Comparison of Two Frameworks for Parallel Computing in Java and AspectJ

João L. Sobral
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga
PORTUGAL
jls@di.uminho.pt

Miguel P. Monteiro
Departamento de Informática
Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
2829-516 Caparica
PORTUGAL
mmonteiro@di.uminho.pt

Carlos A. Cunha
Escola Superior de Tecnologia
Instit. Politécnico de Viseu
Campus de Repeses
3504-510 Viseu
PORTUGAL
cacunha@di.estv.ipv.pt

ABSTRACT
This report presents an AspectJ framework for parallel computing and compares it with a Java framework providing equivalent functionality (concurrency/parallelization, distribution, profiling and optimizations). We detect several relative benefits in the AspectJ implementation, namely greater levels of uncoupling among framework features, a greater level of obliviousness from framework code (avoidance of adapters and concern specific hooks) and possibility of framework features to be used stand alone. The downsides are that composition of aspects can be tricky, which has a strong influence of the overall framework design. Generation of source code for some features remains a convenient implementation technique. AspectJ avoids it in more cases than in the Java version, but not in all.

Keywords
Aspect-oriented frameworks, AspectJ, parallel programming

1. INTRODUCTION
Most reported aspect-oriented frameworks fall into two categories: (1) existing object-oriented (OO) frameworks that were extended from a certain point on with additional functionality by means of aspect technology [1][2] and (2) existing OO frameworks, in which various crosscutting concerns were identified and extracted to aspects [11]. In both cases, the original, OO architecture is kept largely in place, with no significant redesign. Such relatively minor tweaking risks missing the full benefits that aspect-oriented programming (AOP) can bring to framework design.

We believe that fully leveraged AOP can yield simpler, less coupled designs than those that can be obtained through plain OO technology. Presently, aspect-oriented frameworks fully developed from the ground up with aspect technology are virtually non-existent. It is desirable that such frameworks be reported to the research community, as they can provide a clearer picture of the implications of AOP on framework design, as well as provide a means to better characterize and assess its advantages over traditional, OO frameworks [14]. To date, this area of research remains largely unexplored.

In this report, we contribute to the understanding of AOP frameworks by describing and comparing two frameworks for parallel programming that were separately developed, using Java and AspectJ technology, respectively. We provide a comparative analysis of both systems and report on various hurdles we felt during development that have a bearing on the evolution of AspectJ systems. To organize the analysis, we use the majority of the 13 criteria proposed in [35] for frameworks in field of parallel computing.

The focus of this report is on frameworks developed *with* AOP technology, not frameworks whose purpose is provide support *for* AOP as an alternative to AOP languages, as is the case with [7]. In addition, the comparison and analysis provided in this report is tailored to the specific field of parallel computing. However, we believe that many of our findings can be beneficial to other domains.

The rest of this report is structured as follows. Section 2 presents an overview of the functionality provided by both frameworks and describes how this functionality is implemented in Java and AspectJ. In section 3, we compare both systems on the basis of the 13 criteria proposed in [35]. Section 4 compares this work against other efforts and section 5 presents future work. Section 6 concludes the report.

2. FRAMEWORKS FOR PARALLEL PROGRAMING
In our previous work, we developed a collection of reusable abstract aspects, coded in AspectJ, that in practice comprise an AOP framework for concurrency [8]. In addition, we developed a collection of pluggable aspects that can help the programmer to convert a sequential application into a parallel equivalent [36]. In earlier work [15][37][38], we developed equivalent functionality using traditional OO framework (coding in C++, Java and C#).

Our previously implemented OO frameworks for parallel computing (i.e., C++ and C# [37][38]) include support for object distribution and automatic optimizations. The latter aim to relieve the programmer from manual optimizing work associated to specific architectures. The goal is to obtain code that is more platform-independent without losing efficiency across a wide range of platforms. The Java implementation [15] is the most recent and complete OO implementation and benefited from the experience gained in developing the previous (C++ and C#) ones. It provides all the features previously implemented, plus an additional feature, based on parallel skeletons [6] (see 2.1), which helps the programmer to structure parallel applications.

Previous OO framework implementations suffer from classic tangling problems as concurrency/parallelization, distribution and optimization concerns cut across multiple framework components.

One of our aims in developing an AspectJ implementation of the previous frameworks was to avoid this tangling, providing the complete set of functionalities in a way that is also easier to use, maintain and evolve. AspectJ was selected due to its wide acceptance, maturity and tool support, as well as for its support being based on static weaving, as parallel computing is a performance-centric field that requires the generation of efficient executables.

2.1 Framework overview

The purpose of all frameworks covered in this report is to ease development of parallel applications by providing the basic support infrastructure for parallel programs. Such infrastructure is implemented through *skeleton composition*. The term *skeleton* [6][9][32] is widely used by the parallel computing community – a skeleton implements a common parallelization mechanism and encapsulates design decisions concerning the structure of a parallel application. Skeletons are akin to design patterns [17], though the term is generally used in the context of parallel programming and is more low level, as a skeleton is generally associated to some concrete implementation. In this context, we regard specific implementations of design patterns, including AspectJ aspects, to be instances of skeletons. To develop a parallel application, the programmer selects a set of skeletons that best fits application requirements and fills the hooks provided by the skeletons with domain specific code. Usually, it must also develop new code to instantiate the selected skeletons and to start skeleton activity, though in same cases the instantiation code can be automatically generated.

Several well-known skeletons exist from some time [6][9]. These include *Farm*, *Pipe*, *Divide/Conquer* and *Heartbeat*. One important feature of skeleton approaches is the ability to *compose* skeletons [10] – either to achieve a more efficient execution or to obtain more complex parallelizations. For instance, a *Farm* can be combined with a *Pipe* to yield a Pipeline of Farming (a Pipe in which each element is a Farm). Another example is the composition of two Farms to yield a two-level Farm. This type of structure closely matches an architecture composed by several machines (i.e., a cluster) in which each node is composed by multi-core processors.

Distribution is an important concern that, due to its nature, must be considered early in the design of the framework. Distribution concerns include remote creation of objects, remote method invocation and access to distributed data structures. Each of the framework skeletons must be suitably structured so that they can be deployed in distributed machines. The framework must provide efficient implementations of each skeleton on shared memory machines (e.g., multi-core) as well as on distributed memory machines (e.g., clusters).

In all frameworks discussed in this report, distribution stands apart from the other features in that it is implemented through code-generation techniques rather than skeletons. Thus, we avoid the need to provide distribution-specific hooks, as well as providing a more efficient implementation – distribution operations are inlined into the source.

Performance and scalability to a large numbers of processing resources are fundamental concerns in all parallel applications. We address the scalability issue by supporting fine-grained parallelism and by incorporating mechanisms into the framework that reduce the excess of parallelism whenever necessary. Thus, two mechanisms are used to control parallelism grain-size: *computation agglomeration* and *communication aggregation*. Computation agglomeration combines parallel tasks into larger tasks by executing inter-object method calls synchronously. Communication aggregation aggregates messages by (delaying and) combining several inter-object method calls into a single call message. Implementations of these mechanisms require the gathering of application execution profile during run-time.

