A Domain-Specific Language for Parallel and Grid Computing

João L. Sobral

Departamento de Informática
Universidade do Minho
4710-057 Braga,
PORTUGAL
jls@di.uminho.pt

Miguel P. Monteiro

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica
PORTUGAL
mmonteiro@di.fct.unl.pt

Abstract

This paper overviews a Domain-Specific Language (DSL) for parallel and grid computing, layered on top of AspectJ. This DSL aims to bridge the gap between sequential code and parallel/grid applications, by avoiding invasive source code changes in scientific applications. Moreover, it aims to promote the localization of parallelization and gridification issues into well defined modules that can be (un)plugged (from)to existing scientific applications. This paper builds on previous work based on AspectJ and presents the main motivations for implementing a DSL in preference to a pure-AspectJ solution. The paper presents the DSL's design rationale, overviews current implementation and open research issues.

1. Introduction

The landscape of parallel computing has drastically changed in the last years due to wide availability of parallel processing capabilities in every desktop machine. Moreover, grid systems connect worldwide computer resources that offer an almost unlimited computing power. This demands an important shift in the way that software is designed and developed. To take advantage of recent and future hardware, applications must be re-designed to be structured along parallel activities that can leverage these intrinsically parallel platforms. In addition, many decades of research on parallelizing compilers did not produce an approach competitive to hand-coded parallelization.

Traditional programming environments for parallel and distributed systems (e.g., OpenMP [3] and MPI [2], which are currently supported by most compilers/systems vendors) do not promote a proper separation of core and parallelization concerns. For instance, is has been argued that sending and receiving messages among processes is the *goto* of parallel programming [10]. Parallelization concerns usually become tangled with domain-specific code, which makes it hard to develop and maintain applications tailored for parallel and distributed systems [11]. This complicates the transition from sequential to parallel applications, as it requires invasive and non-reversible changes. Applications become dependent of a particular parallelization strategy, becoming hard to change parallelization issues without impact on the overall program structure. As a consequence, when scientists adapt sequential applications for parallel execution, they usually lose control of the program structure. Since their code becomes polluted with parallelization issues, they tend to loose the focus on their core activity, which is to develop applications for their research.

In earlier work, we addressed the separation of concerns in parallel computing by relying on AspectJ. We proposed the separation of parallelization concerns into partition, concurrency and distribution [19][16]. We developed reusable aspect libraries for concurrency [6] and partitioning [18]. Within this path, we also identified some kinds of functionality that cannot be implemented by an AspectJ library, so we resorted to the generation of code for that kind of functionality [20].

Our initial approach was to develop an aspect oriented framework that would provide an explicit separation between domain-specific and parallelization issues. Parallelization and gridification [13] issues should be (un)pluggable: enabling scientific code to run on parallel/grid systems should not require invasive changes on source code and these applications should be able to run when these concerns are not included into the build.

Unfortunately, the results derived were unsatisfactory, for two reasons. First, developing the framework proved to be harder than expected, since AspectJ does not provide a suitable model for composing the various concerns involved. The asymmetric model of AspectJ does not properly address the composition of multiple reusable aspects when these aspects act on a shared set of join-points. In addition, the mechanisms provided by AspectJ to manage and configure the composition of aspects are inflexible. Second, we wanted the ensuing system to be used by people without a background in either aspect-oriented programming or parallel computing. Using a general-purpose aspect language is not a suitable approach to meet this requirement. To address the aforementioned issues, we resorted to a DSL that could integrate all features into a unified set of programming abstractions.

The rest of this paper is structured as follows. The next section discusses in more detail the motivations to develop a DSL for parallel and grid computing. Section 3 presents the design rationale under the developed DSL, the main abstractions supported and how they are composed to develop complex parallelization strategies. Section 4 outlines the current DSL implementation and Section 5 concludes the paper.

