of an specific event in the sequence can be used. This example is the distributed version of the example proposed in section 4.1.

Dhamaca has defined difference sequence semantics, in DJAsCo we decide to support only JAsCo-style ones. But this not constitute a serious restriction to show how distributed sequence pointcuts can be implemented.

```
1
2  class statefulBean {
3
4     hook StatefulHook {
5
6        StatefulHook(startmethod(..args1),
7
8          runningmethod(..args2),stopmethod(..args3)) {
9
10           start>p1;
11
12           p1: execution(startmethod) &&
13
14                joinpointhost("192.168.17.248:9876")> p3||p2;
15
16           p3: execution(stopmethod) > p1;
17
18           p2: execution(runningmethod) > p3||p2;
19
20       }
21
22       before p2 (){
23
24           //do the logging:
25
26           System.out.println("executing: P2 ");
27
28       }
29
30    }
31
32 }
```

Listing 4.10: Distributed stateful aspects

4.10 JAsCo cache metrics compared to JBossCache metrics

As a conclusion for this section I propose LOC count as a comparison metric between the DJAsCo distributed version of the replicated cache and the JBossCache implementation.

DJAsCo LOC metrics are: core distribution package 410 LOC; custom sockets 303 LOC; JAsCo pointcut extensions 133 LOC. The replicated cache example metrics are: cache class 60; aspect 26. Total LOC count, considering DJAsCo extensions, is 932.

As mentioned in section 2.1.2 JBossCache main class LOC count was 1741, note that the full implementation in DJAsCo is half the size of one of the classes of the solution implemented with standard Java.

Chapter 5

Conclusions

The development of distributed applications with current programming languages is a difficult task. The resulting programs have tangled code of functional concerns and distribution concerns. In particular because, distribution concerns are often crosscutting concerns w.r.t functional concerns. Aspect oriented programming (AOP) promises to provide means for encapsulation of the so-called crosscutting concerns. However few approaches with language support for distribution have been developed.

This thesis has introduced Dhamaca, an AOP language with explicit support for distribution. The Dhamaca language was designed as an extension of the Java language. It provides constructs for remote pointcuts (in particular *cflow* and *seq*), distributed advice, aspects and neighborhoods.

Dhamaca was designed to have simple concepts to support distribution. Dhamaca's join point model consist of method calls only. The semantics of method calls are modified by the introduction of the concept of groups of nodes. A node is any runtime instance of Dhamaca processes (programs written in Dhamaca). More than one node can be running in a machine. A group of such nodes may form a neighborhood. A node can also belong to many neighborhoods. These groups are created by runtime instances of Dhamaca processes (nodes), such instances can join and leave the groups at any time.

Support for different sequence variants that have been introduced in the AOP field are supported by Dhamaca.

Several examples were presented to show how Dhamaca can be used to write distributed applications. In particular it has been shown how a replicated cache implementation is easily realizable compared to implementations using standard Java platforms, such as JBossCache.

Finally, a prototype of Dhamaca was developed by extending JAsCo. The runtime architecture of JAsCo was augmented with a communication layer implemented in Jgroups. Working at the bytecode level to encapsulate connectors

in messages, it has been also possible to achieve aspect distribution by means of connector distribution.

This thesis paves the way for several leads of future work, in particular at the language level, the implementation level, and the application level. At the language level, of particular interest are: formalizing the semantics of the language using a calculus for distribution with support for neighborhoods. Adding support to have fine grained control over parameter passing in remote advice invocation. Addressing other relevant crosscutting concerns, i.e., concurrency and partial failure[Lop97, SLB02, KG02].

At the implementation level: Exploring techniques for optimization of the cflow and sequence implementations. It is particularly interesting to think how a distributed version of a cflow that do not depend on the middleware infrastructure can be implemented. One approach could be to augment the information passed in the method invocations, so that stack information can be passed.

At the application level: Extend the base of examples in the replicated cache domain by adding to the cache Dhamaca's version support for transactions and object graph mapping [BW05].Using Dhamaca for examples in other domains. A promising application is support for p2p networks with semantic caching realized by introducing the concept of locality [DFJ+96, RD00].

