Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

March 21, 2006

Held in conjunction with the Fifth International Conference on
Aspect-Oriented Software Development (AOSD 2006)

Bonn, Germany

Yvonne Coady, David H. Lorenz, Olaf Spinczyk, and Eric Wohlstadter, editors

Department of Computer Science
University of Virginia
Charlottesville, VA 22904
Technical Report CS-2006-01

Table of Contents

Creating Pluggable and Reusable Non-functional Aspects in AspectC++

Michael Mortensen
Hewlett-Packard
3404 E. Harmony Road, MS 88
Fort Collins, CO 80528
and
Computer Science Department
Colorado State University
Fort Collins, CO 80523
mmo@fc.hp.com

Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins, CO 80523
ghosh@cs.colostate.edu

ABSTRACT

An object-oriented framework enables both black box reuse and white box reuse in client applications, serving as an important infrastructural building block. We are refactoring framework-based applications to modularize cross-cutting concerns with aspects. In this paper, we explore implementation issues we encountered while creating non-functional aspects in AspectC++ that are pluggable and reusable.

1. INTRODUCTION

Aspect-oriented programming can be used to modularize cross-cutting concerns in application software. Non-functional aspects modularize cross-cutting concerns that do not directly affect functionality, but instead improve characteristics such as performance, dependability, and configurability. Though important, they are orthogonal to the primary functionality. Modularizing non-functional cross-cutting concerns with aspects reduces duplicated code. In addition to reducing duplicated code, aspects improve pluggability, since a single pointcut can enable or disable the advice code [6]. Designing aspects with virtual pointcuts and virtual methods allows an aspect to be reused in different applications, with application-specific behavior or pointcuts specified in a concrete aspect.

Object-oriented frameworks serve as infrastructure building blocks for the development of large-scale industrial applications. A framework's application programming interface (API) enables blackbox reuse [14] while framework classes enable white box reuse through object-oriented mechanisms such as composition, inheritance, and polymorphism [13].

Domain-specific frameworks are typically designed as large, layered systems that include a kernel layer of foundational services such as access to the operating system, container classes, and external data stores and domain-specific layers [2]. VLSI chip design involves a complex, multi-step flow that requires many individual tools to share data. VLSI CAD frameworks provide common translators for different file formats and enable applications to focus on a specific task while leveraging framework components for data translation, persistence, and memory models [1].

Frameworks provide extension points for client applications that enable object-oriented reuse. However, such extension points must be anticipated by framework developers during the design of the framework. Aspects provide an additional way of providing extension points for client applications. Advice can be layered on top of existing code whether or not the developer anticipated the extension.

Although frameworks can be designed or refactored to use aspects [5], changing existing frameworks can be problematic due to license restrictions, the large number of applications already using the existing framework implementation, or the effort required to modify a large framework. Rather than modifying these frameworks, we can identify cross-cutting concerns common to many framework-based applications, particularly cross-cutting concerns that relate to framework use and extension. In like manner, Ghosh et al. [4] differentiate between business logic and code that interacts with middleware services, and propose that middleware-related code be modularized as an aspect.

We use aspects to enhance the use of existing, unmodified OO frameworks. We are creating a library of framework-based aspects that represent cross-cutting concerns in framework-based applications. We refactor applications (not the framework) and weave in aspects from the aspect library; the refactored applications use both the aspect-library and the framework. For our aspect library to be beneficial to many framework-based applications, aspects must be designed with pluggability and reuse in mind. While developing an aspect library for an VLSI CAD framework used at Hewlett-Packard, we found that tradeoffs between alternative implementations have implications for reusing aspects. We also found that aspects themselves can be developed in a modular way so that a caching aspect can monitor how much benefit (in terms of cache hit rates) it provides to the system.

The remainder of the paper is structured as follows. Section 2 describes the design and implementation of a caching aspect for framework-based applications. In section 3 an aspect for enabling user-configuration of a large CAD application is considered. Section 4 discusses related work, followed by conclusions and future work in Section 5.

2. CACHING

2.1 Motivation

Caching is used to improve performance while maintaining the same functional behavior. Caching is a common example of a cross-cutting concern that can be modularized as an aspect. The implementation is largely the same across all functions that are cached [7, 8].

Framework-based applications contain functions or methods that implement caching in an ad-hoc way. In VLSI CAD applications, these functions calculate some property of a circuit element (an electrical net or transistor). Calculating the property can involve complex, time-consuming steps, such as reduction of RC networks for resistance calculation or graph traversals to calculate reachability. In a framework-based application, the objects being cached are often instances of framework classes, while the function doing the calculation is application-specific. One of the framework-based applications that we are refactoring is an electrical rules checking (ERC) tool developed at Hewlett-Packard, the `ErcChecker`. The `ErcChecker` consists of approximately 80,000 lines of C++ code.

2.2 Aspect Identification

We can identify cached functions idiomatically since they use a manually inserted static set or static map. The `ErcChecker` contains 38 functions that separately implement similar caching code. These functions were in various classes and did not have a common naming convention, so a list of pointcut expressions grouped by the logical `or` operation can be used to capture them, and the original tangled code can simply be deleted.

2.3 Design challenges

Caching consists of storing the result value for some input. In C++, the input can be one of several things, since we can have procedural functions where the input is a value, method calls where the input is the object invoking the method, or functions or methods where an object is passed as a parameter.

Non-functional aspect pluggability is important since these aspects may not always be needed. For example, user comments in one manually cached `ErcChecker` function indicate that the cache was not improving performance. Caching is only effective if cached values are used: a function that does not recompute the same value during program execution will not benefit from caching. In prototyping and developing caching as an aspect that could easily be enabled and disabled and also monitor its benefits, we found a variety of interesting trade offs and challenges.

2.4 Cache implementations

2.4.1 Two simple caches

We implemented two simple caching aspects: one for caching functions and static methods, and one for caching object-based method calls. These caches are similar to a generic caching strategy described by Lohmann, Blaschke, and Spinczyk [8], although they focus on using C++ template meta-programming with AspectC++, and demonstrate capabilities such as handling an arbitrary number of function or method parameters.

Example code for caching a function or static method based on its first argument is shown in Appendix A, as the `AbstractFunctionCache`. The second cache is structured like the `AbstractFunctionCache`, but the cache key is the object on which the method is called rather than the first parameter. This changes the static map definition:

```
static std::map <
      typename JoinPoint::Target*,
      typename JoinPoint::Result >
   theResults;
```

Retrieving the target is done dynamically with `tjp->target()`.

Both aspects declare a virtual pointcut so that use only requires creating a concrete aspect that defines the pointcut, as shown below. Because the cache creates a static map inside the around advice, a concrete aspect like `MethodObj-Caching` (see below) whose pointcut matches multiple join points will have separate caches for each one.

```
aspect MethodObjCaching :
   public AbstractMethodCache {
      pointcut ExecAroundMethod() =
            execution("% Circle::getArea(...)")
         || execution("% Donut::getArea(...)");
};
aspect SquareCache :
   public AbstractFunctionCache {
      pointcut ExecAroundArgToResult() =
            execution("% SquareArea(...)");
};
```

2.4.2 TrackingCache

In order to measure the effectiveness of the caching concern, we add hit and miss counts directly to the implementations above. However, we immediately recognize two constraints: we must initialize the Statistics before the first access, and after all calls to the caching functions are done the Statistics need to be displayed.

Because a concrete aspect (such as `MethodObjCaching`) may match multiple join points, hits and misses should also be tracked for each join point. We can create a simple C++ struct for representing hits and misses whose constructor does initialization, and can store the struct for each join point with `JoinPoint::signature()`. We print out the map `after` the `main` function has executed to get the final results. We show the statistical tracking implementation in Appendix B. This caching aspect defines the `Stats` struct and overrides the output operator (`<<`) so that `Stats` variables are easily printed.

The around advice (`ExecAroundMethod`) has two lines of cache Statistics code (lines 35 and 40) mixed in with the caching code, with the Statistics functionality is spread across two advice bodies. Any changes tracking cache results will require manually editing these lines, which look like the kind of tangled code that aspects strive to reduce or eliminate.

2.4.3 Caching with a utility class

Since cache hit and miss statistics are linked to cache behavior, next we modify the aspect to use a utility class, which we can extend via inheritance to add tracking information. This provides an aspect implementation that enables easily plugging or unplugging the cache tracking (hit/miss) functionality.

First, we define the helper class, which is a templatize cache class that can handle arbitrary key and value types. Source code is shown in Appendix C. Next, we modify the aspects (parameter based and object-based) to use `AspectCache`. The caching aspect that uses the class can simply instantiate it and call `get` and `set` rather than directly manipulating the cache data. A caching aspect for storing parameter/return value pairs is shown in Appendix D.

Note that the `AspectCache` class cannot be accessed in the `after` advice associated with `main`. Because of C++ scoping rules, the `AspectCache` variable (`theResults`) is only accessible in the `around` advice for the cached function. It is needed there for two reasons. First, we need it to be a static object inside the around advice so that it is created once for each join point, allowing this aspect to have a pointcut that matches many join points and yet have each join point be separately cached. Second, the cache is instantiated with static type information available *only* inside the join point (`JoinPoint::Arg<0>::Type` and `JoinPoint::Result`).

We rely on the destructor for the static class when extending the `AspectCache` via inheritance to create the `CountingAspectCache`, which we show in Appendix E.

The `CountingAspectCache` can be directly substituted in the aspect, or we can have an aspect associated with each class (`AspectCache` and `CountingAspectCache`) and then create a concrete aspect based on whether or not we want tracking. Since the `CountingAspectCache` may be associated with one of many join points, we pass in the join point signature to the constructor so that it will be accessible (at the end of execution) in the destructor (lines 11-17), which prints out join point names and cache statistics.

The `CountingAspectCache` inherits concrete methods from `AspectCache`. While reuses code, it could lead to problems if the `AspectCache` is modified without regard to how that might impact the `CountingAspectCache`. In fact, the relationship between the `CountingAspectCache` and `AspectCache` is analogous to the relationship between `CountingBag` and `Bag` in the example code for the fragile base class problem [11]. An alternative that avoids this would be an abstract caching class that only defines pure virtual methods (which forms an interface class in C++), from which both `AspectCache` and `CountingAspectCache` could be derived and which could be implemented separately.

Since we are providing caching as part of an aspect library for framework-based applications, the best approach seems to be to let users directly access both cache aspects. During development and testing, cache Statistics could be tracked to determine the value of caching a particular function. Before production release, the application developer would switch to the regular (non-counting) cache to reduce cache overhead. For aspect library stability, implementing the cache classes based on a C++ interface base class would be preferred.

2.4.4 Object Identity

Many of the functions cached are based on calculating a value for a parameter that is an object (e.g., pointer to an electrical net, pointer to a transistor). Object identity and aliasing introduce additional complexity to caching. Using pointers is efficient with a map. However, multiple references to the same object result in each one being added to the cache rather than reusing the calculated value, as shown by lines 32, 38 and 39 of example in Appendix F.

Hewlett-Packard's framework, like other VLSI CAD frameworks, reads in a persistent store such as a netlist and builds an in-memory model in which pointers are usually unique and are associated with circuit components. However, in general we need to be careful for cases where we can have different memory references to the same object.

There may also be distinct objects that are equivalent in some respect. In an electrical design, there may be many instances of the same circuit component. In the example in Appendix F, square structs `s1` and `s3` (on lines 24 and 28) have the same coordinates. Depending on the cost to compare objects versus the cost of computing some function, we may want to cache properties of objects and avoid recomputation when we have seen an alias to an object or an equivalent object.

2.5 Impacts on the ErcChecker

Originally, developers manually implemented caching for some functions of a large application to improve performance. The implementation did not use aspects, and resulted in similar code being scattered in many functions. Implementing caching as an aspect improves modularity and results in the removal of duplicated cache code from 27 locations. By using a helper class and inheritance, application developers can easily switch between a fast cache and a slightly slower one that measures the benefit of the cache. We did not find a noticeable performance penalty when running the refactored `ErcChecker`. Benchmark testing of the aspects[1] found that the aspect implementation was only 15 seconds slower than the direct implementation (even with gathering hit and miss data) for a loop of 9 million method calls.

Aspects, like application code, must be maintained. Using inheritance and abstract interface classes when implementing aspects improves cache pluggability, allows the aspect to be reused in many contexts, and that limits the change impact if the aspect or utility classes change. When designing caching as an aspect, subtle language features such as aliasing and domain knowledge about object equivalence are important to consider since they affect cache and application performance.