2.1.1 Farm skeleton

For illustration purposes, in this report, we use the *Farm* skeleton, one simple and popular parallelization mechanism. The *Farm* skeleton comprises a master entity and multiple workers (Figure 1). The master decomposes the input data in smaller independent data pieces and sends a piece to each worker. After processing the data, the workers send their partial results back to the master, which merges them to yield the final result.

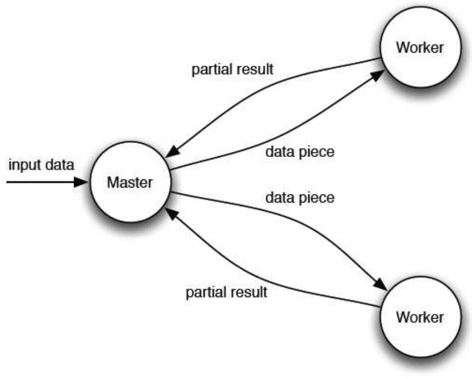

Figure 1: Farm skeleton

A farm skeleton risks being marred by parallelism overheads in cases the task grain-size proves to be too small. Such overheads are due to communication costs and thread/process management. The solution lies in mechanisms to reduce excessive parallelism. A significant gain can be accomplished by incorporating a mechanism that automatically tunes the grain-size of tasks and the number of workers to use on each platform. Automation frees the developer from dealing with these concerns directly.

A single master can be a bottleneck in the presence of a large number of workers (i.e., computing resources). Composition of farm skeletons can address this issue as well, as a farm skeleton can use several masters to improve performance (e.g., by yielding a two level farm).

2.2 Java implementation

Development of the Java framework (JaSkel, see [15]) relied on 3 independent techniques/tools. This decomposition was motivated by the requirement that use of the different bits of functionality should be possible in a broad range of contexts. These tools are:

1. A skeleton library based on Java classes structured according to the *template method* pattern [17];
2. A source code generator which supports distribution of selected object classes;
3. A run-time system that performs adaptive grain-size control and run-time load and data scheduler.

The independence between these tools allows programmers to develop, test and run structured applications in a non-distributed environment, by using the skeleton library. It also allows the use of the distribution generation tool as a stand-alone tool to generate distributed applications on the basis of sequential Java code, or combine this tool with the skeleton library to yield structured parallel applications that run on distributed systems. The run-time system is an additional tool that collects run-time execution profile information and performs run-time optimizations to adapt the application to specific platforms. We chose to provide this functionality as an additional tool to avoid execution overheads, when grain-size control is not required (e.g., when the programmer is in charge of this task or when the application does note require this feature).

2.2.1 Skeleton library

The JaSkel framework includes several common skeletons for parallel computing. We will focus on the implementation of the *Farm* skeleton (Figure 2) to illustrate how skeletons are implemented in this framework. In JaSkel, skeleton composition is supported by means of OO composition and polymorphism: the *Farm* class also extends the *Compute* abstract class (see Figure 2). Thus, it is possible to build a farm where each worker is also a farm.

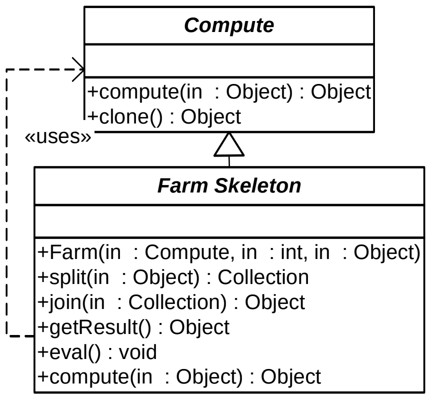
Compute
+compute(in : Object) : Object
+clone() : Object
«uses»
Farm Skeleton
+Farm(in : Compute, in : int, in : Object)
+split(in : Object) : Collection
+join(in : Collection) : Object
+getResult() : Object
+eval() : void
+compute(in : Object) : Object

Figure 2: JaSkel farm skeleton

The farm constructor gets a reference for a cloneable *Compute* worker, the number of workers (an optional parameter) and the initial data to process. Methods *split* and *join* are hooks to plug domain specific code. These methods perform the partition of the input data into pieces that can be processed in parallel and join the collection of processed data pieces. The *eval* method starts the skeleton activity. It calls the *split* method to get a collection of pieces of data, calls the *compute* method on each worker to process each datum and calls the *join* method to merge the processed data. method *getResult* provides access to the processed data. Methods *eval* and *getResult* are separate methods to allow other tasks to execute while the farm is computing (i.e., executing the method *eval*). Figure 3 presents a simple farm in JaSkel.

The JaSkel *Farm* class does not include concurrency related code. *FarmConcurrent* provides this functionality by extending a *Farm*, overriding *eval* and *getResult* methods to perform concurrent calls to workers *compute* methods. The *eval* method spawns a thread per worker to call the *compute* method and the *getResult* method waits until all workers complete their tasks. Extending the *Farm* to a *FarmConcurrent* requires complete new implementations of methods *eval* and *getResult*. This is due to the limitation of the Java (as well as most other OO languages) to effectively modularize concurrency related code. Code in Figure 3 can extend the *FarmConcurrent* instead of *Farm* to use a concurrent farm.

```
public class Worker extends Compute {
        ... // other local data

        public Object compute(Object obj) {
             return(/* processed obj */);
        }
}
class Farmer extends Farm {

        public Collection split(Object initialTask) {
             return(/* split initialTask */);
        }
        public Object join(Collection partialResults) {
             return(/*merge partialResults*/);
        }
}
public class Main {
        public static void main(String[] args) {

             Worker worker = new Worker();
             Object task = ... // new task to process
             Farmer f = new Farmer(worker, numberOfWorkers, task);
             f.eval();
             ... // other processing may be included here
             Object result = f.getResult();
        }
}
```

Figure 3: Simple farm in JaSkel

The *Farm* skeleton is used mostly in the initial development stages as it avoids the introduction of concurrent execution in the farm (and the consequent non-deterministic behaviour). This provides an easier way to trace an incorrect behaviour either to sequential or to concurrency code.

2.2.2 Distribution tool

Although distribution could be implemented by extending skeletons to address distribution concerns, as in [3][18], we opted for a tool that generates source code. This brings several advantages: (1) the distribution tool can be used stand alone, which broadens the range of applications to cover applications that do not rely on skeletons, (2) distribution does not have to be included into the application if it is not needed, (3) explicit hooks to support distribution are avoided and (4) applications that do not use distribution do not incur any extra run-time overhead.

The purpose of the distribution tool is to support object distribution among multiple JVMs. A remote object is an object that may reside in another Java Virtual Machine. A system that transparently distributes objects among several JVM is known as a distributed JVM. A tool that implements a distributed JVM must provide three basic services: remote object creation, remote method invocation and access to remote data. Our tool is based on a well known process [31][37][38] that performs a source code transformation. It is based on 3 classes: proxy objects (PO), implementation objects (IO) and object managers (OM). The code generator analyses source classes retrieving information about each class interface. Each class is renamed to an IO class and a new PO class with the same interface as the original class transparently replaces it. Each node has an OM that implements local object factories to enable remote object creations. A similar strategy implemented through a bytecode rewriter is presented in [12].