Appendix A. Pointcut distribution handler source code

```
/**
 *
 */
package jasco.runtime.distribution;


//JGroups imports
import jasco.runtime.ConnectorRegistry;
import jasco.runtime.JascoMethod;
import jasco.util.logging.Logger;

import org.jgroups.*;
import org.jgroups.blocks.*;
import org.jgroups.stack.IpAddress;
import org.jgroups.util.*;

/**
 * This singleton class manages the comunication of the Jasco
 * joinpoint(methodsInvocations) events with the group of
 * instances that belongs
 * to the same group.
 *
 * @author lbenavid
 */
public class DistributePointcutHandler implements RequestHandler {
private static DistributePointcutHandler _managerInstance;

Channel            channel;
    MessageDispatcher  disp;
```

```
    RspList            rsp_list;
    String             props =
     DistributedOptions.getInstance().getStack("AspectsGroup");


/**
 * This class is a singleton class
 */
private DistributePointcutHandler() {
try {
start();
} catch (Exception e) {
Logger.getInstance().showDebug(e);
}
}

public Channel getChannel() {

return channel;
}

    public void start() throws Exception {
        channel=new JChannel(props);
        channel.setOpt(Channel.LOCAL, false);//Not self delivery
        disp=new MessageDispatcher(channel, null, null, this);
        channel.connect("MessageDispatcherTestGroup");
        Logger.getInstance().showDebug(channel.getView().toString());
    }

public static DistributePointcutHandler getInstance(){
return _managerInstance;
}

public void sendNotification(DistributedJascoMethod jm){
if(!jm.isRemoteJP()){
   if(jm.getHost() == null)
  jm.setHost(new Host((IpAddress)channel.getLocalAddress()));
  jm.setRemoteJP(true);
  Object tmpObject = jm.getCalledObject();
  jm.setCalledObject(null);
      rsp_list=disp.castMessage(null,
```

```
                new Message(null, null, jm),
                GroupRequest.GET_NONE, 0);
      jm.setCalledObject(tmpObject);
      jm.setRemoteJP(false);
}
}

/**
 * This methods execute the trap to the remote pointcut.
 * If the trap sends an error a false is answered, if the
 * traps executes ok a true is answered. This can be changed
 * to send objects, but for now there is no need for that.
 * */
public Object receiveNotification(JascoMethod jm){
Object result;
try {
ConnectorRegistry.executeTrap(jm.getName(), jm.getCalledObject(), jm);
result = true;
} catch (Exception e) {
Logger.getInstance().showDebug(e);
result = false;
}
return result;
}

public Object handle(Message msg) {
Logger.getInstance().showDebug(msg.toString());
        return receiveNotification((JascoMethod) msg.getObject());
    }
/**
 * initializes this manager instance
 *
 */
public static void init() {
if(!initialized())
_managerInstance=new DistributePointcutHandler();
else throw new IllegalArgumentException(
"Cannot initialize distribution manager more than once");
}

public static boolean initialized() {
```

```
return _managerInstance!=null;
}

}
```

Appendix B. Connector distribution handler source code

This appendix shows the source code of the communication handler in charge of connector distribution in dJAsCo runtime instance.

```
/**
 *
 */
package jasco.runtime.distribution;

// JGroups imports
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.util.HashSet;

import javassist.bytecode.ClassFile;

import jasco.runtime.ConnectorRegistry;
import jasco.runtime.connector.Connector;
import jasco.util.logging.Logger;

import org.jgroups.*;
import org.jgroups.blocks.*;
import org.jgroups.util.*;

/**
 * This singleton class manages the comunication of the Jasco
 * connector events with the group of instances that belongs
 * to the same group.
```

```
 *
 * @author lbenavid
 */
public class DistributedConnectorHandler implements RequestHandler {
private static DistributedConnectorHandler _handlerInstance =
new DistributedConnectorHandler();

private Channel channel;

private MessageDispatcher disp;

private RspList rsp_list;

private String
props = DistributedOptions.getInstance().getStack("ConnectorsGroup");

private HashSet connSet = new HashSet();

/**
 * This class is a singleton class
 */
private DistributedConnectorHandler() {
try {
start();
ConnectorRegistry.addConnectorRegistryListener(
new DistributedConnectorListener());
} catch (Exception e) {
Logger.getInstance().showDebug(e);
}
}

public Channel getChannel() {

return channel;
}

public void start() throws Exception {
channel = new JChannel(props);
channel.setOpt(Channel.LOCAL, false);// Not self delivery
disp = new MessageDispatcher(channel, null, null, this);
channel.connect(DistributedOptions.getInstance().getGroupName(
```

```
"ConnectorsGroup"));
Logger.getInstance().showDebug(channel.getView().toString());
}

public static DistributedConnectorHandler getInstance() {
return _handlerInstance;
}

public void sendNotification(DistributedJascoMessage dm) {
rsp_list = disp.castMessage(null, new Message(null, null, dm),
GroupRequest.GET_NONE, 0);
}

/**
 * This methods is executed when a connector notification is received
 */
public Object receiveNotification(DistributedJascoMessage aMessage) {
Object result;
try {
if (aMessage.getType() == DistributedConnectorMessage.ADD_CONNECTOR) {
ByteArrayInputStream in =
new ByteArrayInputStream((byte[]) aMessage.getMsg());
ClassFile cf = new ClassFile(new DataInputStream(in));
(new File("Connector")).mkdir();
File f = new File("Connector/" +
cf.getSourceFile().replaceFirst(".java",".class"));
FileOutputStream out = new FileOutputStream(f);
cf.write(new DataOutputStream(out));
f.deleteOnExit();
out.close();
connSet.add(cf.getName());
}
if (aMessage.getType() == DistributedConnectorMessage.REMOVE_CONNECTOR) {
String fileName = aMessage.getMsg().toString().replace('.', '/') + ".class";
System.out.println(fileName);
File f = new File(fileName);
f.delete();
}
result = true;
} catch (Exception e) {
Logger.getInstance().showDebug(e);
```

```
result = false;
}
return result;
}

public Object handle(Message msg) {
Logger.getInstance().showDebug(msg.toString());
return receiveNotification((DistributedJascoMessage) msg.getObject());
}

/**
 * @return Returns the connSet.
 */
public HashSet getConnSet() {
return connSet;
}

/**
 * @param connSet
 *            The connSet to set.
 */
public void setConnSet(HashSet connSet) {
this.connSet = connSet;
}

/**
 * Return true if the connector is loaded in the local jasco runtime
 */
public boolean isLocal(Connector c) {
return !connSet.contains(c.getClass().getName());
}
}
```

Bibliography

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2005.