3. RUN-TIME USER CONFIGURATION

3.1 Motivation

A second type of extra–functional aspect we consider is run-time user configuration of the `ErcChecker`. ERC systems automate a number of electrical circuit checks. Example checks include checking for proper transistor ratios between the pull-up and pull-down transistors of an inverter, checking for fan-out limits, or checking for drive strength problems [12]. An ERC checking tool typically performs many types of checks on a circuit, reporting violations to the user.

The `ErcChecker` implements 58 different electrical checks as a class hierarchy. Because of the number of queries and

[1]Performance data are from a 2.4GHz Linux Xeon desktop system with 3 GB RAM.

the run-time of the tool (often several hours for a circuit), the `ErcChecker` allows users to turn any of the checks off through a user configuration file. Before time-intensive circuit traversal or electrical query evaluation, each of the 58 types of electrical calls a configuration method to see if the user has disabled that electrical check. Although this does affect the functionality of the final run, this functionality is orthogonal to the primary functionality: the actual electrical rules checking.

3.2 Aspect Identification

In the `ErcChecker`, each class has a static method, `createQueries()`, that calls `ErcQuery::getQueryConfig` before creating and evaluating query objects to see if the query is disabled:

```
void InformAlwaysOnPassGate::createQueries
(bcmInstance *fet)
{
   // See if this query is turned off
   if (ErcFet::getQueryConfig
      ("InformAlwaysOnPassGate")== eOff)
      return;
   ...
```

Because the `ErcFet::getQueryConfig` call to check for run-time user configuration crosscuts all electrical queries, and its behavior is always the same (disable running the check if a user configuration command is found), it is a good candidate for an aspect. A query that fails to call `ErcFet::getQueryConfig` in this manner will function, but cannot be disabled by configuration commands, which is a system-wide policy for the `ErcChecker`.

The original C++ code embedded the literal string for the electrical check's name (such as "InformAlwaysOnPassGate"). This was done because C++ lacks reflection, which otherwise could query the name of the class and method name. AspectC++ provides this level of reflection through `thisJoinPoint`. In this application, the string literals match the class name for the method. Without such regularity, it would be difficult to refactor to an aspect and automatically calculate the electrical check context.

3.3 Design challenges

The aspect must first extract the electrical query name. We could simplify the aspect code by modifying `ErcFet::getQueryConfig()` to accept the join point signature and parse the string. Doing so would change the functionality of an existing behavior to depend on the aspect, which would limit aspect pluggability. In addition, `ErcFet::getQueryConfig()` is an overloaded function that can take additional arguments (such as an electrical net or transistor) to provide more fine-tuned user configuration. The fine-grained calls to `ErcFet::getQueryConfig()` with extra parameters are interspersed throughout the electrical checks and do not have a regular structure.

3.4 Aspect Implementation

The aspect for this concern uses the join point signature to get the calling context (`void InformAlwaysOnPassGate::createQueries (bcmInstance *fet)` and calls the same method as the original code (`ErcFet::getQueryConfig()`).

```
aspect QueryConfigAspect {
   pointcut createQuery() =
      execution("% %%::createQueries(...)");
   advice createQuery() : around() {
     std::string jpName = JoinPoint::signature();
     int first_space = jpName.find(' ');
     int scope_operator = jpName.find("::");
     std::string className=jpName.substr(
          first_space+1,
          scope_operator-first_space-1);
     if(ErcFet::getQueryConfig(className)==eOff)
       return;  //user config exists, SKIP
     tjp->proceed();
   }
};
```

The aspect above handles the call to `ErcFet::getQueryConfig()` that occurs at the beginning of the static method `createQueries` for each `ErcQuery` class, but does not handle the additional fine-grained user configuration calls mentioned in the design challenges section. The fine-grained calls do not occur in a regular way that can be described by an AspectC++ pointcut. Instead, they are associated with loops and conditional statements in the body of the `createQueries` method for each class.

3.5 Impacts on the ERC application

Using an aspect for the call to `ErcFet::getQueryConfig()` inside the `createQueries()` method for each electrical query class improves the modularity of the code. Since the scattered code is only a single method call, there is not significant code reduction, but using an aspect provides an automated way to enforce the policy that each class check the code. A query class that does not call `ErcFet::getQueryConfig()` will not provide the desired user configuration.

The policy itself can easily be enabled, modified, or disabled for the `ErcChecker`. Other VLSI CAD applications developed at Hewlett-Packard provide similar configuration options, although the implementation varies by application. We are investigating how easily this aspect, developed for the `ErcChecker`, can be reused by other applications.

4. RELATED WORK

Lohmann, Gal, and Spinczyk [9] demonstrate that C++ template metaprogramming can be used to develop code with an aspect-oriented style, but without the obliviousness of aspects: everything must be explicitly instantiated through templates, which need to have the extension points designed in. Lohmann, Blaschke, and Spinczyk [8] have shown that aspects in AspectC++ can use templates and template metaprogramming. Their work on caching influenced our work, but their focus is on using template metaprogramming to create a single aspect that can cache a function with an arbitrary number of parameters, while our focus is on the implementation trade-offs that occur in making any cache pluggable and reusable.

Lohmann et al. [10] use AspectC++ to modularize extra-functional (also called non-functional) aspects. They focus on emergent properties that result from architectural properties, while we focused on explicit non-functional aspects. They also observe that while non-functional aspects are feasible, the implementation can contain challenges and subtle details that require in-depth analysis of their viability.

Duclos, Estublier, and Morat [3] consider mixing component-based design with aspect-oriented programming, which is similar to our goal of using aspects with existing frameworks. Their approach, rather than extending frameworks with a single aspect language, proposes providing two languages so that component development and component use with aspects are done separately.

5. CONCLUSIONS AND FUTURE WORK

Aspects are effective for modularizing non-functional crosscutting concerns. We explored two types of non-functional concerns: caching and user run-time configuration. Although aspects were well suited for both, we identified specific issues that should be considered when using aspects:

- Non-functional aspects should be designed (where appropriate) to monitor their own usefulness.
- Non-functional aspects should make use of stand-alone classes, aspect inheritance, interfaces, and other features that promote ease of maintenance of aspects.
- Subtle implementation issues such as object identity and object equivalence can complicate seemingly simple aspects.
- When refactoring to use an aspect, obtaining the context (such as the query name in the user configuration aspect) may be different in the aspect code than the equivalent functionality in the (original) base code. For example, the original base code might not have a regular naming structure that facilitates using pointcuts.
- A non-functional aspect may only be able to be *partially* factored to an aspect, as our `QueryConfigAspect` demonstrated.

These issues should not discourage the use of aspects, but can provide guidance and insight as aspect-based refactoring and design is performed.

The aspects improve pluggability of the non-functional features. The caching aspect is generic enough to be reused in multiple applications. The run-time configuration aspect modularizes a feature that is used in other VLSI CAD applications, although its current implementation may be too closely tied to how run-time configuration is done in the `ErcChecker`.

We continue to identify aspects for refactoring the ErcChecker. This work may yield additional insights into design considerations, and may identify more aspects. We are also interested in seeing if the fine-grained calls to `ErcFet::getQueryConfig()` described in Section 3.3 can be refactored to one or more additional aspects.

6. ACKNOWLEDGMENTS

The authors are grateful for the use of AspectC++ (www.aspectc.org) and for valuable feedback from Andreas Gal and Olaf Spinczyck through the AspectC++ mailing list.

7. REFERENCES

[1] Si2: Silicon integration initiative. http://www.si2.org/?page=72.

[2] D. Baeumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zuellighoven. Domain-driven framework layering in large systems. *ACM Comput. Surv.*, 32(1es):5, 2000.

[3] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 65–75. ACM Press, Apr. 2002.

[4] S. Ghosh, R. B. France, D. M. Simmonds, A. Bare, B. Kamalakar, R. P. Shankar, G. Tandon, P. Vile, and S. Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.

[5] S. Hanenberg, R. Hirschfeld, R. Unland, and K. Kawamura. Applying aspect-oriented composition to framework development - a case study. In *International Workshop on Foundations of Unanticipated Software Evolution, FUSE 2004*, Barcelona, Spain, 2004 (to be published).

[6] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, Nov. 4–8 2002. ACM Press.

[7] R. Laddad. *AspectJ in Action.* Manning Publications Co., Grennwich, Conn., 2003.

[8] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04).* ACM, 2004.

[9] D. Lohmann, A. Gal, and O. Spinczyk. Aspect-Oriented Programming with C++ and AspectC++ (Tutorial). In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004).* ACM Press, Mar. 2004.

[10] D. Lohmann, O. Spinczyk, and W. Schrder-Preikschat. On the configuration of non-functional properties in operating system product lines,. In D. H. Lorenz and Y. Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, Mar. 2005.

[11] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, July 1998.

[12] S. M. Rubin. *Computer Aids for VLSI Design.* Addison-Wesley VLSI Systems Series. Addison-Wesley, 1987.

[13] D. Schmidt and M. Fayad. Object-oriented application frameworks. *Communications of the ACM*, 10(40):32–38, 1997.

[14] J. van Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software Practice & Experience*, 10(31):277–300, October 2001.

APPENDIX

A. ABSTRACT FUNCTION CACHE

```
%#include <map>
aspect AbstractFunctionCache {
   pointcut virtual ExecAroundArgToResult() = 0;
   advice ExecAroundArgToResult() : around()
   {
      JoinPoint::Result *result_ptr = NULL;
      static std::map <
        typename JoinPoint::Arg<0>::Type,
        typename JoinPoint::Result > theResults;
      JoinPoint::Arg<0>::Type *arg_ptr =
          (JoinPoint::Arg<0>::Type*) tjp->arg(0);
      JoinPoint::Arg<0>::Type arg = *arg_ptr;
      if( theResults.count( arg ) ) {
         //already have the answer, return it
                result_ptr = tjp->result();
         *result_ptr = theResults [ arg ];
      } else {
         //proceed and store the answer
         tjp->proceed();
             result_ptr = tjp->result();
         theResults [ arg ] = *result_ptr;
      }
   }
};
```

B. TRACKING CACHE

```
struct Stats {
   int hits, misses;
   Stats() : hits(0),misses(0) {}
};
std::ostream& operator <<(std::ostream& os,
                          const Stats& stats) {
   os << "(Stats hits: " << stats.hits
      << " misses: " << stats.misses << ")";
   return os;
}
aspect AbstractTrackingMethodCache {
  pointcut virtual ExecAroundMethod() = 0;
  std::map<std::string, Stats> theStats;
  advice execution("% main(...)") : after()
  {
     std::map<std::string,Stats>::iterator iter;
     for(iter = theStats.begin();
       iter != theStats.end(); ++iter) {
       std::string jpName = (*iter).first;
       Stats  jpStats = (*iter).second;
       std::cerr << " JP: " << jpName
                 << " "<< jpStats << std::endl;
     }
  }
  advice ExecAroundMethod() : around()
  {
     JoinPoint::Result *result_ptr = NULL;
     static std::map<typename JoinPoint::Target*,
       typename JoinPoint::Result > theResults;
     JoinPoint::Target *target = tjp->target();
     if( theResults.count( target ) ) {
       //cache already had answer, return it
       result_ptr = tjp->result();
       *result_ptr = theResults [ target ];
       theStats[JoinPoint::signature()].hits++;
     } else { //not in cache, proceed and store
       tjp->proceed();
       result_ptr = tjp->result();
       theResults [ target ] = *result_ptr;
       theStats[JoinPoint::signature()].misses++;
     }
  }
};
```

C. CACHE CLASS FOR CACHING ASPECT

```
template < typename Key, typename Value>
class AspectCache {
protected:
   std::map< Key, Value> theData;
public:
   AspectCache() {}
   virtual ~AspectCache() {} //important for OO
   virtual bool has(Key& k)
   {  return (theData.count(k)>0); }
   virtual void add(Key& k, Value& val)
   {  theData[ k ] = val; }
   virtual Value& get(Key& k) {
      assert( has(k) ); //ensure we have it
      return theData[k];
   }
};
```