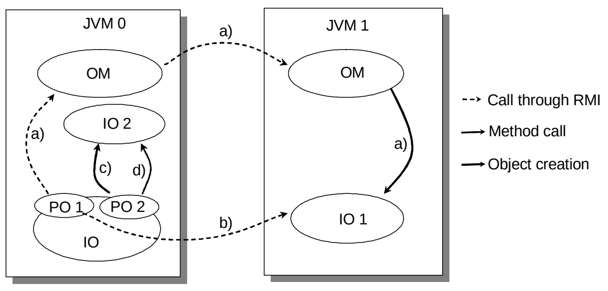

Figure 4: Run-time system for object distribution

Figure 4 presents an example of how PO, IO and OM collaborate to implement remote object creation and remote method calls. Whenever an object was created in the original code a new PO object is created instead. This PO requests the IO creation to the local node OM (JVM 0, *call a)* in the figure), which may locally create the IO object or forward the request to a remote OM, which locally creates the requested object (example shown in the figure, *call a)*). After remote object creation the PO transparently redirects local method calls to the remote IO (*call b)* in the figure).

```
public class Server {
      public void process(int[] num) {
            ... // method implementation
      }
}
```

Figure 5: Server class

```
01  public interface IServer {
02        public void process(int[] num) throws RemoteException;
03  }
04
05  public class ImplementionServer implements IServer { // IO class
06        public void process(int[] num) {
07              ... // original method implementation
08        }
09  }
10
11  public class ObjectManager {              // OM class
12        IServer factoryServer() {
13              return( new ImplementationServer() );
14        }     ...   // register server as OM is externally visible
15  }
16
17  public class Server {        // PO class
18        IServer myRemoteServer;
19
20        Server() {          // request remote object creation
21              ObjectManager remoteOM = ... // get reference to OM
22              myRemoteServer = remoteOM.factoryServer();
23        }
24        public void process(int[] num) {  // remote method invocation
25              myRemoteServer.process(num);
26        }
27  }
```

Figure 6: Generated code for simple Server class

Figure 6 presents a simplified example of Java RMI generated code for a *Server* class (Figure 5). The interface *IServer* (lines 01-03) is created due to Java RMI requirements. The original *Server* class is rewritten to *ImplementationServer* class (lines 05-09). The *ObjectManager* is an object that implements a remote *ImplementationServer* factory (lines 11-15). The original Server class is replaced by a PO (lines 17-27) that transparently requests the remote object creation to OMs (lines 20-23) and redirects the *process* calls for remote execution (lines 24-26).

2.2.3 Run-time system

The run time system is in charge of performing load distribution by selecting the most adequate JVM for the creation of each object. It also performs several optimizations to support grain-size control of parallel tasks.

```
public class Server {     // PO class
   IServer myRemoteServer;
   ImplementionServer myLocalServer;

   Server() {
      if ( agglomerateComputation() ) {    // locally create server object
         myLocalServer = new ImplementationServer();
      } else {
         ObjectManager remoteOM = ... // get a reference a remote OM
         myRemoteServer = remoteOM.factoryServer();
      }
   }
   public void process(int[] num) { // performs local/ remote invocation
      if (agglomerateComputation() {
         myLocalServer.process(num);
      } else {
         myRemoteServer.process(num);
      }
   }
}
```

Figure 7: Generated code for computation agglomeration

```
public class ImplementionServer implements IServer { // IO class
   ...
   public void processN(Vector args) { // performs N calls to process
      for(int i=0; i<args.size(); i++) {
         process((int[]) Vector.elementAt(i) )
      }
   }
}
public class Server { // PO class
   ... //
   Vector args = new Vector();

   public void process(int[] num) {
      if (agglomerateComputation() ) {
         myLocalServer.process(num);
      } else {
         If ( aggregateComunication() ) {
            args.add(num);
            if ( args.size() == callsPerMessageLimit ) {
               myRemoteServer.processN(args);
               args.clear();
            }
         } else myRemoteServer.process(num);
      }
   }
}
```

Figure 8: Generated code for communication aggregation

The first optimization is computation aggregation, which locally creates an object, without OM involvement. This is implemented by allowing the PO object to directly create an IO without performing a request to the local OM (*call c)* in Figure 4). Latter calls to this object are also performed directly, as in standard Java objects (*call d)* in Figure 4). Java RMI also performs a similar optimization in RMI calls to local objects.

The implementation of local object creation requires minor changes to the distribution tool. It mainly involves changes into the PO generated code to implement local IO creation and to implement direct calls this local IO. Figure 7 presents these changes in shaded. The generated code includes a test to determine when to apply agglomeration. In those cases it performs direct IO creations and method calls to *myLocalServer*.

Communication aggregation aims to reduce communication costs by packing several method calls into a single network message (a similar functionality is provided by ARMI [39]). This optimization can take advantage of the increase in network bandwidth (e.g., 10 Gbit/s Ethernet) and reduce penalties due to network latencies, by using a smaller number of larger messages. The framework implements communication aggregation by packing several remote method calls, performed on the same target object, into a single method. This is implemented by introducing a new method into the IO class and by calling this method in the PO class (Figure 8 in shaded).

2.2.4 *Limitations of the OO implementations*

Experience acquired during several years of evolution of the parallel computing framework exposed several weaknesses of this type of approach, most of which are related to framework limitations and excessive complexity to develop and maintain the proposed tools. The first limitation is related to the introduction of concurrency concerns into JaSkel skeletons. These concerns are introduced by providing skeleton implementations that support concurrent execution. Current implementation overrides specific skeleton methods, leading to a considerable increase in code size and harming skeleton implementation understandability. A serious problem is that skeleton base classes must be designed upfront (e.g., to include hooks) to allow concurrent execution to be composed through inheritance. Another limitation is that it is not possible to use concurrency as stand alone framework feature. This could be overcome through the use of an external tool to generate concurrency related code. However, this would increase the complexity of the tools that generate source code.

Generative patterns are an alternative implementation strategy to the skeleton library. Generating pattern code can avoid method overriding as application specific code can be included by modifying the implementation of specific methods, supports more flexible skeleton configurability (by means of template parameters) and can also generate concurrency related code. However, this strategy has two limitations: it introduces another source code generator and the changes to the generated code are non-reversible, which can lead to problems when, for some reason, the pattern code must be re-generated.

Implementation of the distribution tool requires the use of a parser generator tool that analyses source code and generates new case specific classes. The tool must traverse the abstract syntax tree (built by the parser) several times to generate PO, IO and OM classes. A similar strategy is required to implement the run-time system (including generation of code to tune grain size of parallel tasks) and to implement code to collect execution profiling.

These code generator tools are independent from the user point of view. However, at the level of the framework implementation, they are tightly coupled. Programmers extending the code generator tools must learn how to parse Java code and how to traverse the generated syntax tree to generate the required code. It also requires knowledge of all the generated code (e.g., run-time system implementations must be aware of minute details related to distribution generated code, see Figure 8). To add support for a new feature, the programmer must identify and edit all points where code related to new features should be placed. One of the main sources of complexity is the generation of tangled code that mixes distribution, profiling and optimization concerns. This imposes a limitation to independent development (evolution of the distribution code and run-time system). Code generation tools also have problems to trace errors into the original code since they rewrite the original code to new classes. This makes it hard to debug generated code and to trace errors to the corresponding location in the original code.