[AFT99] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[Ban05] Bela Ban. Jgroups - a toolkit for reliable multicast communication. Published online, URI http://www.jgroups.org, 2005.

[Bor01] Jerry Bortvedt. Jsr 107: Jcache - java temporary caching api. Pulished online. URI: http://jcp.org/en/jsr/detail?id=107, 2001.

[BW05] Bela Ban and Ben Wang. *JBossCache Reference Manual V. 1.2*. JBoss Inc, 2005.

[CGPV97] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 93–110, London, UK, 1997. Springer-Verlag.

[Chi04] Shigeru Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, January 2004.

[CMT04] Denis Caromel, Luis Mateu, and Eric Tanter. Sequential object monitors. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming,18th European Conference*, number 3086, pages 316–340, Oslo, Norway, June 2004.

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn THór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 330–341. Morgan Kaufmann, 1996.

[DFL+05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[DT04] Rémi Douence and Luc Teboul. A pointcut language for control-flow. In *Proc. of the 3nd Int. Conf. on Generative Programming and Component Engineering (GPCE'04)*, volume 3286 of *LNCS*, pages 95–114. Springer-Verlag, 2004.

[GJL05] Bill Gallagher, Dean Jacobs, and Anno Langen. A high-performance, transactional filestore for application servers. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 868–872, New York, NY, USA, 2005. ACM Press.

[GK97] Anurag Mendhekar Chris Maeda Cristina Videira Lopes Jean-Marc Loingtier John Irwin Gregor Kiczales, John Lamping. Aspect-oriented programming. *The European Conference on Object-Oriented Programming (ECOOP)*, LNCS 124, June 1997.

[GKG01] Jim Hugunin Mik Kersten jeffrey Palm Gregor Kiczales, Eric Hilsdale and William G. Griswold. An overview of aspectj. *ECOOP*, 2001.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Jac03] Dean Jacobs. Distributed computing with bea weblogic server. In *CIDR*, 2003.

[JD05a] Tom Van Cutsem Sebastian Gonzales Wolfgang De Meuter-Theo D'Hondt Jessie Dedecker, Stijn Mostinckx. Ambient-oriented programming in ambienttalk. In *Demostration OOPSLA*, 2005.

[JD05b] Tom Van Cutsem Wolfgang De Meuter Theo D'Hondt Jessie Dedecker, Stijn Mostinckx. Ambient-oriented programming. Vrije Universiteit Brussel, 2005.

[KG02] Jörg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto, CA, USA, February 1997.

[Lop97] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, November 1997.

[MN04] Michiaki Tatsubori Muga Nishizawa, Shigeru Chiba. Remote pointcut - a language construct for distributed aop. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, 2004.

[Pap79] C. H. Papadimitriou. SERIALIZABILITY OF CONCURRENT DATA BASE UPDATES. Technical Report MIT/LCS/TR-210, 1979.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[RD00] Qun Ren and Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 210–221, New York, NY, USA, 2000. ACM Press.

[RD01] Mario Südholt Rémi Douence, Olivier Motelet. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns, LNCS, long version is Technical Report no. 01-3-INFO*, 2001.

[RD02] M. Südholt R. Douence. A model and a tool for event-based aspect-oriented programming (eaop). *french version accepted at LMO 03*, December 2002. 2nd edition,.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190, New York, NY, USA, 2002. ACM Press.

[SM] Inc. Sun Microsystems. Using custom socket factories with java rmi. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/socketfactory/index.html.

[SVJ03] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[TS02] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning, 2002.

[TSCI01] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" java software. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 236–255, London, UK, 2001. Springer-Verlag.

[VS] Wim Vanderperren and Davy Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. pages 120–134.

[Wu05] Pengcheng Wu. Efficient and expressive aspect-oriented programming languages. Technical report, Published online. URL: https://lists.ccs.neu.edu/pipermail/colloq/2005/000210.html, July 2005.

[WV04] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT '04/FSE-12: Proceedings of*

the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pages 159–169, New York, NY, USA, 2004. ACM Press.

[WV05] Maria Agustina Cibran Bruno De Fraine Wim Vanderperren, Davy Suvee. Stateful aspects in jasco. The European Joint Conferences on Theory and Practice of Software, 2005.

[WVD01] Bart De Win, Bart Vanhaute, and Bart De Decker. Security through aspect-oriented programming. In *Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security*, pages 125–138, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.