D. CACHING ASPECT THAT USES CACHE CLASS – PARAMETER VERSION

```
aspect AbstractFunctionCache {
   pointcut virtual ExecAroundArgToResult() = 0;
   advice ExecAroundArgToResult() : around()
   {
      JoinPoint::Result *result_ptr = NULL;
      static AC::AspectCache <
         typename JoinPoint::Arg<0>::Type,
         typename JoinPoint::Result > theResults;
      JoinPoint::Arg<0>::Type *arg_ptr =
         (JoinPoint::Arg<0>::Type*) tjp->arg(0);
      JoinPoint::Arg<0>::Type arg = *arg_ptr;
      if( theResults.has( arg ) ) {
         result_ptr = tjp->result();
         *result_ptr = theResults.get( arg );
      } else {
         tjp->proceed();
         result_ptr = tjp->result();
               theResults.add( arg, *result_ptr );
      }
   }
};
```

E. COUNTING ASPECT CACHE

```
template < typename Key, typename Value>
class CountingAspectCache :
public AspectCache <Key,Value> {
```

```
protected:
   int hits, misses;
   std::string joinPointName;
public:
   CountingAspectCache(std::string jpName) :
      AspectCache<Key,Value>()
      {hits=misses=0; joinPointName = jpName;}
   ~CountingAspectCache() {
      std::cerr << "Deleting CountingAspectCache,"
                << " jp=" << joinPointName
                << " hits="  << hits
                << " misses =" << misses
                << std::endl;
   }
   void add(Key& k, Value& val) {
      misses ++;
      AspectCache<Key,Value>::add(k,val);  }
   Value& get(Key& k) {
      hits++;
      return AspectCache<Key,Value>::get(k); }
};
```

F. OBJECT IDENTITY EXAMPLE

```
/* Simple C-style struct, and a
   function that calculates its area */
struct Square {
    int x1,y1,x2,y2;
};
void InitSquare(Square* s, int x1,int y1,
                int x2, int y2) {
   s->x1 = x1; s->y1 = y1;
   s->x2 = x2; s->y2 = y2;
}
int AreaOfSquare(Square *s) {
   int w = s->x1 - s->x2;
   int l = s->y1 - s->y2;
   int area = w*l;
   if(area<0)
      area = -area;
   return area;
}
int main() {
   Square *s1,*s2,*s3,*s4;
   Square s5;
   s1 = (Square*) malloc(sizeof(Square));
   InitSquare( s1, 0,0, 2,2);
   s2 = (Square*) malloc(sizeof(Square));
   InitSquare( s2, 3,3, 4,4);
   s3 = (Square*) malloc(sizeof(Square));
   InitSquare( s3, 0,0, 2,2);
   /*s3 and s1 have the SAME contents but
     are different memory objects */
   InitSquare( &s5, 4,4, 7,7);
   s4 = &s5;  /* s4 is a pointer to s5 */
   /* consider a sequence of calls that call
      AreaOfSquare while cached by an aspect*/
   int y = AreaOfSquare(s1);
   y = AreaOfSquare(s2);
   y = AreaOfSquare(s3);
   y = AreaOfSquare(s4);
   y = AreaOfSquare(&s5);
}
```

Coordination as an Aspect in Middleware Infrastructures

Mercedes Amor, Lidia Fuentes, Mónica Pinto
Departamento de Lenguajes y Ciencias de la Computación,
University of Málaga, 29071 SPAIN
{pinilla,lff,pinto}@lcc.uma.es

ABSTRACT

In this paper we discuss the shortcomings derived from having coordination and computation tangled in the same software entities and from having coordination protocols scattered through the several components participating in an interaction. We show a possible solution to this problem by using aspect-oriented techniques to separate coordination as an independent entity in middleware infrastructures.

Keywords
Coordination, AOSD, MultiTEL, CAM/DAOP, MALACA

INTRODUCTION

Developing distributed applications can be seen as the combination of two distinct activities: a computing part that comprises programming a number of entities (objects, components, agents, web services) involved in manipulating data, and a coordination part responsible for the communication and cooperation between these entities.

Given a set of possibly heterogeneous computational entities, the purpose of the coordination paradigm is to provide mechanisms and primitives to specify the synchronized interaction for putting all these components together, and make them interact in such a way that form a single application. This paradigm provides a clean separation between individual software components and their interactions within their overall software organization. This separation, together with the high-level abstractions offered by coordination models, allows viewing computational entities as black boxes, promoting their reuse in different applications.

Most standard middleware infrastructures directly support some kind of coordination mechanisms based on traditional coordination models such as the publish-and-subscribe, the tuple-based and the blackboard models [1]. However, such mechanisms are not sufficient for managing complex interaction protocols, typically comprising several lines of code. For instance, in the publish and subscribe coordination model, the support for expressing patterns about distributed events and algorithms for detecting correlations among these events are still largely unexplored [2]. By failing to support separate abstractions for representing such complex protocols, most standard middleware infrastructures force programmers to distribute and embed them inside the interacting components..

In this paper we propose a possible solution to this problem encapsulating the coordination among a set of software entities in an aspect. The coordination aspect is defined as an entity that encapsulates the interaction pattern or coordination protocol that governs the communication and interchange of information among two or more software entities. A coordination protocol can be defined as the list of messages and/or events that a software entity is able to send and receive, along with a set of coordination rules that state the order in which messages and events must be interchanged by the participant entities of the interaction. A coordination protocol can be specified by a STD (State Transition Diagram), or any other similar formalism. Other works for enhancing traditional coordination model by embedding the description of coordination information in a third-party entity are starting to appear [2,3,4,5].

After this introduction the paper is organized as follows. Next section briefly describes the state of the art of coordination in component-based and multi-agent domains. We continue with our motivation to separate coordination as an aspect and with the solutions we have adopted to separate coordination in MultiTEL [6], CAM/DAOP [7] and MALACA [8]. Some initial discussion about how coordination might need to interact with other aspects in the middleware infrastructure is presented in the following section. Finally, we present the conclusions of the paper.

STATE OF THE ART

Coordination plays a central role in technologies used to develop distributed applications such as component technologies, web services and agent technologies.

With respect to component-based technologies, they usually provide coordination mechanisms based on the publish-and-subscribe coordination model. Examples of these are CORBA, CCM/CORBA and J2EE. The main advantage of this mechanism is that communication is anonymous. Therefore, suppliers and consumers of events are decoupled among them due to the use of a third-party object that acts as intermediary between them. However, by using this mechanism software developers do not achieve a complete separation between computation and coordination for two main reasons: (1) the coordination model is spread throughout the objects acting as suppliers, consumers and event channels. That is, suppliers do not simply throw events that are intercepted by the middleware infrastructure and managed by the coordination entity. Instead, suppliers and consumers need to create and/or localize the event channels, including code related to the particular coordination model they follow as part of the code of the objects that implement consumers and suppliers; (2) the event channel does not usually provide support to encapsulate complex interaction protocols. In consequence, this service is mainly suitable for applications that just need to be aware of "change notifications". Basically, a supplier produces an event and all the consumers registered in the event channel will receive the notification that such event was produced. The third-party objects in the Jini Distributed Event Service are an exception to the latter shortcoming. The reason is that Jini does not impose anything about the implementation of third party objects. They just need to implement the interfaces to register or to notify events but do not need to extend a particular object of a specified type. Therefore, Jini provides an event based programming model plus the possibility of encapsulating a coordination protocol in the third party objects. As commented before, other proposals that extends the publish and subscribe model to address the coordination of activities between decoupled components are [2,3,4]. Finally, ObjectPlace [5] is another proposal extended the traditional tuple space coordination model with support for role-based coordination between individual components.

Coordination also plays an important role in web services that use a loosely coupled integration model to allow flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration. However, the use of standards such as SOAP, WSDL and UDDI and interaction following a loosely coupled is not enough for such integration. Systems integration requires an additional layer able to describe and perform web services composition and orchestration. The latter term is the description of interactions and message flow between services in the context of a business process. This concept is not new; in the past it has been called workflow. Until now, different XML-based languages have been introduced to cover web services orchestration where two of the more well-known are the Web Services Choreography Description Language (WS-CDL) [9] and the Web services Business Process Execution Language (WS-BPEL) [10]. WS-CDL describes the set of rules that explains how different web services may act together, and in what sequence, giving a flexible and integral view of the process. WS-BPEL enables the composition of existing web services into a more complete web service. The composition is defined in terms of a workflow process consisting of a set of activities and the composition itself is exposed as a web service.

Nowadays, Multi-Agent Systems (MAS) are an effective paradigm for the design and implementation of complex software applications. Agent-based applications are understood as a system consisting of autonomous agents whose interactions are coordinated through an organizational structure [11]. The necessity of coordinating agents in MAS relies on the common idea that an agent should be able to engage in, possibly, complex communications with other agents in order to exchange information or to ask their collaboration in pursuing a goal. Currently two coordination models are the most representative and effectively used to realize agents coordination mechanisms: Interaction protocols and Shared Dataspaces. A large portion of the agent community, which includes the standardizing organism FIPA, considers coordinating agents using interaction protocols, i.e. predetermined patterns of interaction. This approach is based on considering coordination essentially as a problem of communication. Thus the Agent Communication Language (ACL) plays a fundamental role, providing a means to achieve a higher-level of interoperability between agents. We can find in the agent models supporting this coordination model that the functional part of the code of an agent is interleaved with code that is there only to support coordination. This dependency derived from the intermingled code makes difficult the reuse of the agent functionality in other applications and platforms.

MOTIVATION

The conclusion is that coordination models provide support for loosely coupled interaction, mainly focusing on decoupling the source and the target of the communication. However, they do not achieve a complete separation among computation and coordination, since they do not provide support for the description of complex interaction protocols. As stated before, a complex interaction protocol specifies not only the information interchanged by coordinated entities but also the coordination rules.

To illustrate this shortcoming we will focus on an auction system and, concretely, in the simplified interaction between the seller and the buyer. In this example, Figure 1 shows the entities participating in an auction, while Figure 2 describes a scenario of the interaction among them. However, when the system is finally implemented using, for instance, a component based approach, the coordination information shown in Figure 2 is usually lost. That is, the coordination protocol among buyers and sellers is directly hard coded as part of the implementations of the *Seller* and the *Buyer* components. As a consequence, there may be situations in which a change in the interaction protocol requires changing the component implementations.

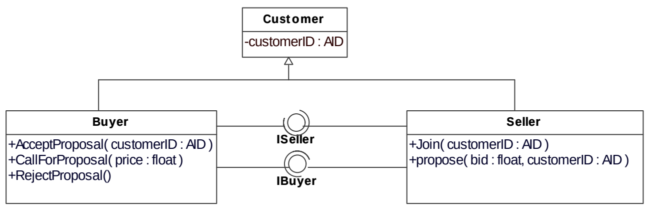

Customer
-customerID : AID
Buyer
+AcceptProposal(customerID : AID)
+CallForProposal(price : float)
+RejectProposal()
ISeller
IBuyer
Seller
+Join(customerID : AID)
+propose(bid : float, customerID : AID)

Figure 1. Part of the design of an Auction System.

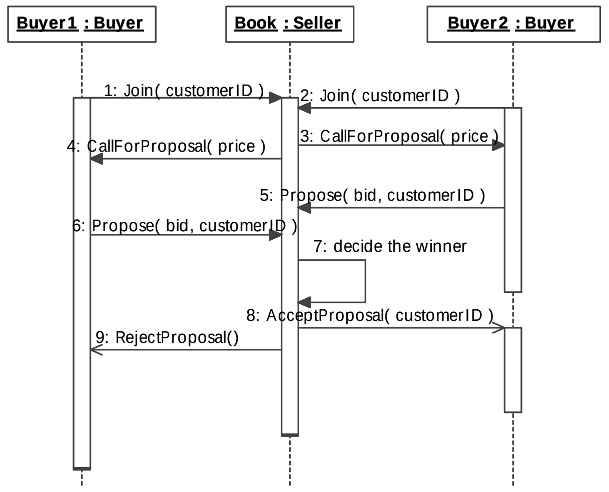

Buyer1 : Buyer
Book : Seller
Buyer2 : Buyer
1: Join(customerID)
2: Join(customerID)
3: CallForProposal(price)
4: CallForProposal(price)
5: Propose(bid, customerID)
6: Propose(bid, customerID)
7: decide the winner
8: AcceptProposal(customerID)
9: RejectProposal()

Figure 2. Buyer – Seller Interaction Protocol.