A minor problem arose in the Java implementation. Java does not include multi-inheritance, which makes it more complex to generate RMI code. The application specific classes must extend both skeleton classes and RMI classes.

In summary, introducing new features requires careful planning to ensure non interference with current framework features that requires knowledge about source code generation and about the structure of generated code. This is a consequence of the generation of tangled code, where distribution, profiling and optimization concerns are mixed in the generated code. Using code generation tools instead of libraries pushes the tangling effect from framework template classes to the generated code.

2.3 AspectJ implementation

The AspectJ implementation framework combines libraries of reusable aspects [8][20] with generation of AspectJ source code [5]. AspectJ enables the modularization of more framework features than its Java counterpart. However, some framework features are not conveniently implemented by a library of reusable aspects. In some cases, that would entail extensive use of reflection and in others a library based approach is simply not feasible. For instance, distribution related code requires the remote creation of an object, implemented by a remote object factory. The factory must be generated, since it is not possible to remotely create objects in AspectJ.

A library of reusable aspects brings the same benefits as a tool that generates source code to implement a specific framework feature. In cases where an approach based on a library is not feasible, generation of AspectJ (rather than Java) code yields a solution that is more modular. Generation of AspectJ code can also have a performance advantage over reusable aspects, as it avoids two of the three main costs of an aspect library: retrieval of joinpoint context and management of global joinpoint history. See [8] for a discussion of the performance costs involved in a library of reusable aspects. The source code generator can generate an aspect tailored for the specific case (for instance, Figure 12 presents an aspect generated specifically for the *Server* class).

2.3.1 *AspectJ Skeletons*

An important issue is how to provide functionality similar to a farm skeleton (Figure 2 and Figure 3) in AspectJ. The skeleton is based on a Compute class and a Farm class. Our experience with the JaSkel framework revealed that the Farm class usually has a *defining* role [20] in the pattern (i.e., the class only exists in the context of the pattern and must be written from scratch), as it plays a role of interfacing between domain specific code and the skeleton implementation. On the other hand, Compute has a *superimposed* role, as it usually results from an adaptation of a class that already plays a role in the domain specific code.

```
public abstract aspect Farm {
   public abstract Collection split(Object initialTask);
   public abstract Object join(Collection partialResults);

   public abstract pointcut computeCall(Object task, Object tgt);
   public abstract pointcut objectCreation();

   Object around() : objectCreation() {
      workers = … // clone target object numberOfWorkers times
   }
   Object around(Object initialTask,Object target) :
         computeCall(initialTask,target) {
      workers = … // get array of workers of target object
      Collection tasks = split(initialTask);
      Iterator i = tasks.iterator();
      Collection oTasks = new Vector();
      int taskId = 0;

      while (i.hasNext()) {
         int workerIndex = taskId % numberOfWorkers;
         oTasks.add( proceed(i.next(), workers[workerIndex],) );
         i.remove();
         taskId++;
      }
      return(join(oTasks));
   }
}
```

Figure 9: Farm AOP skeleton

```
public aspect Farmer extends Farm {
   public Collection split(Object initialTask) {
      return(/* split initialTask */);
   }

   public Object join(Collection partialResults) {
      return(/*merge partialResults*/);
   }

   public pointcut objectCreation() :
      call (SomeCoreClass.new()) && within(Main);

   public pointcut computeCall(Object initialTask, Object tgt) :
      call(* SomeCoreClass.someMethodCall(..)) &&
      args(initialTask) && target(tgt) && within(Main);
}

public class Main {
   public static void main(String[] args) {
      SomeCoreClass worker = new SomeCoreClass();
      Object task = ... // new task to process
      Object result = worker.someMethodCall(task);
   }
}
```

Figure 10: Sample use of a farm AOP

In light of the above considerations, the Farm mechanism is implemented by an abstract aspect, with abstract methods to provide *split* and *join* functionality. The Compute functionality is implemented by adapting a class from the domain specific code to which the farm parallelization applies. The skeleton *compute* method becomes becomes an abstract pointcut that indicates the point in the core functionality where the skeleton activity should start. Association between the class that implements the Compute functionality and the farm aspect is provided by another pointcut that captures the joinpoints in which instances of that class are created. Whenever the domain specific code does not expose joinpoints suitable to superimpose the Compute role, an approach similar to JaSkel must be used (i.e., a class Compute must be created). Figure 9 presents a sketch of the AspectJ implementation of farm.

In the advice associated to pointcut *objectCreation*, the farm aspect clones an object (*numberOfWorkers* times). In the advice associated to pointcut *computeCall*, the aspect calls method *split* to partition the original data into pieces that are sent to workers. Partial results are merged by calling method *join*.

Application of the farm AspectJ skeleton to a specific case entails the implementation of *split* and *joins* methods by a concrete aspect, as well as the specification of pointcuts *computeCall* and *objectCreation* (see example in Figure 10). In AspectJ, composition of skeletons is a bit tricky, as it requires another aspect to capture *proceed* calls performed in an advice, something that is not currently covered by the AspectJ joinpoint model. This was implemented by resorting to *Chained Advice* idiom [19] to allow a reusable aspect to explicitly expose this joinpoint.

Implementation of a concurrent farm entails creating a thread per each call to a worker object, to obtain concurrent processing. This must be performed in advice acting on pointcut *computeCall*. Extending the *Farm* abstract aspect to yield a concurrent farm is not feasible in AspectJ as it is not possible to override an advice. The alternative and more effective way is to provide an additional aspect to plug concurrency into the farm skeleton. In [8], we presented an AspectJ collection of concurrency patterns and mechanisms. This collection includes the *FutureReflectProtocol* aspect that can be plugged into the *Farm* aspect to yield a concurrent farm. *FutureReflectProtocol* requires the definition of two pointcuts: *futureMethodExecution* and *useOfFuture*. The former defines the points where the computation methods are invoked and the latter defines the joinpoints where the result of the computation is needed. . Unfortunately it is also too tricky to plug this functionality directly into the Farm reusable aspect, again due to AspectJ limitations, as now we need to capture two different joinpoints (one is the proceed, the second requires an explicit joinpoint inside the abstract join method). Figure 11 presents an outline of an aspect that achieves a concurrent farm but it advises joinpoints from code functionality (i.e., it is a case specific aspect).We can also use aspect *FutureReflectProtocol* to provide functionally similar to methods *eval* and *getResult* (i.e., to perform other processing while the farm is computing, see section 2.2.1)

```
public aspect ServerConcurrency extends FutureReflectProtocol {

      protected pointcut methodCall(Object servant) :
            call(* SomeCoreClass.someMethodCall (..)) &&
            target(servant);

      protected pointcut useValuePoint(Object servant) :
            call(/* some join specific hook*/) &&
            withincode(Object Farmer.join(Collection)) &&
            target(servant);
}
```

Figure 11: Introduction of concurrency into farm skeleton

There are two important advantages of AspectJ implementations relative to Java. First, concurrency can be used as a stand alone framework feature (our concurrency library was previously presented and applied to several cases). Second, it avoids code duplication present in Java implementation (and other OO), since concurrency can be plugged into the Farm without rewriting aspect code.