2. Motivation for a DSL for Parallel and Grid Computing

One of our main motivations to build a DSL was to "hide" details specific to AOP from the user of the framework. For instance, in [6] we introduced a reusable aspect intended to replace the intrusive *synchronized* Java keyword. To apply this abstraction to a concrete method, the programmer has to write something like the following code:

```
aspect aspect_name extends SynchroniseProtocol {
   pointcut synchronisedUsingCapturedLock():
      <pointcut definition>;
}
```

The programmer has to write an *aspect* that *extends* the *SynchroniseProtocol* and to implement the *pointcut* named *synchronisedUsingCapturedLock* providing the *poincut definition*.

Some mechanisms could be implemented by annotations (e.g., `@Synchronized`). However, this leads to the non-localization of parallelization issues. In addition, not all mechanisms can be implemented through annotations (e.g. see the Waiting guards in [6]). To compound the problem, not all mechanisms can be implemented by a reusable aspect in AspectJ. One such mechanism is the injection of distribution code (e.g., to make a class RMI enabled, allowing it to be remotely accessible). Although in this particular case we can modularize distribution code into a single aspect [15] or to automate the generation of code [5][21], it is not possible to build a reusable aspect in AspectJ for that purpose. On the other hand, it is it relatively easy to use a code template that can be applied to each specific case through simple text pattern replacement.

We also aim for an easier and more explicit aspect instantiation, including the ability to apply the same abstraction more than once to a given system, as well as an explicit control over aspect precedence. The AspectJ mechanism to specify aspect precedence has significant limitations, namely in cases when the aspects extend a common super-aspect. Moreover, declaring precedence in AspectJ requires the implementation of a new aspect just for that purpose. We want a simpler AOP language, with a compositional model that is easier to understand and use by people without a background in AOP. One of such models was suggested in [12].

Despite these limitations AspectJ has nice features that make it attractive to our specific application domain:

- aspects can modularize many parallelization and gridification issues and can be easily (un)plugged from domain-specific code;
- in many cases where it is not possible to develop a reusable aspect, it is generally easier to generate a case specific aspect (from a predefined code template) than to generate plain Java for the same purpose.

Thus, to take advantage of AspectJ's features and overcome its shortcomings, we developed a DSL on top of AspectJ.

3. Overview of the DSL

The first motivation of our DSL was to hide the details specific to AspectJ, as much as possible. Using the example of the replacement of the Java *synchronized* keyword, the only thing that the programmer should care about is the name of the mechanism to be used (in this case, *Synchronized*) and the pointcut definition (e.g., joinpoints where the mechanism applies). Following this rationale, we just need to supply the name of the mechanism and the pointcut specification. Among several possible alternatives, we selected a syntax akin to that of C++ templates. In this specific case, we instantiate the mechanism through:

```
Synchronized<pointcut definition>
```

In general, all mechanisms implemented as reusable aspects in AspectJ can be applied to concrete cases using this type of syntax. Abstract pointcuts, abstract methods and configuration parameters (e.g., integer values) are all specified by template parameters. We followed a similar strategy for mechanisms that cannot be implemented through a reusable aspect. In such cases, all parameters required to generate the code are provided through template parameters.

To illustrate the rationale of this DSL we present a simple example: the RayTracer benchmark from the Java Grande Forum [14] that was previously studied using a pure AspectJ solution [19]. This benchmark renders an image of sixty spheres. The domain-specific code creates an instance of a RayTracer and calls the method render, specifying an image interval to render:

```
RayTracer rt = new RayTracer();
Interval interval = new Interval(0,500);
Image result = rt.render(interval);
```

To develop a parallel version of this code, we could create several instances of *RayTracer* objects and call the method *render* on each instance, providing a different image interval to render and combining the partial results produced by each instance.

This kind of functionality can be provided through a reusable aspect. To replicate instances of class *RayTracer*, we can develop a reusable aspect that creates an aggregate of instances instead of a single instance by using a marker interface:

```
interface Aggregate {};

Aggregate around() : call (Aggregate+.new()) {
   for(i=0; i<numberOfWorkers; i++)
      farm[i] = proceed();
   return(farm[0]);
}
```

Next we can build an aspect that applies this *replicate* functionality to the *RayTracer* class, using the AspectJ construct *declare parents : RayTracer implements Aggregate*.