In Figure 2 when an auction finalizes, the *Seller* component decides the winner bid (step 7) and notifies to just one of the *Buyer* components that his/her bid won, using the *AcceptProposal(customerID)* operation (step 8). The other buyers are notified that they did not win invoking the *RejectProposal(customerID)* operation (step 9).

Let us suppose now that all the buyers have to be notified about which is the winner. This is a change in the coordination among the components and, with the current approach, implies to modify the implementation of the *Seller* component. Thus, the seller has now to invoke the *AcceptProposal(customerID)* operation over all the buyers.

This shortcoming may be overcame if the seller and the buyers use, for instance, the publish-and-subscribe interaction style. In this case, components register in a common communication channel. The use of a third-party object facilities the 1-to-many communication. This means that the seller just publishes an anonymous *AcceptProposal(customerID)* event and all the buyers subscribed to that event will receive the notification.

Another change may be to make the auction *public* (one in which all bids are shared among the participants). We desire to (re)use the *Buyer* and *Seller* components in Figure 1 and Figure 2. In this case the *Seller* component should have to notify the *Buyer* components that a new bid is proposed, and this requires modifying its implementation. Concretely, after receiving the *Propose(bid,customerID)* operation invoked by *customerID* buyer, the Seller has to notify the bid to the rest of buyers using the *CallForProposal(bid)* operation. Once again this is part of the interaction protocol codified crosscutting the functionality of the Seller component. In this case, even using the publish-subscribe interaction style it is needed to change the implementation of the seller component. The reason is that this change modifies the interaction protocol, which is tangled with computation and scattered throughout the participant components. The solution to this problem comes across not including the coordination rules as part of the components. This information should be included in the third-party object of coordination models. However, traditional coordination models do not provide support for this.

The extension of a coordination model with the necessary support to encapsulate complex interaction protocols may avoid to hard code this information as part of the coordinated entities. However, most of the realizations of a coordination model for particular technologies introduce some non desirable dependencies among the coordinated entities. For instance, the event service in CORBA follows a publish-and-subscribe coordination model. However, its implementation uses RPC (Remote Procedure Call). This implies that the channel is a CORBA object and it has to be located by the components before using it.

This problem can be alleviated if the responsibility of locating the adequate channels falls to the middleware infrastructure. This approach is followed in the CCM model of CORBA, where components just throw events and it is the container the responsible of managing channels.

The next section will show how AOSD mechanisms provide a natural solution to solve the limitations of the coordination models discussed in this section.

SEPARATING COORDINATION IN MIDDLEWARE INFRASTRUCTURES

In this section we describe our experience separating coordination as an aspect in three different middleware infrastructures we have developed: MultiTEL, CAM/DAOP and MALACA. Readers can obtain detailed information of these middleware infrastructures in [2,3,4].

A common feature of these infrastructures is that coordination is separated from computation and moved into a coordination aspect (Figure 3, label 1). Other important feature is that the weaving information is not part of the definition of components or aspects. Instead, it is explicitly described and allocated in the middleware infrastructure (Figure 3, label 2), which is the responsible of weaving components and the coordination aspect at runtime. This is very useful for implementing applications in open systems, in which components evolve over time. Thus, components do not need to choose or localize the proper coordination aspect, since this is the middleware infrastructure responsibility. This characteristic increases the reusability of components in different contexts. As shown in Figure 3, both load-time weaving and runtime weaving mechanisms may be provided, with divergent outcomes on flexibility and performance.

Other advantage of separating the coordination aspect is that it is easier to control the different states of a complex interaction (Figure 3, label 3).

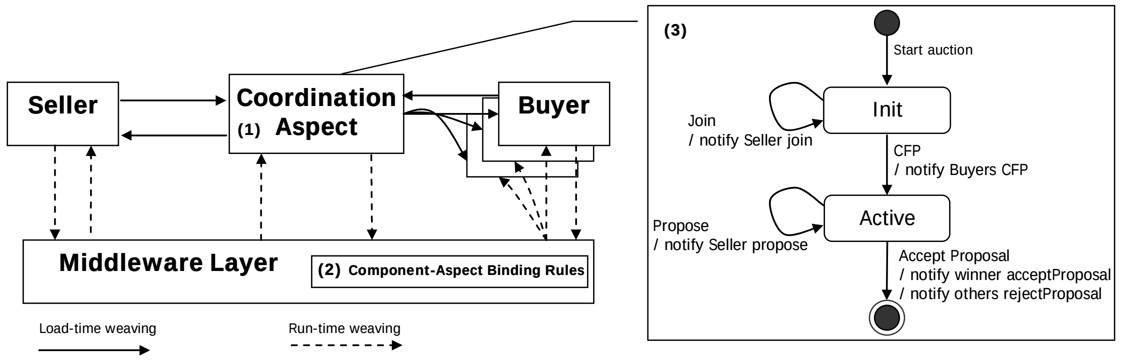


Figure 3. Separation of coordination in Auction System

Going back to our example, let us suppose that it is needed to control that once the auction begins (the *Seller* component sends the first *CallForProposal()* operation), no new buyers can join the auction. This kind of constraint can be easily satisfied with a coordination aspect that encapsulates a protocol with different states.

Coordination in MultiTEL

MultiTEL (Multimedia TELecomunication services) is a complete development solution that applies component and framework technologies to deploy multimedia and collaborative applications. Essentially, MultiTEL provides separation of concerns between computation and coordination by encapsulating computation into Components and locating coordination into Connectors. Therefore, the architecture of a MultiTEL application can be seen as a collection of components that interact through connectors.

In MultiTEL, components are passive entities that react to external stimuli by propagating events. Connectors are abstractions of complex coordination patterns that implement coordination protocols. By handling the events propagated by external components a connector can coordinate the execution of the components participating in the same interaction or task. The interaction protocol encapsulated in connectors is specified as a STD and it is implemented using the State Design Pattern in Java.

Another important characteristic of MultiTEL is that the weaving information to connect components and connectors is specified separately from components and connectors and stored in the middleware infrastructure. Weaving information is specified in a Service Description Language (LDS). Afterwards this information is provided to the distributed middleware infrastructure that will use it at runtime to weave components and connectors dynamically. This increases both components and connectors reusability and evolution and it provides a powerful mechanism for late binding among them.

Coordination in CAM/DAOP

CAM/DAOP (Component-Aspect Model/Dynamic Aspect-Oriented Platform) is a model and a middleware infrastructure that combines the benefits of both component-based and aspect-oriented disciplines in the development of complex distributed applications.

CAM/DAOP is an extension of MultiTEL and shares with it features such as the explicit description of the application architecture and the use of this information by a middleware infrastructure to perform the late binding between components and aspects. Therefore, CAM/DAOP also shares with MultiTEL the advantages introduced by these features. There are also some differences. The main difference is that the main entities of CAM/DAOP are Components and Aspects. This makes possible to separate any extra-functional properties that result to be tangled or scattered throughout the components in an application. This has the advantage that we increase even more the reusability and evolution of both Components and Aspects.

Another difference is that in CAM/DAOP Components can interact using both a message-based and an event-based programming models. The handling of events is resolved at runtime by a coordination aspect that plays the same role than the Connector in MultiTEL. Additionally, the coordination aspect can intercept messages among components. In this case, the coordination aspect introduces an additional advantage since it can be also used as an *adapter* for resolving component incompatibilities and integration problems.

A final advantage is how the coordination aspect is implemented. Concretely, the interaction protocol encapsulated in the aspect is XML-based specified conforming to an XML Schema. Thus, the implementation of the aspect consists on building an interpreter for these protocols, instead of providing different implementations of the aspect for each interaction protocol. Using this approach the application can evolve at runtime only by changing the XML description of the coordination protocol.

Coordination in MALACA

MALACA provides separation of concerns in the scope of MAS and more concretely in the context of agent internal architecture. MALACA is a component-based and aspect-oriented agent model. Inside the agent architecture, agent functionality is provided by components and it is separated from its communication, which is modelled by aspects.

Three main aspects are recognized as mandatory since they cover the basic agent communication functions in a FIPA-platform: Distribution, Coordination through an interaction protocol and ACL representation. Aspects are weaved at runtime by a third party component (internal weaving infrastructure) of the architecture following a set of aspect composition rules. Weaving is performed when the agent communicates, that is when it sends and receives messages.

As said before, agents of MAS mostly use interaction protocols to coordinate. This way they achieve a separation between how agent coordinates and how it performs its tasks. However, it has an impact on the internal development of agents, since most object-oriented infrastructures tangle in the same architectural unit computation and coordination, this time inside the agent.

In the MALACA architecture, each ongoing conversation is coordinated by a coordination aspect. To perform protocol-compliance the coordination aspect uses a XML-based description of the interaction protocol, which is given to the coordination aspect during instantiation. Currently MALACA provides a common implementation of the coordination aspect that is able to parse and use the XML description of an interaction protocol that follows a concrete XML Schema. This feature avoids implementing a coordination aspect for each interaction protocol the agent has to support. Each role participating in the interaction is described as an STD, in terms of states and transitions. The transition from a state to another carries out the execution of the agent functionality, which is also included as part of the XML-based STD description.

INTERACTION WITH OTHER ASPECTS

The most relevant relationship of the coordination aspect with other aspects at the middleware infrastructure level is with the distribution aspect. Let us consider that a distribution aspect encapsulates both localization and communication sub-concerns [12]. Therefore, when an anonymous interaction occurs – i.e. a component throws an event: (1) the coordination aspect is the responsible to manage the events thrown by components to determine which component(s) is/are the recipient of the information and forwards a message to it/them, and (2) once the targets are decided, the interaction is considered as an explicit component to component interaction and thus requires interacting with the localization and the communication sub-concerns of the distribution aspect. The relationship between the coordination and distribution aspects at the infrastructure level is also present in the agent domain. In the scope of the agent internal infrastructure, the distribution aspect encapsulates the access to a concrete agent platform through its done message delivery.

CONCLUSIONS AND FUTURE WORK

In this paper we have shown that it is possible to use aspect-oriented techniques to improve the separation between computation and coordination in middleware infrastructures. The main benefits of the proposed solution are: (i) facilitate the reusability of components in different contexts; (ii) well-designed support for the definition of complex interaction protocols; (iii) increase the maintainability and adaptability of final applications.

Comparison with other approaches enhancing traditional coordination models to provide support for the definition and management of the orchestration among a set of interacting components is needed and it is planned as part of our future work. Performance evaluation of the presented approach will also be made.

ACKNOWLEDGMENTS

This work has been funded in part by the Spanish CICYT project with number TIN2005-09405-C02-01 and in part by the AOSDEurope NoE with number IST-2-004349.

REFERENCES

1. Papadopoulos, G. A., et al. “Coordination models and languages”, Advances in Computers 46 pp. 329-400. 1998.
2. Li, G et al.” Composite Subscriptions in Content-Based Publish/Subscribe Systems”, Middleware 2005, LNCS 3790, pp. 249–269, 2005.
3. Pietzuch, P. R. et al. ”Composite event detection as a generic middleware extension”, IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks, January/February 2004.
4. Ulbrich, A. et al. “Programming abstractions for contentbased publish/subscribe in object-oriented languages”. In CoopIS/DOA/ODBASE (2), pages 1538–1557, 2004.
5. Schelfthout, K. et al. “Middleware for protocol-based coordination in dynamic networks”, in 'MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing', ACM Press, New York, NY, USA, pp. 1—8, 2005.
6. Fuentes, L., et al. “Coordinating distributed components on the web: An integrated development environment”, Software Practice & Experience, 31(3):209-233, 2001.
7. Pinto, M., et al. A dynamic component and aspect oriented platform, The Computer Journal, 48(4):401-421, 2005.
8. Amor, M., et al., Training compositional agents in negotiation protocols using ontologies, Journal of Integrated Computer-Aided Engineering, 11(2):179-194, 2004.
9. W3C. “Web Services Choreography Description Language Version 1.0”, http://www.w3.org/TR/ws-cdl-10/
10. OASIS Open, Inc. “Web Services Business Process Execution Language Version 2.0. “, http://www.oasis-open.org/
11. Omicini, A. et al, eds. Coordination of Internet Agents. Models, Technologies, and Applications. Springer, 2001
12. Loughran, N., et al. Requirements and Definition of AO Middleware Reference Architecture, http://www.aosd-europe.net/documents/d21.pdf, 2005

AspectJ for Multilevel Security

Roshan Ramachandran
Computer Science
Victoria University of Wellington, NZ
ramachrosh@mcs.vuw.ac.nz

David J. Pearce
Computer Science
Victoria University of Wellington, NZ
djp@mcs.vuw.ac.nz

Ian Welch
Computer Science
Victoria University of Wellington, NZ
ian@mcs.vuw.ac.nz

ABSTRACT

A multilevel security (MLS) system has two primary goals: first, it is intended to prevent unauthorised personnel from accessing information at higher classification than their authorisation. Second, it is intended to prevent personnel from declassifying information. Using an object-oriented approach to implementing MLS results not only with the problem of code scattering and code tangling, but also results in weaker enforcement of security. This weaker enforcement of security could be due to the inherent design of the system or due to a programming error. This paper presents a case study examining the benefits of using an aspect-oriented programming language (namely AspectJ) for MLS. We observe that aspect-oriented programming offers some benefits in enforcing MLS.