2.3.2 *AspectJ distribution*

In the OO distribution tool we introduced proxy objects, implementation objects and object managers (Figure 4) that collaborate to transparently implement object distribution. Proxy objects and object managers play defining roles in this collaboration, as they are used only in the context of distribution. On the other hand, distribution-aware objects (i.e., implementation objects) that provide domain-specific functionality are generated by the tool, based on the original classes. Thus, they have a superimposed role in the context of distribution.

Following a procedure similar to that of the previous section, the proxy object becomes an aspect that must intercept all domain-specific object creations and method calls, to make a class separate (i.e., distributed). Proxy object is the core of the distribution concern as it implements the code to transparently distribute objects to remote machines. Implementation objects are no longer required, as we can use an aspect to superimpose that role into domain specific classes. Object managers have a defining role in the distribution context. However, these are not visible outside the distribution aspect, which is why these objects were implemented as plain Java classes.

Implementation of distribution using Java RMI requires the generation of one interface per each remote class. We also need to create a remote object factory (i.e., object manager) to implement remote creation of objects. Thus, we still need to use a source code parser and a source code generator. Figure 12 presents the code equivalent to Figure 6 in the AspectJ implementation. The most significant differences between these versions are shaded.

```
01  public interface IServer {
02        public void process(int[] num) throws RemoteException;
03  }
04
05  declare parents Server implements IServer;    // replaces OM class
06
07  public class ObjectManager {       // OM class
08        IServer factoryServer() {
09              return( new Server() );
10        }
11  }
12
13  public aspect ServerDistribution {       // replaces PO class
14        IServer myRemoteServer;
15
16        Object around() : call (Server.new(..)) && /* … */ {
17              ObjectManager remoteOM = ... // get a reference to OM
18              myRemoteServer = remoteOM.factoryServer();
19              return new Server(); // return a fake local
20        }
21        void around(int[] num): call(void Server.process(..))
                                  && args(num) && !within(*Distribution)) {
22              myRemoteServer.process(num);
23        }
24  }
```

Figure 12: Generated AspectJ code for simple Server class

The code of the implementation object was simply replaced by an AspectJ intertype declaration (line 5). We still need to generate code for the *IServer* interface and for *ObjectManager* class. Proxy objects were replaced by an aspect that intercepts local calls and forwards them to the remote object (lines 16–23).

In spite of still requiring a source code generation tool, the AspectJ implementation has the advantage of generating modularized distribution code. As such, we do not need to change any of the class implementations from the domain specific code.

2.3.3 *AspectJ run-time*

The run-time system implements application profiling, computation agglomeration and communication aggregation. In this section, we do not provide details concerning profiling since there are several well known AspectJ implementations [22]. The point is that profiling can be provided as a library of reusable aspects that can also be used in a stand alone way, outside the context of the framework's run-time system.

Computation agglomeration is based on local creation of specific object instances instead of requesting these creations to remote JVMs (see Figure 7). To implement this feature the distribution aspect should not be applied when the object is intended to be created locally (e.g., to locally create a *Server* object we just need to avoid the execution of both around advices in Figure 12, lines 16-23). Thus, we can generate an aspect that executes on the same joinpoints as the distribution aspect, but that has higher precedence and this way it can avoid the execution of the distribution related advices of Figure 12 (see Figure 13). In this framework is not a problem for an aspect to perform proceed only in some cases, since the distribution aspect never proceeds the original call. However, it can pose problems in other cases if we need to have an aspect with lower precedence. A similar problem was reported when trying to compose a cache aspect with a cache profiling aspect [26], as the cache aspect does not always perform a proceed.

```
public aspect ServerAgglomeration {
   Object around() : call (Server.new(..)) {
      if ( ! agglomerateComputation() ) {
         return(proceed()); // proceed to distribution concern
      } else {
         return( new Server() ); // avoids distribution concern
      }
   … //
```

Figure 13: Alternative implementation of agglomeration

Communication aggregation must generate an aspect that also specifically acts on the distribution aspect. It must introduce the method processN into IServer and Server class. It also must overwrite implementation of both distributions advices.

2.3.4 *Discussion of AspectJ implementation*

Although AspectJ implementations attain greater levels of modularity, some hurdles are felt when trying to compose these aspects together. Aspect composition is an important issue in the design of the framework, as features can be plugged into the same core functionality (e.g., they must share joinpoints in domain specific code). One solution to this composition problem is to provide *anchor pointcuts* that are captured by multiple aspects. Aspect precedence is another important issue. Advices must execute in a specific order: farm, concurrency, computation agglomeration, communication aggregating and distribution. To ensure this order, all advices are of type *around*, since AspectJ implements different execution orders for before, around and after advices.

Some aspects depend on other aspects, which require the design of specific hooks in aspect code to support aspect composition (e.g., composition of concurrency into the farm aspect). Some framework features simply can not be implemented by a library of reusable aspects (e.g., communication aggregation requires the introduction of a new method in the target class). In other features, we resorted to code generation to avoid the use of refection and to provide maximum efficiency (e.g., in Figure 12 we could provide a single advice to intercept all calls, as proposed in [34]). Framework extension hooks must be designed upfront, just as with OO frameworks (by providing abstract methods and abstract pointcuts). This is a consequence of limitations in AspectJ's joinpoint model as regards aspect-specific joinpoints. In addition, in AspectJ we can extend abstract aspects but cannot override advice implementation, which is proving to be an important constraint to framework evolution, since evolution paths must be anticipated.

Even if AspectJ does not fully support independent development of all framework features (e.g., the case with computation agglomeration) we nevertheless think that AspectJ enables a higher level reasoning about features than when code is tangled (as is the case with OO). To understand this AspectJ framework, it is vital to understand the collaborations among aspects. Since joinpoints in domain-specific code are captured by multiple aspects in a very specific order, understanding and extension of this framework requires a grasp of this advice execution chain. To correctly compose new functionality into the framework, new aspects must be plugged in the correct points of this advice execution chain.

3. DISCUSSION

We base our discussion on the set of criteria proposed in [35] which were adopted by MacDonald for an ideal pattern–based parallel programming system [24]. Conceptually, MacDonald treats patterns as modular units, in a way that is akin to skeletons [6] (and as we of course do with aspects), which facilitates the comparison presented in this section. In [24], MacDonald uses the following 13 criteria:

1. *Separation of specification*
2. *Hierarchical resolution of parallelism*
3. *Mutual Independence*
4. *Extendibility*
5. *Large collection of useful patterns.*
6. *Openness.*
7. *Correctness guarantees.*
8. *Commonly–used language*
9. *Language Non–Intrusiveness*
10. *Performance*
11. *Tool Support*
12. *Tool Usability*
13. *Application Portability*

In the remainder of this section, we discuss the various approaches to framework development on the basis of the aforementioned criteria and in the light of the experience gained when developing the frameworks mentioned in this report. However, as criteria 12 and 13 are out of scope of the report, they are covered. The order of the criteria is the same as above. Each of the sub-sections that follow deals with one criterion, starting with brief a description of it.