A reusable aspect to call the method *render* on each *RayTracer* instance requires access to the aggregate (created by the replicate aspect) and the specification of two abstract methods. The first (*divideFunction*) specifies how the interval is divided among instances of class *RayTracer* and the second (*combineFunction*) specifies how to combine the results computed by each instance:

```
abstract Vector<P> divideFunction(P);
abstract R combineFunction(Vector<R>);
abstract pointcut poincutName(/*…*/);

R around(/*…*/) : poincutName(/*…*/) {
   Vector<P> parms = divideFunction(/* args */);
   Vector<R> results = new Vector<R>();
   for(i=0; i<numberOfWorkers; i++)
      results.add( proceed(farm[i], parms[i]) );
   return(combineFunction(results));
}
```

divideFunction and *combineFunction* have the purpose, respectively, to divide a data set into disjoint parts and to combine results obtained by independently processing each part. This is a type of parallelization common in many parallel applications, known as MapReduce [7] or farm [8]. AOP implementations have the advantage of promoting an explicit separation between domain-specific and parallelization issues. In addition, parallelization issues become (un)pluggable: they do not require invasive changes on source code and applications can still run when these concerns are not included into the build.

The DSL includes two abstractions to provide the above functionality. The first, *Replicate<Class T>* creates several replicas of class T (the number of replicas may be provided by a parameter or selected by the run-time system). The second, *DivideCombine<Class T, pointcut P, divideFunction, combineFunction>* implements the MapReduce functionality.

Using these abstractions we can write the parallelization of the RayTracer as follows:

```
Replicate<RayTracer>

Vector<Interval> split(Interval in) {
        … // split in into sub-intervals
}
Image join(Vector<Image> in) {
        … // join rendered sub-images
}
DivideCombine<RayTracer, render, split, join>
```

The previous code can parallelize the ray tracer to take advantage of computing systems based on shared memory. Additional abstractions were designed to develop applications that can take advantage of clusters and grids of clusters. The *Separate<Class T>* enables an instance of a class to be remote (e.g., by using Java RMI) and makes it possible to remotely create and to receive remote method invocations on machines on a local network (e.g., cluster). *GridSeparate* has a similar purpose but for grid systems, where computing systems are connected through wide area networks (it relies on a grid middleware to remotely deploy instances of a class). A rich set of abstractions address concurrent execution [6], application partitioning [18], including abstractions to cope with data dependences among tasks, and grid specific issues [17], such as load distribution and fault-tolerance.

The ability to compose the set of abstractions provided by the DSL is particularly useful in parallel and grid computing. One way to improve application scalability is to decentralize application control in a hierarchical manner. Grid systems are usually made of clusters of clusters of multi-core machines. A hierarchy of control closely matches the physical hierarchy of the systems, being able to better taylor the system's structure to the local bandwidth (e.g., by using communication based on shared memory when object instances are placed on the same machine). This hierarchy of control can be achieved by applying the *Replicate* abstraction *more than once*. For instance, instead of having a single aggregate of ray tracers (e.g., with 256 instances) we could have an aggregate of smaller aggregates (e.g., an aggregate of 16 ray tracers, each one with 16 ray tracers).

The main composition problem of using reusable aspects to implement the DSL is that reusable aspects were not designed to expose joinpoints for other aspects (e.g., *proceed* is not supported in AspectJ's joinpoint model). We could try to overcome this shortcoming by using some intricate solution (e.g., by encapsulating every *proceed* into a method), but at this stage we decided to simply avoid the use of *proceed*. In our specific case, our *replicate* aspect template would be something like:

```
T around() : call (T.new()) {
   for(i=0; i<numberOfWorkers; i++)
      farm[i] = new T();
   return(farm[0]);
}
```

Replicate<RayTracer> abstraction replaces T by *RayTracer*. This solution solves our main composition issue, in that every abstraction exposes a set of joinpoints (*new RayTracer()* in this case). This allows us to write *Replicate<Replicate<RayTracer>>* to create a two-level hierarchy of *RayTracer*.

A more subtle composition issue is that each abstraction acts *incrementally*: it can only intercept joinpoints in the domain-specific code or introduced by the previous aspect in the chain (but not both). This way, the DSL presents a simpler rationale than AspectJ and includes support for incremental development.