1. INTRODUCTION

Multilevel security (MLS) [3] was developed by the US military in the 1970's to allow users to share some information with certain classes of user while preventing the flow of sensitive information to other classes of user. MLS achieves this by labelling data with classifications and assigning fixed clearances to users. The relationship between the classifications and clearances are used to determine if access by a user to data is permitted or disallowed. Note that the terms "clearance" and "classification" in the context of military systems refers to the security levels top secret, secret, confidential and unclassified. For example, a secret military plan will have different levels of information that correspond to various ranks.

MLS, once thought only to be significant to military systems, is also used in other domains like trusted operating systems, and in grid applications, where administrative users can set multilevel policies on their applications, thereby providing a fine grained control on the community users.

The Bell-LaPadula security model (BLP) [3] is a formalisation of MLS. BLP defines two rules which, if properly enforced, have been mathematically proven to prevent information at any given security level from flowing to a "lower" security level. These rules are called No Read Up (NRU) and No Write Down (NWD). The NRU rule states that a subject cannot read an object that has a higher security level. Whereas, NWD states that a subject cannot write to an object that has a lower security level.

To enforce MLS in a system, access control involves three entities: an object (the entity to which access is requested), a subject (the entity requesting access) and a reference monitor [1]. It is the responsibility of the reference monitor to check whether every access by the subject to an object is validated by the rules (i.e. BLP) set up by the MLS. To help in taking this decision, an object is assigned a classification and the subject is assigned a clearance.

Object-Orientation is an attractive approach to implementing MLS because objects are a natural way to represent system data and well-defined interfaces are a natural place to enforce access. However, there are several serious problems with this approach. Firstly, an object-oriented approach to implementing MLS can result in code tangling and scattering if access control is manually inserted into methods that read or write to sensitive objects. Although patterns such as the Proxy pattern can be used to structure object-oriented programs so that access control code is localised there are still maintenance problems because the relationship between the proxy and its object must be maintained at all times to prevent leakage of a pointer allowing uncontrolled access. More seriously, an object-oriented approach does not provide any support for preventing the introduction of security holes through poor design or bad programming. In other words, object-oriented programming does not support programmers in achieving stronger enforcement of security.

The aspect-oriented [10] approach is often perceived as an improvement over the conventional object-oriented approach in dealing with the issues of code tangling and crosscutting. Most previous work on implementing security using AOP have focused upon the concerns of reducing tangling and crosscutting [10, 12]. This paper describes how an AOP language (in particular AspectJ [2, 9, 10]) can actually go further to achieve stronger enforcement of MLS.

2. MOTIVATION

Figure 1 shows the class diagram of a payroll system with MLS. The payroll system is used by administrators and managers to track employee information, salary details, hourly rate etc. At the heart of the diagram is the PayrollSystem class, this implements functionality for adding new employees (Employee), changing their hours worked (WorkInfo) and changing their hourly rate (PayInfo). It is assumed that users of varying clearances access the payroll system and we want to control information flows between them. Note that we are not concerned with the problem of maintaining the integrity of the system — this would require the enforcement

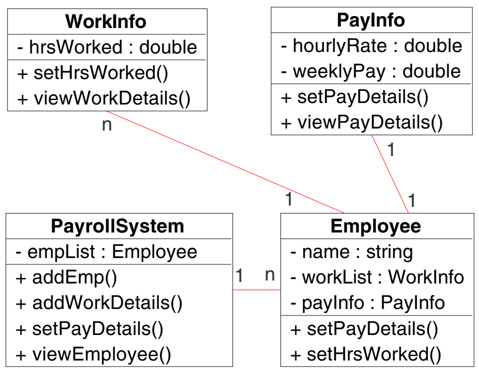


Figure 1: Class diagram of a payroll system

of a BIBA-style policy [4].

Consider a scenario where there are two types of users with different rights: managers and employees. Managers are allowed to view all information in the system, while employees can only view the non-sensitive information contained within the Employee class. Using BLP to enforce this requires setting up the system of clearances and classifications as follows: Employee instances are assigned a "low" classification, WorkInfo and PayInfo are assigned "high" classifications, managers are assigned "high" clearances and employees are assigned "low" clearances.

With this particular choice of classifications and clearances we can see that our access requirements described above are satisfied by enforcement of the BLP policy. The NRU rule restricts employees to reading Employee objects because all employees have clearances lower than the classifications of all objects except for Employee objects. While the NRU allows managers access to all objects because the manager's "high" clearance is equal or higher than all the classifications of all the other objects in the system. Furthermore, information cannot be passed by managers to employees because managers are prevented from writing to objects that can be read by employees (namely the Employee objects) by application of the NWD rule.

An object-oriented approach to implementing the payroll system would be to define a component that behaves as a reference monitor. This would take three arguments: the user's clearance, target object's classification and whether the requested action is a read or write operation. The reference monitor is responsible for deciding, based on the BLP rules, whether or not the user is authorised to access the object. For this to work, the programmer must either: insert access control code in all methods that read or write the protected objects (e.g. `setPayDetails()`); or create proxies containing the access checking code for the protected objects. Either way, he/she must *manually* identify those methods which need access control. This, of course, leaves open the possibility of programmer error — that some method may read/write a protected object without access control simply because the programmer missed it. Since humans are fallible, we believe it follows that an object-oriented approach is an inherently error prone way of enforcing MLS.

The problem is that we want code to be generated and added automatically based upon a declarative specification that defines what is a controlled information flow. Ideally this would be as finer-grained as possible to allow greatest control. Aspect-Oriented Programming offers such an approach and, we argue, can provide a stronger enforcement of security than is possible through conventional Object-Oriented Programming. This stems from the fact that AOP languages allow quantification over a set of *join points*. For example, we can capture all reads and writes to a given set of objects automatically using field `set` and `get` pointcut designators (as in AspectJ). Once we intercept a field read or write *joinpoint*, we can use *before advice* to perform the authorisation. This way, we avoid having to manually identify which accesses need authorisation. This guarantee provided by the quantification property of aspect-oriented languages is very difficult to achieve in traditional object-oriented languages.

```
abstract aspect BLPPolicy {

 abstract pointcut read(Object o);
 abstract pointcut write(Object o);

 // No Read Up (NRU)
 before(Object o) : read(o) {
  int oc = classification(o);
  int sc = clearance(Thread.current());
  if(sc < oc && sc != TRUSTED) {
   throw new SecurityException();
 }}

 // No Write Down (NWD)
 before(Object o) : write(o) {
  int oc = classification(o);
  int sc = clearance(Thead.current());
  if(oc < sc && sc != TRUSTED) {
   throw new SecurityException();
 }}

 abstract int clearance(Object o);
 abstract int classification(Object o);
}}
```

Figure 2: The generalised BLPPolicy aspect.

An AspectJ implementation of the BLP policy for the payroll system is shown in Figures 2 and 3. The two rules of the BLP model are encoded in an abstract aspect, where the concepts reading and writing are represented, but left unspecified. In this way, the policy can be (re)used simply by extending `BLPPolicy` and declaring these concepts appropriately (as done in Figure 3). In the implementation, we have simply used `Integer`s for the clearance and classification values; a more general approach might be to use an (abstract) lattice which the user would define appropriately. The use of the special `TRUSTED` status will be discussed later.

Our implementation shares some similarity with the AspectJ implementation of the Subject-Observer protocol [7]. One difference is that we have not made the roles (i.e. subject and object) explicit using interfaces as role markers (as done in [7]). This would have simplified our pointcut definitions to some extent. However, unlike application classes, AspectJ does not allow system classes to be modified, forcing us to use the somewhat less elegant approach.

3. CASE STUDY: FTP SERVER

This case study focuses on implementing MLS in a non-trivial (approx 20 classes), third-party application called jFTPd [8]. This im-

```
aspect PayrollPolicy extends BLPPolicy {

 pointcut read(Object o) :
   (get(* WorkInfo.*) || get(* PayInfo.*) ||
    get(* Employee.*)) && target(o);

 pointcut write(Object o) :
   (set(* WorkInfo.*) || set(* PayInfo.*) ||
    set(* Employee.*)) && target(o);

 int clearance(Object o) {
  // lookup thread clearance, possibly
  // resulting in user authentication
 }
 int classification(Object o) {
  // lookup object classification
}}
```

Figure 3: The payroll system's security policy.

plements an ftp server and we chose this as a case study for several reasons:

- An FTP server is a good example of an application requiring MLS enforcement. Users can read (download) and write (upload) files to the ftp server. Confidentiality needs to be taken care of to ensure secret files are only read by users with proper clearance. Likewise, users must be prevented from downgrading a file's classification (accidentally or otherwise).

- jFTPd is implemented in the object-oriented paradigm and has a built-in (albeit basic) security policy. This allows some comparison with the AOP implementation of MLS.

- jFTP is a third-party application with sufficient complexity that it might expose any limitations on the reusability of our abstract BLP policy aspect.

3.1 Overview of jFTPd

jFTPd is a Java implementation of an FTP server. jFTPd is multithreaded, so many users can connect and transfer files simultaneously. Figure 4 shows the original class diagram of jFTPd. For clarity, only classes concerning the access control for writing or reading files and basic ftp authentication are shown. The main class is *FTPHandler* which takes care of incoming connection setup requests. For every request, `FTPHandler` creates a new thread (i.e. `FTPConnection`) and assigns the incoming connection to it. This services all FTP commands from the client by delegating them to the `doCommand()` method (in `FTPConnection`). This in turn delegates onto a number of command-specific processing methods.

`FTPConnection` maintains a session for each connected user (i.e. `FTPUser`) and an object (i.e. `FTPSecuritySource`) which acts as a reference monitor containing the access control logic. Calls to this are scattered throughout the methods of `FTPConnection` to restrict certain types of access. Specifically, users may only access files in their home directory, whilst the anonymous user may be allowed (depending upon the configuration) certain access to files in the anonymous directory.

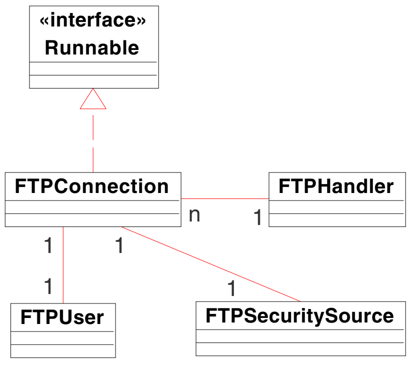


Figure 4: Original structure of jFTPd

3.2 AspectJ implementation

Our AspectJ implementation of the MLS security policy proceeds, as before, by extending `BLPPolicy`. The goal is to intercept all read and write operations to perform the required authorisation checks. But, what are the read and write operations in this case? In the previous example, the objects (object as in MLS) were `Employee`, `WorkInfo` and `PayInfo` instances, while the subjects were threads. In this case, however, the objects are files and the subjects instances of `FTPConnection`. Therefore, instead of using field `set` or `get` pointcuts to capture read or write operations, we must intercept all attempts to read or write files.

Figure 5 outlines the implementation. Several details have been omitted for brevity. The read and write pointcuts are defined in terms of the Java standard library calls for reading and writing streams. We must intercept on, for example, `InputStream` rather than `FileInputStream`; otherwise, accesses to an instance of the latter with a static type of the former will be missed. When a read or write to some stream type is intercepted, we have access to the stream object in question. From this, we must determine the true subject (i.e. the actual file being manipulated). Unfortunately, the Java APIs do not permit this directly (e.g. we get cannot get back a file name from an instance of `FileInputStream`). To overcome this, we intercept the creation points of these streams and manually associate with them the file name in question.