3.1 Separation of specification

MacDonald states that there should be a clean separation between the parallel structure of a program and the application code as to allow both parts of a parallel program to evolve independently. This closely corresponds to the classic tenet of separation of concerns [29], applied to specific case of parallel programming. Both approaches achieve this separation, though AspectJ enables us to go further than Java. The greater ease in uncoupling domain-specific code from framework features facilitates independent development (though there are still problems, as described in [40]). Aspects allow for a less monolithic solution. A greater independence and uncoupling between framework (and domain-specific) features can be observed (e.g., skeletons, concurrency and distribution features can be used stand alone). The use of abstract pointcut hooks instead of template method hooks amounts to an easier way to experiment with different skeletons (i.e., parallelization) as there are no explicit calls between domain specific code and framework API (e.g., it avoids the use of object factories to create objects). We consider it a consequence of the greater level of modularity and obliviousness achieved with AspectJ.

3.2 Hierarchical resolution of parallelism

This is the ability to allow patterns to be composed hierarchically, refining the computation within a given pattern using another pattern [24]. In the limit, this criterion implies the ability to compose a framework with another, something that can be hard and sometimes impossible with traditional OO. With AspectJ, this is more feasible, though it can still be hard, as we report in this report.

With OO, the Composite pattern [17] provides an elegant way to hierarchically structure solutions in many cases. However, composite structures whose elements are heterogeneous still require a common interface to all components. In the specific case of skeletons, all skeletons must provide a common interface to be hierarchically composed. This can constrain skeleton development. In JaSkel we had problems to compose a farm with a pipeline, as the pipeline interface requires an additional method to connect pipeline elements. This is not necessary with AspectJ.

We use the term *recursive pointcut* to refer to the case in which advice acting on the joinpoints captured by a pointcut give rise to new joinpoints that can be captured by pointcuts within the same aspect and/or other aspects. This process can extend to multiple levels. This kind of recursion is generally considered a bad thing, particularly that involving the same pointcut. However, we envision cases in which it may actually be desirable. For instance, to hierarchically compose AOP skeletons we need to apply the same aspect or a combination of aspects to other aspects, leading to recursive pointcuts. It is still not clear whether this solution brings more advantages than that based on the Composite pattern. Furthermore, AspectJ bears significant limitations to the quantification over aspect-originated joinpoints (e.g., it is not possible to quantify over a specific aspect advice). Further research is required on this front.

It is generally hard to compose various traditional OO frameworks into a single system. One important cause for this limitation is *inversion of control* [13]. This problem can be ameliorated by AOP as it can avoid inversion of control.

3.3 Mutual independence

According to [24], there should be no rules regarding how patterns (meaning features in this context) can be composed, i.e., patterns should be context insensitive with respect to one another. With OO, this can be partially achieved with upfront design. Our experience suggests that the same applies to AspectJ, though to a lesser extent. We conclude that the ideal of *programmer obliviousness* proposed in [16] is not (fully) realized by AspectJ.

Skeletons cannot be considered to be mutually independent in that all skeletons must be based on a common interface to enable the various compositions. Aspects are more flexible, as it is enough for them to define pointcuts that capture the relevant events, regardless of interfaces. However, although the AspectJ framework attains a greater level of uncoupling between features, it also presents problems for the composition of multiple features into a single, coherent system. Difficulties in composing multiple reusable aspects that capture the same joinpoints force us to resort to AspectJ idioms such as *anchor pointcuts* [19]. Hence the need for some upfront design, which harms aspect independence.

AOP's mechanisms for quantification and implicit calls to sections of behaviour hold the potential to deliver a greater level of separation and independence between the framework's various components, as well as between domain specific classes. This leads to the possibility that AOP frameworks may not show some of the defining characteristics of OO frameworks, such as (1) tight coupling between components, (2) inversion of control [13] and (3) pattern density [33]. For instance, in the AspectJ framework we were able superimpose multiple roles to the same domain specific class without interference (e.g., compute and remote object behavior) and each aspect could introduce its own defining roles in a way that is independent of other aspects (e.g., farm and proxy). On all three fronts, the AspectJ implementation is a step forward relative to all OO implementations mentioned in this report, included the one in Java.

3.4 Extendibility

According to [24], a user should be able to incorporate new patterns into the tool, in such a way that new patterns are indistinguishable from the ones originally supplied with the tool. This is a known problem in OO frameworks, as it is hard to deal with unanticipated extensions and modifications. AspectJ's pointcut mechanism is advantageous in this regard as it does not require explicit framework hooks. To some extent, it is possible to extend components not specifically prepared for such extension. However, there are limits to this capability due to AspectJ's joinpoint model not covering the aspect space as thoroughly as it covers the class space. For instance, it is not possible to capture joinpoint originating from a specific advice. Such difficulties motivate the use of some aspect-oriented patterns such as *Chained Advice* [19] and the express use of classes and interfaces within the advice just to expose the required joinpoints.

3.5 Large collection of useful patterns

According to [24], the supplied patterns should cover a broad range of applications. With AspectJ, we were able to broaden the applicability of various skeletons, some of which could now be used stand alone. However, further work is required to assess the extent to which AspectJ expands the applicability of components.

3.6 Openness

According to [24], the programmer should be able to access low–level mechanisms, such as the underlying message passing system, in their applications. Otherwise, the programmer is limited to developing applications that can be expressed using the available patterns.

We believe that the ability and desirability to provide access to low-level mechanisms greatly depends on how the system is structured. The full impact of aspect mechanism on such structures is not well known. The openness brought by AOP seems to be an advantage in some situations (e.g., it is easier to replace a component of the framework with another). However, we defer such study to future work.

3.7 Correctness guarantees

According to [24], a good parallel programming system should provide some correctness guarantees, for instance against occurrences of deadlock, or to ensure that the application matches the desired parallel structure, or that the correct type of data is sent and received between processes. By MacDonald's own admission, it is not likely that a tool will appear that can fully prevent users from introducing logic errors into program code or prevent the selection of inappropriate parallel structure.

The limitations of AspectJ make it ill-suited to deal with this issue effectively. This is partly due to the greater level of independence of aspects (i.e., not taking into account the effects of other aspects on the overall system), combined with poor support to the management of aspect interactions. A global view of the complete/full system is lacking.

Generally, framework components need to have some assumptions about their surrounding environment. Inversion of control provides that, as well as preventing various kinds of client errors. For instance, the *Template Method* pattern [17] provides a rigorous mechanism to enforce rules, invariants and contracts. That ability is absent in pointcuts, which again pushes back to the client of the framework the burden of following them. This requires a permanent conscious effort, which may feel too loose and error-prone to a developer used to the discipline of traditional OO frameworks. Instantiation provides one example. OO frameworks often provide ready-made code for the instantiation of objects, guaranteeing that are coherent and well-formed. In many cases, AOP enables client code to instantiate objects as if the aspects were not present. This freedom makes it hard or impossible to generate framework instantiation code. This results in a style of programming that has a more natural feel, but makes the framework more vulnerable to badly-formed objects.

3.8 Commonly–used language

According to [24], a system would ideally use an existing, commonly–used programming language, with no modifications to either syntax or semantics. This way, users would be able to directly reuse their existing sequential code in parallel applications and the system could take advantage of expertise in an existing language.