This incremental model of composition leads us to view our approach as more "transformation oriented". We start with a sequential program and apply a set of programmer-specified transformations (i.e., high-level parallel programming abstractions) that generate the parallel version of the same code. Actually, we derived these programming abstractions by looking at sequential and parallel versions of the same program, and devised a set of abstractions supporting the automation of the code transformations. This yields a parallelization approach that is non-invasive, modular and (un)pluggable. For more complex transformations, we introduced a constructor that allows us to *store* the result of a transformation into a variable.

This rationale allows us to write some nice combinations of the *Separate* and *Replicate* abstractions. For instance, the following example creates an aggregate of ray tracers that are distributed along the nodes of a local cluster (through the *Separate* abstraction) and that are replicated again on each cluster node (for instance to take advantage of multi-core nodes).

```
Replicate<Separate<Replicate<RayTracer>>>
```

This composition model also allows us to *extend* abstractions by composition. For instance, to address machines with different speeds, we implemented a scheduler service [17] that selects the fastest instance of an aggregate for each method call and a failover service that addresses faulty nodes in a grid environment. Both services can be combined with the *Replicate*/*DivideCombine* template to adapt and configure applications for grid environments.

4. Overview of the DSL's Implementation

Our current prototype [4] uses the ANTLR tool to recognize programs written into the DSL syntax and generates code by processing predefined code templates.

Each programming abstraction in the DSL is implemented by one code template. A simple string replacement tool generates application specific aspects from code templates. We generate one aspect for each template instantiation in a DSL program and no changes are performed to the original sequential program (although, changes are performed by the AspectJ weaver to compose the base program with the generated aspects). For instance, the replicate template is as follows:

```
<T> around() : call (<T>.new()) <TARGET> {
   for(i=0; i<numberOfWorkers; i++)
      farm[i] = new <T>();
   return(farm[0]);
}
```

The <T> tag is replaced by the target class and the <TARGET> tag is replaced by the target aspect where the abstraction applies (e.g., the previous aspect in the chain).

To illustrate the composition of multiple aspects we show the code generated to implement *Replicate<Replicate<RayTracer>>*. In this case two aspects are generated:

```
aspect replicate1 {
   RayTracer around() : call (RayTracer.new())
      && within(/*MAIN*/) {
         for(i=0; i<numberOfWorkers; i++)
            farm[i] = new RayTracer();
         return(farm[0]);
   }
}
```

```
aspect replicate2 {
   RayTracer around() : call (RayTracer.new())
      && within(replicate1) {
         for(i=0; i<numberOfWorkers; i++)
            farm[i] = new RayTracer();
         return(farm[0]);
   }
}
```

In the case of templates that have joinpoints as parameters, we follow a similar strategy, although we use additional tags to refer to different kinds of template parameters. Currently we only support templates where the poincut refers to one method call. We use several tags to identify each element of that method (such as <METHOD>, <RT>, <ARGS> and <ARGS_L>). Our AspectJ code template for the *DivideCombine* is as follows:

```
DivideCombine<T, POINTCUT, MDIVIDE, MCOMBINE>

<RT> around(<ARGS>) :
      call (<RT> <T>.<METHOD>(..)) /* ... */ {
   Vector parms = <<MDIVIDE>>(<<ARGS_L>>);
   Vector<<RT>> results = new Vector<<RT>>();
   for(i=0; i<numberOfWorkers; i++)
     results.add(farm[i].<METHOD>(parms[i]));
   return(<MCOMBINE>(results));
}
```

The DSL can be extended with new templates, either by combining existing templates or by building new code templates. This is possible since each DSL abstraction is implemented by one code template and the template tool can manage new templates.

Although the current solution enables us to overcome the main limitations detected in AspectJ, especially by improving its composition ability, this solution has some limitations. First, we cannot support polymorphic pointcuts. Indeed, the DSL presently supports pointcuts relating to calls to a single method. Second, this solution suffers a *code bloat* problem similar to that of C++ templates. Performance of the current DSL implementation is comparable to the use of case specific aspects (e.g., hand written, already presented in [16][17]).

5. Conclusion

This paper presents a DSL for parallel and grid computing layered on top of AspectJ. The DSL aims to localize parallelization and gridification issues into modules that can be (un)plugged from scientific programs.