Threads other than instances of `FTPConnection` are regarded as having system/administration roles and given the special `TRUSTED` status (see Section 4.5 for more on trusted subjects). When a read or write operation is intercepted, we must determine the true subject (i.e. the user) in order to determine the appropriate clearance level. This information is maintained by `FTPConnection` as part of the original implementation. However, as outlined in Figure 4, instances of `FTPConnection` implement the `Runnable` interface (rather than extending `Thread`). This poses a problem as, although we can determine the thread responsible for a read/write operation via `Thread.currentThread()`, the Java API again provides no way to obtain the instance of `FTPConnection` it is associated with. As before, we overcome this by intercepting creation points for `FTPConnection` objects and maintaining their thread associations explicitly.

4. DISCUSSION

In this section we discuss our approach's granularity of control, how authentication is accommodated, the problems of covert channels, how improved modularity and maintainability are achieved,

```
public aspect JFTPdPolicy extends BLPPolicy {
 Map<Object,String> objects = ...;
 Map<Thread,FTPConnection> subjects = ...;

 pointcut read(Object o) : target(o) &&
  (call(* InputStream.read*(..)) ||
   call(* Reader.read*(..)) ||
   ...;

 pointcut write(Object o) : target(o) &&
  (call(* OutputStream.write*(..)) ||
   call(* Writer.write*(..)) ||
   ...;

 // *** OBJECT intercepts ***
 after(String s) returning(Object o): args(s)
   call(FileInputStream.new(String)) {
  objects.put(o,s);
 }

 after(File f) returning(Object o): args(f)
  : call(FileInputStream.new(File)) {
  objects.put(o,f.getName());
 }

 ... // other object creation intercepts

 // *** SUBJECT intercepts ***
 after(FTPConnection f) returning(Thread t)
  : call(Thread.new(Runnable)) && args(f) {
  subjects.put(t,f);
 }

 ... // other subject creation intercepts

 // *** BLPPolocy methods ***
 int clearance(Object o) {
  FTPConnection f = subjects.get(o);
  if(f == null) { return TRUSTED; }
  else {
   String user = f.getUser().getUsername();
   ... // lookup clearance for "user"
 }}

 int classification(Object o) {
  String file = objects.get(o);
  if(f == null) { return TRUSTED; }
  else {
   ... // lookup classification for "file"
 }}
 ...
}
```

Figure 5: An AspectJ-based MLS security policy for jFTPd.

the issue of trusted subjects and how to appropriately handle security failures.

4.1 Granularity of Control

In the case study we enforced security by intercepting system calls rather than application calls. For example, we intercepted file reads and writes rather than defining a read or write pointcut that intercepted application methods related to file access. The benefit of our system call approach is we get complete assurance that every read or write operation on a file has been validated by following the rules of No Read Up and No Write Down. Thus, we are not affected by the introduction of new application-level methods because these will use system calls to interact with the file system and therefore will be constrained by the BLP security policy. However, we are affected by changes to system classes that introduce new system calls. For instance, any newly introduced system API that deals with the reading or writing to files needs to be reflected in the existing pointcuts used to control access to system classes. However, we consider this to be less of an issue than the problems of traditional object-oriented languages, for several reasons: firstly, it's unlikely that system APIs will change frequently: secondly, there are fewer things which need to be identified by the programmer.

4.2 Authentication

In the case study we used existing jFTP authentication mechanisms to ensure that users are authenticated before making authorisation decisions. An open question is how to implement authentication within our framework. For example, should authentication be treated as a separate aspect or part of the authorisation aspect because it is a fundamental precondition for proper application of a security policy? The advantage of a separate aspect is that authentication could be easily plugged in and out, for example we could swap a local authentication mechanism for a federated authentication mechanism.

4.3 Covert Channels

Our current implementation does not address the problem of covert channels, that is information flows that are not controlled by a security mechanism. For example, a jFTP user with a "high" clearance could communicate with a user with a lower clearance by creating files with a "high" classification whose existence can be seen but the contents not read. This could be used to signal information read by the user with a higher clearance to the user with the lower clearance. To address this we could extend our implementation to hide the existence of files that cannot be accessed by the current user.

Another type of covert channel might arise if there are information flows not intercepted by our pointcuts. For example, in the case study we concentrated upon information flow via file reading and writing and assumed that no other information flows between threads was possible. However, what if we were wrong? Threads might have access to shared static fields and use these for communication. There are at least two solutions. We could use some form of static analysis to determine that flows between threads other than the ones we are dynamically checking do not occur. Unfortunately this might not be practical for all programs because of complexity. Alternatively, we could implement some form of dynamic pointcut for inter-thread information flow. This might be realized by intercepting all state changes (including variable reads/writes and parameter passing) and enforcing BLP. The disadvantage of this approach is the cost of intercepting all state changes. However, it may be possible to use it in conjunction with a more constrained form of static analysis to ensure enforcement is added only where threads may communicate with each other rather than throughout the application.

4.4 Modularity

By having the MLS access control logic in an aspect supports the well known AOSD notion of improved modularity and maintainability. This also makes it possible to have multiple authorisation aspects woven into the same base object. For example, in jFTPd, we could have a separate authorisation aspect executing after the `JFTPdPolicy` aspect to give finer control over anonymous users.

4.5 Trusted Subjects

There are sometimes legitimate reasons to relax the BLP policy for some subjects. Consider a listener thread that receives and dispatches requests to worker threads or a logging thread that updates a range of application objects. These threads may need to violate the BLP rules in order to function. Traditional MLS systems allow this by introducing the notion of trusted subjects who are allowed to violate the security rules.

In our case study this notion is satisfied because threads other than those associated with instances of `FTPConnection` are treated as trusted. We believe these threads are worthy of that trust simply because we have inspected their code to determine there are no violations of the BLP rules. This is not ideal, although it remains unclear how else such trust can be established.

4.6 Security Failures

Our current implementation raises a `SecurityException` when the BLP policy is violated. This may cause the client to crash if it does not expect this type of exception and it could also leave the application in an inconsistent state if the exception happens after a thread has updated one object but was denied access to another related object[1]. We can address the first problem by adding application-specific code to our aspects that create exceptions our client will be able to handle. The second problem is more problematic and may require the use of transactions to allow all-or-nothing semantics to be applied to controlled operations. The drawback of this is the requirement to understand the application semantics sufficiently in order to put all the security-related operations within a single transactional scope.

5. RELATED WORK

Implementing authentication and authorisation as crosscutting concerns for distributed object-oriented programs has already been addressed by several researchers, for example Lasagne [11], System K [12] and jFTPd [5, 6]. Lasagne implements an access control list style policy for a dating application. System K implements a MLS security policy for a third-party distributed editor. jFTPd implements an access control list style policy for a third-party FTP server.

These papers differ from our approach in that they use AOSD techniques with pointcuts that are tied to manually identified methods. This can lead to error if a method is missed or misidentified. Instead we concentrate on how to avoid this problem by determining pointcuts using fundamental properties of the application that are independent of functionality. For example, pointcuts tied to file update are better than pointcuts tied to specific methods believed to update files. This is an approach that was suggested by Welch and Stroud with respect to their implementation of MLS using a metaobject protocol[12].

[1]Consider a transfer of money between two different types of account.

6. CONCLUSION

In this paper, we have described an approach to incorporate MLS using aspects. In the case study, we used AspectJ to intercept Java library calls in order to provide a strong enforcement of MLS policy. By following this approach, we reduce the burden on programmers to correctly identify all the places in the code where authorisation is required. This quantification achieved by AspectJ is difficult to achieve in an object-oriented language.

We observe that even though the object-oriented paradigm provides a natural way to implement multilevel security, it falls short of preventing security flaws caused by bad programming or poor design. In this paper, we have demonstrated how aspects, in comparison to object-orientation, can guarantee better security assurance when implementing multilevel security.

7. REFERENCES

[1] J. P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, Oct. 1972.

[2] AspectJ home page. http://eclipse.org/aspectj/.

[3] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. *MITRE technical report, MITRE Corporation, Bedford Massachusetts*, 2997:ref A023 588, 1976.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[5] B. De Win, W. Joosen, and F. Piessens. AOSD & security: A practical assessment. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.

[6] B. De Win, W. Joosen, and F. Piessens. Developing secure applications through aspect-oriented programming. In *Aspect-Oriented Software Development*, pages 633–650. Addison-Wesley, Boston, 2005.

[7] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *In Proc. conference on Object-Oriented Programming Systems, Languages and Applications*, pages 161–173. ACM Press, 2002.

[8] Ftp server with remote administration. http://homepages.wmich.edu/ p1bijjam/cs555

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP 2001*, volume LNCS 2072, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.

[10] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[11] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. 23rd Int'l Conf. Software Engineering (ICSE'2001)*, 2001.

[12] I. S. Welch and R. J. Stroud. Re-engineering security as a crosscutting concern. *Comput. J*, 46(5):578–589, 2003.

Sharing of variables among threads in heterogeneous grid systems by Aspect-Oriented Programming

Laurent Broto
UMR 5505 IRIT
118 route de Narbonne
F-31062 TOULOUSE
CEDEX4
broto@irit.fr

Jean-Paul Bahsoun
UMR 5505 IRIT
118 route de Narbonne
F-31062 TOULOUSE
CEDEX4
bahsoun@irit.fr

Robert Basmadjian
UMR 5505 IRIT
118 route de Narbonne
F-31062 TOULOUSE
CEDEX4
basmadji@irit.fr

ABSTRACT

Nowadays, the need for grid systems is becoming more and more relevant to applications demanding either large amount of data storage or computation. In this paper, we propose an approach that permits thread's distribution in hetereogenous grid systems without any middleware layer or thread's API modifications so that the programmer feels comfortable and shares variables in complete transparency. In order to achieve this task, we will use the Aspect-Oriented Programming and Java's mechanism such as introspection and RMI.

Keywords

grid, thread, aspect, AspectWerkz, java, scalability, introspection

1. INTRODUCTION

A grid system is the collection of either homogeneous or heterogeneous computers called nodes which are interconnected to each other in order to enhance the overall system capabilities. Amongst these capabilities, the followings are the three principal ones[1]: enhanced data storage by aggregating the storage spaces of each node, rapid execution of computation-demanding applications by taking advantage of the computing power of every node and the aggregation of the bandwidths of the nodes in order to obtain a larger bandwidth for every connection destined for the outside of the grid's domain.

On the other hand, the applications of such grid systems must be either compiled at every node of the gird or written in a language portable to every node of the grid.

The realization of the first case is a tedious and costly task. Therefore, we have concentrated our efforts on the second case by taking Java as the programming language because of its multiplatform characteristic.

Nevertheless, in order that the applications written in Java benefit from the computing grids, they must be decomposable into pieces that can be migrated from one node to another. Java's threads cannot accomplish this task, so a more convenient solution must be found.

1.1 Contribution

We have implemented Java facilities that permit migration of the Java threads. This permits the application programmer to use grid systems in order to write simple, highly scalable and efficient programs. In this paper, we describe an approach called *MigThread* that permits the migration and execution of Java threads on different nodes of the grid and corresponds to the following specifications: sharing of variables among the threads, no modification of the JVM (Java Virtual Machine), Java's syntax and the program interface to which the programmer is accustomed. Finally, no deployment of additional application to the nodes other than the JVM of course.

In order to satisfy these specifications, we used the following mechanisms: adding of *advices* to certain portions of the user code thanks to the Aspect-Oriented Programming, Java's introspection mechanism which allows the program to discover the variables and the methods of a class from its name, Remote Method Invocation (RMI), grid file system's facilities which permit to access a Java application from any node and distant process execution by SSH.

We were forced to go back to the notion of process in order to realize efficient migration and then to return to the notion of threads (starting from these processes) in order to permit sharing of variables.

Our approach does not include the following functionalities: checkpointing, serialization, scheduling, fault tolerance and synchronization of Java threads. However, these functionalities will be included in upcoming releases.

1.2 The Plan

The rest of this paper is organized in the following manner: section 2 gives a quick overview of the approaches used to distributes threads in the grid system whereas section 3 presents Aspect-Oriented Programming as well as AspectWerkz. Section 4 presents our approach and its mechanism. Finally, the obtained results as well as the perspectives are discussed in section 5 and the conclusion is given in section 6.