Aspects and jointpoint models are relatively novel concepts that can pose problems to OO programmers. Clearly, use of AspectJ (as opposed of Java) goes against this recommendation. The Java system of course follows it, but the traditional composition mechanisms present serious disadvantages, as documented in this report. Frameworks like JBoss AOP and Aspectwerkz seek to have it both ways, by relying on plain Java and separating most aspect-specific specifications in separate files. Unfortunately, these are usually represented using *another* language (usually XML). All approaches have their followers and it is still not clear which approach will emerge as the winner, if any.

3.9 Language non–intrusiveness

According to [24], the application code written by a programmer should not have to accommodate the programming model provided by the system. A negative example is a message–passing library that requires the program to be restructured to accommodate the extra communication calls that need to be inserted by the user.

The above concept is very close to that of *code obliviousness* [16]. The Java approach is significantly invasive and clearly does not meet this criterion. The Java framework presented in this report compels programmers to break the various bits of functionality throughout multiple classes, according to rules dictated by the framework rather than by the characteristics of the domain-specific code.

The AspectJ approach goes a long way to meet the criterion, though it also has its limits. With AspectJ, it is often possible – though not always – to plug the aspect into the system without the need for invasive changes. However, cases arise in which prior refactoring is needed. We detect three distinct reasons to refactor: (1) to expose the desirable joinpoints, (3) to exposed needed context information and (2) to remove dependencies between distinct stages of the algorithm to be parallelized. In the latter, it may not be a realistic goal to achieve code obliviousness, as many modifications are a consequence of the nature of the algorithm. In the first two cases, AspectJ's inter-type declarations are helpful. One instance is in the use of class adapters to make a domain specific class amenable to quantification (e.g., by providing required joinpoints as well as context information [36])

3.10 Performance

According to [24], it should be possible to achieve the best possible performance for a program, subject to the selection of the parallel patterns. AOP languages such as AspectJ rely on a joinpoint model that statically resolves joinpoints in a significant number of cases. The AspectJ compiler generates bytecodes whose performance is acceptable when compared to that obtained from traditional Java compilers In addition, the reduced level of coupling between domain specific code and the parallel structure allows for an easier way to experiment with alternative parallelizations and pick the one with the best performance.

With Java, various features such as the mechanism associated to the synchronized keyword incur overheads even when the application runs in a sequential context. This is not a problem in the Java framework because additional features such as these are provided through source code generation. With AspectJ, it is possible to uncouple these features as well. For instance, it is possible to define an aspect that performs synchronization on a given set method calls so that the synchronized keyword can be absent from the base code. When the aspect is not included in the build, the system does not incur any additional overhead [8]. Similar performance benefits were reported in the context of middleware systems [42]. Use of reflection can incur significant overheads (e.g., uses of thisJoinPoint) but this problem is more acute in systems such as JAC [30] that heavily rely on reflection.

3.11 Application portability

According to [24], the system should allow applications to be ported to different architectures. The performance of a program may suffer on an inappropriate architecture, but the application should continue to run. Executable portability is not an issue in current AspectJ compilers as they generate standard JVM compatible bytecodes.

The reduced coupling between domain specific code and parallel structure in AspecJ allows for an easier way to select the parallelization with best performance for a specific platform. However, more dynamic AOP approaches that can swap aspects at run-time can be helpful to implement frameworks that automatically choose the best parallelization for a specific platform.

4. RELATED WORK

In [20], Hannemann and Kiczales present a comparison of implementations in Java and AspectJ of the Gang-of-Four patterns [17]. In our previous work [8] we presented a collection of reusable AspectJ implementations of well known concurrency patterns and mechanisms. Both works focus on how to implement each pattern in AspectJ and do not address the composition of these implementations. Work in [4] specifically analyses composition problems among patterns. In our work, composition of patterns as skeletons in the framework is essential. However, as the composition takes place in a framework context, we were able to use more case specific solutions.

Constantinides et al. [7]propose an OO framework that provides some of the services of aspect-orientation. Though it was developed with concurrent applications in mind, the authors hope that the underlying ideas are applicable to other a broader range of domains.

Several extensions to AOP where proposed to make it aware of distribution issues [28][27]. These are based on the concept of *remote pointcut*, i.e., the ability to intercept a joinpoint in a remote machine. With remote pointcuts, it is possible to execute an advice on a different machine from the one that originates the joinpoint. We did not explore this concept in our work. Furthermore, we would need a different perspective on remote pointcuts, as we would like to apply a proceed on a specific remote machine (to implement a specific object distribution strategy) and not the other way around (i.e., a remote machine to execute an advice associated to a local joinpoint). We believe that it easer to implement such a push model of advice execution than a pull method of remote advice, as it involves more localized decisions that contribute to scale parallel applications to a high number of compute resources.

Previous works provide AOP solutions to modularize distribution related concerns [34][41] and tools to generate source code for distribution concerns [5]. Other frameworks support some features through AOP [1]. JAC [30] and COOL [23] provide full AOP frameworks that also address distribution and concurrency related concerns. Our work differs from these systems in that we provide a complete AOP framework for parallel computing developed *using* AOP technology (e.g., AspectJ).

Frameworks for parallel computing built with OO technology have been proposed in [24][3][18]. Harbulot [21] was among the first to report on the use of aspects to modularize parallel structures.

5. FUTURE WORK

AOP enables client code to instantiate its objects as if the aspects were not present. This makes the framework more vulnerable to badly-formed objects. The concept of XPIs [40] may provide a contribution to overcome this limitation, by providing contracts that serve the same purpose as hooks of traditional OO frameworks. Annotations can be an alternative way to provide hooks for aspects to compose. They also serve the purpose of documenting the base code design decisions when this code becomes more oblivious of the framework context.

Dynamic proxies that were introduced in Java 1.3 remain unexplored as a means to simplify distribution related concerns. For instance, with dynamic proxies it is no longer required to generate an interface per remote object. This could significantly simplify aspects for distribution.

Aspect oriented frameworks, built with AOP technology, are a largely unexplored field of research. Longer term experience with AOP frameworks is required to fully assess its capacity for evolution. Moreover, sets of rules to refactor [25] existing OO frameworks can help to bring a broader acceptance of AOP in framework development.

6. CONCLUSION

This report presents an AspectJ framework and compared it with a Java framework that provides equivalent functionality. Both frameworks resort to source code generation to conveniently implement various framework features, namely to avoid tangling and to achieve a greater level of unplugability and obliviousness.

Use of source code generation tools to implement specific framework features is widespread, as it yields important advantages relative to traditional library OO approaches. Library implementations are generally marred by tangling problems. Code generation brings the following advantages: (1) the tool can be used stand alone and broadens the range of applicability, (2) a given feature does not have to be included when not needed, (3) explicit hooks to support the a specific feature are avoided and (4) applications not using the feature do not incur extra run-time overheads. A library of reusable aspects provides similar benefits without the need for source code analysis (see for instance [8]) In light of these findings, we conclude that it is advantageous for AOP frameworks to be structured around libraries of reusable aspects.

AspectJ avoids use of source code generation in more cases than in the Java version, by supporting specific features through an aspect library. However, some framework features cannot be fully implemented by an aspect library. In those cases, generation of AspectJ code rather than Java brings similar advantages and allows more independent development and eases framework evolution.