This DSL results from an effort to modularize parallelization concerns by means of AOP. The original motivation for this DSL was to hide AOP technology and to provide a more flexible and explicit way to compose abstractions developed as reusable aspects.

AspectJ enables us to quickly develop an initial DSL prototype to assess the feasibility of this kind of approach. The current DSL implementation is being used to parallelize and gridify several scientific applications [1], without polluting them with parallelization and gridification issues.

The DSL incorporates abstractions whose implementations tested the capabilities of AspectJ to the limit. The addition of new kinds of abstractions requires other kinds of tools. One is the support for distributed data structures (e.g., the partitioning of an array and the subsequent distribution among a set of machines, and the management of data dependences). AspectJ lacks a suitable joinpoint for this, as transforming these data representations involves fine-grained joinpoints and overhead is very sensitive to implementation strategy. Another issue is the support for a clearer separation of concerns in parallel computing, based on multiple concern-specific languages, one for each kind of concern.

Acknowledgments

This work was supported by AspectGrid project (Pluggable Grid Aspects for Scientific Applications, GRID/GRI/81880/2006), SeARCH (CONC-REEQ/443/EEI/2005) and SOFTAS (POSI/EIA/60189/2004) all funded by Portuguese Fundação para a Ciência e Tecnologia and European funds (FEDER).

References

[1] AspectGrid project web page, http://gec.di.uminho.pt/aspectgrid

[2] Message Passing Interface Forum. http://www.mpi-forum.org/

[3] OpenMP architecture review board, OpenMP Application Program Interface, Version 2.5, May 2005, http://www.openmp.org

[4] Brito, E., Tool for combining re-usable aspects – PSL, Technical Report. Universidade do Minho, March 2008.

[5] Ceccato, M., P. Tonella, P., *Adding Distribution to Existing Applications by means of Aspect Oriented Programming*, IEEE SCAM'04, September 2004.

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

[7] Dean, J., Ghemawat, S., *MapReduce: simplified data processing on large clusters*, Communications of the ACM, 51(1), January 2008.

[8] Fernando, J., Sobral, J., Proenca. A., *JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing*, IEEE CCGrid'06, Singapore, May 2006.

[9] Foster, I., Kesselman, C., *The GRID2 Blueprint for a New Computing Infrastructure*, Morgan Kauffman, 2004.

[10] Gorlatch, S. *Send-Receive Considered Harmful: Myths and Realities of Message Passing*, ACM TOPLAS, 26(1), January 2004.

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

[12] Lopez-Herrejon, R., Batory, D., Lengauer, C, *A disciplined approach to aspect composition*, Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 06), USA, January, 2006.

[13] Mateos, C., Zunino, A, Campo, M., *A survey of Approaches to Gridification,* Software: Practice and Experience, to be published.

[14] Smith, A., Bull, J., Obdrzálek, J., *A Parallel Java Grande Benchmark Suite*, Supercomputing 2001, Denver, USA, November 2001.

[15] Soares, S., Loureiro, L., Borba, P., *Implementing Distribution and Persistence Aspects With AspectJ*, OOPSLA '02, Seatle, USA, November 2002.

[16] Sobral, J., *Incrementally Developing Parallel Applications with AspectJ*, IEEE IPDPS'06, Rhodes, Greece, April 2006.

[17] Sobral, J., *Pluggable Grid Services*, 8th IEEE/ACM International Conference on Grid Computing (Grid 2007), Austin, Texas, September 2007.

[18] Sobral, J., Cunha, C., Monteiro, M., *Aspect-Oriented Pluggable Support for Parallel Computing*, VecPar'06, Rio de Janeiro, Brasil, June 2006.

[19] Sobral, J., Cunha, C., Monteiro, M., *Aspect-Oriented Support for Modular Parallel Computing*, 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'06), University of Virginia, TR CS 2006-1, Bonn, Germany, March 2006

[20] Sobral, J., Proença, A., *Enabling JaSkel Skeletons for Clusters and Computational Grids*, IEEE Cluster (Cluster 2007), Austin, Texas, September 2007.

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