2. STATE OF ART

Much work has been concentrated on migrating threads in grid systems. These are based on a middleware layer which provides grid services; examples of such a middleware layer are GTPE[6] (Grid Thread Programming Environment) of the University of Melbourne which is based on GridBus Broker[2] and PROACTIVE of INRIA[5]. Other works are based on distributing Java's virtual machine like the cJVM[3] approach. It is important to note that the middleware approach is based on modifying the syntax of the Java language whereas the second approach is based on modifying the virtual machine.

3. THE ASPECT ORIENTED PROGRAMMING

The computer programs are generally composed of two important perspectives: a functional perspective which permits the program to realize the things that it was devised for and the non-functional perspective which permits the program to be integrated into its environment or to respect certain specifications. Examples of non-functional perspectives are authentication or the event logging procedures. However, such perspectives are often blended in with the functional code, what makes their evolution and separation a very difficult task. These scattered pieces of code of the program are called *cross cutting concern*.

The aspect programming[4] permits to separate this non-functional code from the functional one when writing the program. There exists different approaches to achieve this task but we will quickly present only *AspectWerkz*[7].

3.1 AspectWerkz

An aspect contains a code called *advice* that will be executed at *pointcuts*. AspectWerkz proposes to define the pointcuts in an XML file and the advices in Java. The pointcuts are described by a logical language which permits to specify particular points in the code where the advices can be applied. These points can be, for example, method calls or reading and writing of variables.

For instance, the following files permit to display "the foo method will be called" every time the method foo of the application is called.

- the file that contains the advice

```
public class aspect_foo
{
  public void call_foo_method()
  {
    System.out.println("the foo
        method will be called");
  }
}
```

- and the file that describes the pointcut:

```
<aspectwerkz>
  <aspect class="aspect_foo">
    <pointcut
      name="foo_call"
      expression="execution(* *.foo
          (..))"
    />
    <advice
      name="before_foo_call"
      type="before"
      bind-to="call_foo_method"
    />
  </aspect>
</aspectwerkz>
```

AspectWerkz possesses a static and a dynamic weaver. In our approach, we used only the static one.

3.2 The Aspect Programming of our Approach

We are going to use the aspect programming in order to apply an advice before reading or writting the variables of the *run()* method. Thus, in our approach, we will be using a *get* or *set* pointcut types. Unlike AspectJ, AspectWerkz's API will allow us to discover the read or written variable. Then the advices will permit to update this variable before carrying out any operation related to it.

4. OUR APPROACH

Our approach is made up of two parts: migration of threads towards a distant node and the sharing of variables among the migrated threads.

4.1 Migration Towards a Distant Node

To achieve this task, a *MigProcess* class is created. Its main purpose is to realize a migration towards a distant node but does not allow any communication among the migrated tasks. Hence, we can consider these migrated tasks as processes. This class retrieves the API of JAVA threads but on the other hand modifies the signature of the *join()* method and makes it *final* so that neither it can be overridden nor it can be overloaded by the classes that inherit it. When the *start()* method is called, the task peels off in order to be executed on another node. When the *join()* method is called, the father process remains blocked until the execution of the distant process terminates. This method is used in order to synchronize the father with the child processes. We can notice in this class the existence of a *main()* method which allows it to become an executable class. This method will permit to resume the migrated process at the distant node.

In order to realize a migration, the following steps must be taken: separation of the migrating code from the principle application, migration towards the distant node, distant execution and synchronization between father and child processes.

In this paper, we will not delve into the details of these different steps. On the contrary, we consider that the threads are already distributed among the nodes and see how the variables can be shared among the threads.

4.2 The Principle of Variable Sharing Among Local Objects and Migrating Threads

4.2.1 *Highlights of the Proposition*

During the application's compiling phase, the programmer must call the AspectWerkz weaver which will weave the predefined aspects to his code. These aspects will play the role of capturing all read and write operations of the variables of the migrating thread's *run()* method. For each read and write operation, a call to a variable's address server is performed in order to know the address of the machine (thread) that holds the most recent value of the concerned variable. If this updated value is not found in the local migrating thread, it will be obtained by an RMI call to the thread that has the most recent value.

Hence the table of this variable's address server is updated. It is important to note the presence of a distributed thread context: not all updated values of the variables are found in the same migrating thread. Each migrating thread possesses a portion of the application's context and works locally as long as this portion is neither read nor written by another node of the grid. These mechanisms are included in the *MigThread* class which is the basic class of our approach and all migrating threads must inherit it.

4.2.2 *Mechanism*

As explained above, each migrating thread must inherit the MigThread class. Thus, each will behave like a Java thread but with a context distributed among several nodes of the grid. The MigThread class extends JVM's *UnicastRemoteObject* class which makes it an RMI component. Likewise, every instance of the MigThread class is associated with an instance of the MigProcess class. Finally, this class uses the *start()* and *join()* methods of the standard thread API.

When the *start()* method of the migrating thread is called, the *start()* method of the ancestor class MigThread is executed.

This method performs the following operations:

1. if it is the first migrating thread of the application, an RMI register is created at the local node of the application and an empty static hashtable is reserved (belonging to the MigThread class),

2. the MigProcess class associated with this instance of the class is started.

The migrating thread migrates to the distant node and the *main()* method of the MigProcess is called. This method executes the *pre_run()* method which creates an RMI register at the distant node and then executes the *run()* method of the migrating thread.

4.2.3 *The Address Server*

The server of the variable's address is encapsulated within the MigThread class. It consists of a static hashtable and an RMI register using the *CommWithBase* interface. The RMI register and the hashtable are created during the first execution of the distant thread. They remain on the machine that executes the Java application and the hashtable's *static* modifier permits it to remain unmodified even if all the migrating threads terminate their execution.

The *CommWithBase* interface defines the following method:

```
public String getAddress(String var,
    String host, String name);
```

A call to this method is included in the advices which are bind to the code of the application in order to know the migrating thread that has the most recent value of the variable *var*.

The *host* and *name* parameters are used by the server in order to update the hashtable. In fact, we consider that a migrating thread that asks for the address of a variable, will obtain the most updated value. For each call to the *getAddress()* method, the table is updated.

This table is initially empty and as, during the migration, the context of the migrating threads is serialized, we consider that the first migrating thread that asks for a variable is also the one that possesses the most updated value.

4.3 The Migrating Threads

Every migrating thread carries with it an RMI server included in the superclass MigThread. This RMI server permits a thread to provide another thread a variable that will be updated in its context. Likewise, each thread is able to ask another thread for the value of a variable.

Therefore, the architecture for the exchange of variables is not centralized but is totally "point to point". On the other hand, the server of the variable can be a real bottleneck which imposes a client/server architecture to retrieve the addresses of the variables.

At the compilation time, the *run()* method of the migrating threads is modified by the AspectWerkz's preprocessor which captures all the read and write operations of the variables. When a read or write operation is called, the methods of *ExtendedInformation* interface are called by the advices.

4.3.1 *The ExtendedInformation Interface*

This interface is composed of the following two methods:

```
public void getVar(String variable);
public void setVar(String variable);
```

It permits the advices to update the variable that looks to be read or written.

The *getVar()* method, called during each read operation, sends a query to the address server in order to retrieve the address of the variable *variable*. If this variable is local, it stops at this point. If the variable is not local, its type is determined. A second query is sent to the thread possessing the updated value and then an update is performed locally. Thus the thread can perform the read operation.

The *setVar()* method, called during each write operation, sends only one query to the address server in order that this latter updates its address table. Thus the thread can perform the write operation.

4.3.2 The RMI Server of the Migrating Threads

This server works with the *Information* interface. This interface permits the threads to ask for the variables and possesses the methods of the following type:

```
// A method to retrieve an object called
     variable
public Object getObject(String variable)
     throws RemoteException,
    NoSuchFieldException,
    IllegalAccessException;

// A method to retrieve an integer
    called variable
public int getInt(String variable)
    throws RemoteException,
    NoSuchFieldException,
    IllegalAccessException

// The same for every type
```

When one of these methods is called, the asked variable is searched by the introspection mechanism using its name. When it is found, its name is sent to the thread that asked for it or an exception of *NoSuchFieldException* type is raised.

4.3.3 The Used Aspects

4.3.3.1 The advices.

The advices that we used are of two types: *before_reading* and *before_writing*. They recognize the name of the variable which will be read or written and then they call the *getVar()* or *setVar()* methods according to the case.

4.3.3.2 The pointcuts.

The pointcuts are dynamically generated before the weaving phase of the user. In fact, AspectWerkz can not determine the class from its superclass. Hence, we must find all the classes that extended the MigThread class and then for each of these classes the following type of pointcut must be created:

```
<pointcut name="write_AMigratingThread"
    expression="set(* *.AMigratingThread
    .*) AND withincode(* *.
    AMigratingThread.run())" />
<pointcut name="read_AMigratingThread"
    expression="get(* *.AMigratingThread
    .*) AND withincode(* *.
    AMigratingThread.run())" />
<!-- and so forth for all the classes
    that extend MigThread -->
```

for the *AMigratingThread* class.

The advices are attached to these pointcuts thanks to the following definitions:

```
<advice name="before\_reading" type="
    before" bind-to="
    reading_AMigratingThread" />
<advice name="before\_writing"  type="
    before" bind-to="
    writing_AMigratingThread" />
<!-- and so forth for all defined
    pointcuts -->
```

If the inheritance manager of the AspectWerkz's pointcut language would have been integrated, only two pointcuts and two pointcuts-advice associations will have to be necessary.

4.4 Synthesis

Below is the flowchart of the read and write operations of a variable.

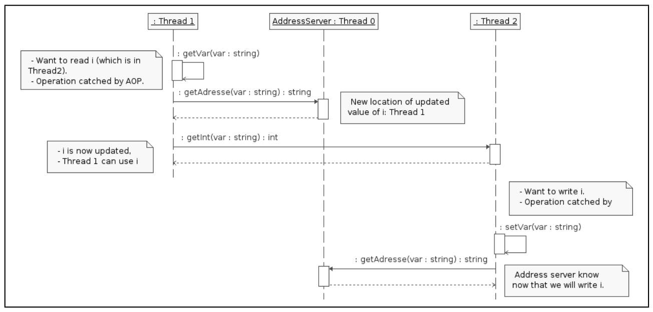
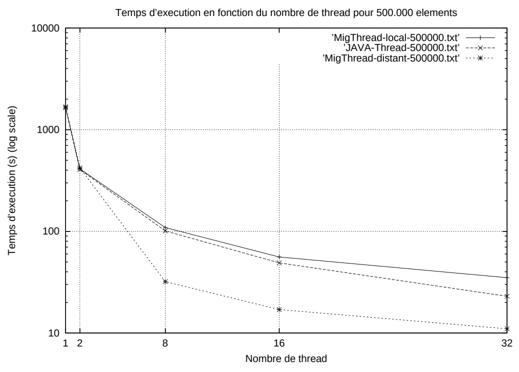

: Thread 1
AddressServer : Thread 0
: Thread 2
- Want to read i (which is in Thread2).
- Operation catched by AOP.
: getVar(var : string)
: getAdresse(var : string) : string
New location of updated value of i: Thread 1
: getInt(var : string) : int
- i is now updated,
- Thread 1 can use i
- Want to write i.
- Operation catched by
: setVar(var : string)
: getAdresse(var : string) : string
Address server know now that we will write i.

Figure 1: Flowchart of the read and write operations of a variable

5. RESULTS AND PERFORMANCES

We tested our approach by using a simple bubble-sorting algorithm on a grid composed of 4 machines each possessing dual processors. This sorting method is not the optimum one and this is acceptable because our main objective was to prove the validity of our approach. It executes on an array whose elements are inversed and functions in its worst case at $O(n^2)$.

We present the results in the form of a curve graph where the horizontal axis represents the number of used threads to sort an array of 500000 elements and the vertical axis represents the execution time to sort this array. On the diagram, we will find the curves for sorting the array using Java threads, local MigThreads and distant MigThreads.