AspectJ allows the superimposition of multiple roles to the same domain specific class, without the target object being aware of its role in the framework context. This also allows framework roles to be implemented in a more independent way. Each aspect can introduce its own defining roles in a way that is independent of other aspects. However, we noticed interference problems when joinpoints originated by the defining roles must be captured by multiple aspects, which had to be addressed by upfront framework design and by using AspecJ idioms to support composition of specific aspects

Use of abstract pointcuts in framework design instead of template methods leads to new ways to design frameworks as they do not require explicit framework hooks. However, current AspectJ capabilities should be improved to support a more flexible composition among aspects.

7. REFERENCES

[1] *JBoss AOP*. http://jboss.com/products/aop.

[2] *Spring AOP*. http://www.springframework.org/.

[3] Aldinucci M., Danelutto M., Teti P., An advanced environment supporting structured parallel programming in Java, Future Generation Computer Systems, vol.19 n.5, Elsevier, July 2003.

[4] Cacho N., Sant'Anna C, Figueiredo E, Garcia A., Batista T, Lucena C., Composing design patterns: a scalability study of aspect-oriented programming. AOSD 2006, Bonn, Germany, March 2006.

[5] Ceccato M., Tonella P., Adding Distribution to Existing Applications by means of Aspect Oriented Programming, 4th IEEE SCAM, Chicago, USA, September 2004.

[6] Cole D., Algorithmic Skeletons: structured management of parallel computation, Pitman/MIT press, 1989.

[7] Constantinides, C. A., Bader, A., Elrad, T. H., Netinant, P., Fayad, M. E. Designing an aspect-oriented framework in an object-oriented environment. ACM Computing Surveys, 32(1es): 41 (2000).

[8] Cunha C., Sobral J., Monteiro M., Reusable Aspect-Oriented Implementation of Concurrency Patterns and Mechanisms, AOSD'06, Bonn, Germany, March 2006.

[9] Darlington J., Field, A., Harrison, P., Kelly, P., Sharp, D., Wu, Q., Parallel Programming using Skeleton Functions, 5th Conference on Parallel Architectures and Languages Europe (PARLE'93), LNCS vol. 396, Springer 1993.

[10] Darlington J., Guo Y., To H., J. Yang, Parallel Skeletons for Structured Composition, ACM PPoPP'95, Santa Clara, USA, 1995.

[11] van Deursen A., Marin M., Moonen L., AJHotDraw: A showcase for refactoring to aspects. In Linking Aspect Technology and Evolution Workshop (LATE), March 2005.

[12] Factor M., Schuster A., Shagin K., JavaSplit: a runtime for execution of monolithic Java programs on heterogenous collections of commodity workstations, IEEE Cluster Computing, Hong Kong, December 2003.

[13] Fayad M., Schmidt D., Object-Oriented Application Frameworks, Communications of the ACM, 40(10):32–38, 1997.

[14] Fayad M., Schmidt D., Johnson R., Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley 1999.

[15] Fernando J., Sobral J., Proenca A., JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing, CCGrid'2006, Singapore, May 2006.

[16] Filman R., Friedman D., Aspect-oriented programming is quantification and obliviousness, Aspect-Oriented Software Development, pages 21–35. Addison-Wesley, 2005.

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

[18] Gorlatch S., Dunnweber J. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In Future Generation Grids, Springer, 2006.

[19] Hanenberg S., Schmidmeier A., Idioms for Building Software Frameworks in AspectJ; 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, MA, March 17, 2003.

[20] Hannemann, J., Kiczales, G., Design Pattern Implementation in Java and AspectJ, OOPSLA 2002, November 2002.

[21] Harbulot, B., Gurd, J., Using AspectJ to Separate Concerns in Parallel Scientific Java Code, AOSD 2004, Lancaster, UK, March 2004.

[22] Laddad R., AspectJ in Action – Practical Aspect-Oriented Programming, Manning 2003.

[23] Lopes C. V., D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA, November 1997.

[24] MacDonald S., From Patterns to Frameworks to Parallel Programs, PhD thesis, Department of Computing Science, University of Alberta, 2002.

[25] Monteiro M. P., Fernandes J. M., Towards a Catalogue of Aspect-Oriented Refactorings. AOSD 2005, Chicago, USA, March 2005.

[26] Mortensen M., Ghosh S., Creating Pluggable and Reusable Non-functional Aspects in AspectC++, The 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS’06), Bonn, Germany, March 2006.

[27] Navarro L., Südholt M., Vanderperren W., Fraine B., Suvée D., Explicitly distributed AOP using AWED, AOSD'06, Bonn, Germany, March 2006.

[28] Nishizawa M., S. Shiba S., Tatsubori M., Remote pointcut - a language construct for distributed AOP, AOSD’04, 2004.

[29] Parnas D. L., On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), pp. 1053-1059, December 1972.

[30] Pawlak R., Seinturier L., Duchien L., Florin G., Legond-Aubry F., Martelli L., JAC: an aspect-based distributed dynamic framework, Software: Practice and Experience, vol. 34, no. 12, Oct. 2004.

[31] Philippsen M., Zenger M., JavaParty – transparent remote objects in Java. Concurrency: Practice and Experience, vol.19 n.11, November 1997.

[32] Rabhi F., Gorlatch S., (ed), Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2003.

[33] Riehle D., Brudermann R., Gross T., Mätzel K., Pattern Density and Role Modeling of an Object Transport Service. ACM Computing Surveys, 32(1es): 10, (March 2000).

[34] Soares S., Eduardo Laureano E., Borba P., Implementing distribution and persistence aspects with aspectJ. OOPSLA 02, Seattle, USA, November 2002.

[35] Singh A., Shaeffer J., Szafron D., Experience with parallel programming using code templates, Concurrency: Practice and Experience, vol.10, n.2, February 1998.

[36] Sobral J., Cunha C., Monteiro M., Aspect-Oriented Pluggable Support for Parallel Computing, VecPar'2006, Rio de Janeiro, Brasil, June 2006.

[37] Sobral J., Fernando J., ParC#: Parallel Computing in .Net, Parallel Computing Technologies 2005 (PaCT'05), Russia, September 2005, LNCS vol. 3606, Springer 2005.

[38] Sobral J., Proença A., A Run-time System for Dynamic Grain Packing, Euro-Par'99, Toulouse, France, September 1999, LNCS vol. 1685, Springer 1999.

[39] Saunders S., Rauchwerger L., ARMI: an adaptive, platform independent communication library, ACM PPoPP 03**,** San Diego, USA, 2003.

[40] Sullivan, K. J., Griswold, W. G., Song, Y., Cai, Y., Shonle M., Tewari, N., Rajan, H., Information Hiding Interfaces for Aspect-Oriented Design, ESEC/FSE 2005, Lisbon, Portugal, September 2005.

[41] Tilevich E., Urbanski S., Smaragdakis Y., Fleury M., Aspectizing Server-Side Distribution, IEEE ASE 2003, Montreal, Canada, October 2003.

[42] Zhang C., Jacobsen H., Resolving Feature Convolution in Middleware Systems, OOPSLA’04, Vancouver, Canada, October 2004.