Here we can notice that the three curves overlap for a number of threads of 1 and 2. This is normal because the machines on which these tests were performed possess a double processor. We can equally notice the absence of any significant overhead by the MigThread approach using 1 or 2 processes. For a thread number of 8, the curves start to diverge: the portion of the curves of the Java thread and the local MigThread approaches have a similar variation whereas the distant MigThread approach gets benefit from distribution. The results of the distant MigThread approach are completely in conformity with the theoretical results of parallelism with 8 threads executing on 8 different processes. For 16 threads, both the distant MigThread and Java thread approaches obtain the same gain in performance. This is due to the progressive drop off of the complexity because the original array is partitioned into several sub-arrays. The local MigThread approach begins to become less efficient than the Java thread one. In fact, it is clear that these two approaches are both theoretically efficient but the overhead which is due to the network connection of the local loop starts to be a dominant factor in performance degradation. Finally, with 32 threads, local MigThread approach is less efficient than the Java thread approach. The distant MigThread approach varies in performance as the Java thread approach because the use of 8 processors masks away the problem of overhead due to the migration of the MigThreads.

We performed another test with 5000 elements. The results weren't good enough because of the migration overhead.

6. CONCLUSION

We have succeeded in distributing an application throughout a heterogeneous grid by respecting our constrains. Indeed, by using Java, the execution of applications in a heterogeneous grid becomes plausible. On the other hand, neither Java's parallel API nor its virtual machine are modified and the application can be distributed among several nodes of the grid without deploying any application other than the JVM.

The obtained performances, by not taking into account the problem related to the migration time of the MigThreads through the network, are almost the same compared to the ones that could have been obtained if the machine on which threads were executed possess the same number of processors as the grid to which the MigThreads are migrated.

Nevertheless, the absence of synchronization makes coding the program a difficult task. Hence it is important that this synchronization be done in an explicit manner for example by means of a semaphore server. That is the reason why we are actually using AOP approach in order to synchronize the threads by using classical Java threads for synchronization.

Hence we will have a complete solution of distributing Java applications in a heterogeneous grids.

7. REFERENCES

[1] V. Berstis. Fundamentals of grid computing. In *Redbooks*, 2002.

[2] R. Buyya and S. Venugopal. The gridbus toolkit for service oriented grid and utility computing: An overview and status report. In *First IEEE International Workshop on Grid Economics and Business Models (GECON 2004, April 23, 2004, Seoul, Korea)*, 2004.

[3] IBM. A scalable single system image vm for java on a cluster, 2000. http://www.haifa.il.ibm.com/projects/systems/cjvm/overview.html.

[4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[5] ProActive. Inria, 1999. http://www-sop.inria.fr/oasis/ProActive.

[6] H. Soh, S. Haque, W. Liao, K. Nadiminti, and R. Buyya. Gtpe (grid thread programming environment). In *13th International Conference on Advanced Computing and Communications (ADCOM 2005)*, 2005.

[7] WebSite. Aspectwerkz - plain java aop - overview, 2005. http://aspectwerkz.codehaus.org/.

Classifying and documenting aspect interactions

Frans Sanen, Eddy Truyen, Wouter Joosen
Distrinet Research Group
Department of Computer Science
K.U.Leuven
Celestynenlaan 200A,
B-3001 Heverlee - Belgium
Tel. (+32) (0) 16 32 76 02
frans.sanen@cs.kuleuven.be

Andrew Jackson, Andronikos Nedos, Siobhán Clarke
Distributed Systems Group,
Trinity College Dublin
Dublin 2 - Ireland,
Tel (+353) (0) 16081543
anjackso@cs.tcd.ie

Neil Loughran, Awais Rashid
Computing Department
Infolab 21,
Lancaster University
(+44) (0) 1524 510503
loughran@comp.lancs.ac.uk

ABSTRACT

In this position paper, we propose an approach for classifying and documenting aspect interactions. Making aspect interactions explicit will result in an important form of knowledge that can be shared or used over the course of system evolution.

Keywords
Aspect-Oriented Software Development, Interaction

1. INTRODUCTION

Aspects enable isolating and modularizing crosscutting concerns. In the literature, aspects that are orthogonal to one another have often been used to illustrate the benefits of Aspect-Oriented Software Development (AOSD) [8]. However, are all aspects truly orthogonal in reality? It is a simple question with an equally simple answer. Beyond some trivial lower-level aspects, there will be interdependencies between the more interesting aspects, resulting in aspect interactions.

Our motivation is based on the relevance of the problem. Aspect interactions are both technology and domain independent. The domain independence can be motivated by the example interactions that are known for numerous application domains, such as telecommunications, email, thermo control, policy-based, multimedia, middleware and other systems [2, 3, 4, 6, 11, 13, 15, 16]. Due to this technology and domain independence, solutions for aspect interactions will be generally applicable. As a consequence, aspect interactions can directly endanger the integrity, availability and reliability of their enclosing infrastructure. Future expectations aren't good either, because, in our opinion, the amount of aspect interactions will keep increasing. Since software development is subject to reduced time-to-market cycles, an aspect-oriented system will need to cope with evolution beyond delivery and unanticipated changes during the maintenance phase. Additionally, more and more aspects will be developed by third parties, independently of the existing platform. Garlan et al. [10] argument that software elements developed independently of each other are the main reason causing interactions because they make implicit or explicit assumptions that are only valid if the software element is operating in isolation. Applying their idea to AOSD gives us aspects, which make implicit or explicit assumptions that are only valid if the aspect is the only aspect composed with the base system, as the main reason causing interactions. Unfortunately, current systems cannot guarantee correctness when aspects are not tightly integrated with the base system.

There exists a large set of publications on interactions, especially in the telecommunications domain [4, 13]. However, interactions also arise in complex systems such as middleware and product lines [2, 15]. In the aforementioned systems, there is a common agreement and wide acknowledgement of the necessity of handling interactions. First of all, it is important to realize that not all interactions are harmful. Both positive and negative aspect interactions exist. Therefore, we propose a classification of aspect interactions in which this distinction is reflected. The main benefit from this classification is that by having different types of interactions common patterns of interaction and their response types may arise. Once we have a substantial list of interactions, analysis of the examples for a specific type of interactions may reveal these common patterns. Secondly, to the best of our knowledge, a means for explicitly documenting aspect interactions is lacking. Therefore, we propose an approach for documenting aspect interactions, and, hence, making them explicit. This will result in an important form of knowledge that can be shared or used over the course of system evolution. Aspect interactions then can be addressed if they are negative or kept as part of system documentation if they are positive.

Obviously, this form of knowledge about interactions can be useful at different stages of the software life cycle. For instance, a middleware platform can be developed to be able to interpret implementation-oriented interactions automatically whereas an architectural design tool can exploit interactions exposed by requirements-level composition and analysis. Especially the latter case of an architectural blueprint capturing design know-how that resolves certain axiomatic interactions – interactions that are invariant – is the long term result we aim for.

The rest of this paper is structured as follows. In Section 2 we present the example application we use to give examples of aspect interactions. We propose a classification of aspect interactions in Section 3. Our proposed approach to document aspect interactions explicitly is elaborated and illustrated in Section 4. Section 5 discusses our work. Some related work is covered in Section 6. Finally, this position paper is concluded in Section 7.

2. EXAMPLE APPLICATION

The example application we use to give examples of aspect interactions in our classification is an online auction system that allows people to negotiate over the buying and selling of goods in the form of English-style auctions. The system only allows for enrolled users to log into the system to start a session in which they can buy, sell or browse through auctions available. When one wants to follow an auction, one must first join the auction. Once one has joined an auction, one may make a bid. One only can bid before an auction closes. After it is closed, the system calculates the highest bid and checks if it meets the reserve price issued by the seller. Finally, the relevant parties are informed of the outcome of the auction.

In this application context, we limit our scope to three concerns[1]: persistence, security and context-awareness. Based on discussions with experts from all three domains, we came up with an initial breakdown for these concerns into sub-concerns in order to make the interaction documentations as concrete as possible. An important criterion for these breakdowns was the orthogonality of the sub-concerns to structure further analyses more easily. Security encloses authentication, authorization, non-repudiation, integrity, confidentiality and auditing. Persistence is broken down into state encoding, state change detection, state-access, transactions, caching and logging. Our context-awareness concern breakdown distinguishes between context monitoring, inference and action. We start from the assumption that all these sub-concerns can be realized through aspects.

3. CLASSIFICATION

Aspects can interact in multiple ways. In what follows, we distinguish between four different types of aspect interactions: mutual exclusion, dependency, reinforcement and conflict.

- *Mutual exclusion* encapsulates the interaction of mutual exclusiveness. For example, if there are two aspects that implement similar policies, or algorithms, the situation can arise where only one such aspect must be used. No mediation is possible because the aspects are not complementary: only one of them can be used, the other cannot. With respect to the auction example this might simply be the choice of a scheme for caching particular records in order to increase performance. If two mutually exclusive schemes were weaved, results will be unpredictable or undesirable. Another example might be a specific product configuration using persistence that included both relational and object mapping schemes, which could result in data duplication or wrong encoding in the target database.
- *Dependency* covers the situation where one aspect explicitly needs another aspect and hence depends on it. A dependency does not result in a problem or erroneous situation as long as the aspect on which another one depends is ensured to be present and not changed. To illustrate this situation, two simple dependencies are the following: authorization depends on authentication and context actuation depends on context monitoring. Without the latter, the former cannot perform correctly.
- *Reinforcement* arises when an aspect influences the correct working of another aspect positively and hence reinforces it. There can be no doubt that this type of interaction is a positive one. When an aspect reinforces another aspect, extended functionalities become possible and extra support is offered. As an example, the monitoring of an auction user's location (context-monitoring) allows for an extended authorization policy. For instance, it is possible to realize that only users within the country in which an auction is taking place are allowed to make a bid.
- *Conflict* captures the situation of semantical interference: one aspect that works correct in isolation does not work correctly anymore when it is composed with other aspects. Hence, an aspect influences the correct working of another aspect negatively. Typically, a conflict can be solved by mediation because the aspects –in a sense- are complementary. To illustrate this situation, non-repudiation and state encoding are in conflict because one does not want to persist repudiated data.

4. DOCUMENTING ASPECT INTERACTIONS

4.1 Template

We propose a template with clearly defined semantics for describing interactions unambiguously. This template consists of a record structure consisting of a number of attribute-value pairs and is explained below.

We have found that, when documenting aspect interactions, it is important to explicitly state the relevant conditions that hold when an interaction occurs. Obviously, these conditions evaluate over the context of the interaction. In our template, these conditions are called explaining predicates.

An aspect interaction description consists of the following attributes:

- *Name*, containing the name of the interaction.
- *Aspects involved,* containing the aspects involved in the interaction. Involved aspects typically are at the level of sub-concerns.
- *Type*, containing the type of interaction. We distinguish between four different types of interaction: mutual exclusion, dependency, reinforcement and conflict. These were discussed in Section 3.
- *Example,* containing an informal example of the interaction.
- *Explaining predicates,* containing the predicates that can be used to explain the interaction. Predicates typically express conditions that evaluate over a number of parameters representing the relevant context information of the interaction. Each explaining predicate is again described through attribute-value pairs.
    - *Name,* containing the name of the explaining predicate.
    - *Aspect,* containing the aspect the explaining predicate is referring to.
    - *Definition,* containing the definition indicating in which states and/or under which conditions the explaining predicate holds.
    - *Parameters,* containing the parameters that are relevant for evaluating the explaining predicate.
- *Description,* containing an informal description of the interaction in terms of the explaining predicates.
- *Time of response,* containing the appropriate time in the software lifecycle to respond to the interaction. Software lifecycle stages we consider to be relevant are the following: requirements, architecture, design, implementation, middleware, deployment.
- *Type of response,* containing an informal description of the mechanism, action and/or structure that uses the knowledge about the interaction to solve it (in case of a conflict or mutual exclusion) or instantiate it (in case of a dependency or reinforcement).

[1] These three concerns were already studied as a part of [21].

4.2 Illustrated

We illustrate the applicability of our template in documenting aspect interactions with an example taken from the online auction system as described in Section 2. Users can participate in an auction through their handheld device. As mentioned in the application description, each buyer and seller will have an associated session. In order to ensure integrity of each bid, a symmetric session key is part of each user session. For obvious reasons, this session key only has a limited lifetime reducing the chance that the integrity of a bid can be broken. Each time the session key expires, a new one has to be generated, which is very computationally intensive. When the handheld device power is low, a clear aspect interaction arises. The integrity security sub-concern is in conflict with the device power context, in which one wants to avoid intensive computations. The example described within our template is shown in Table 1.

Table 1. Example aspect interaction description.

Name	Extend session key lifetime if device power is low
Aspects involved	Integrity, Context monitoring
Type	Conflict
Example	There shouldn't be generated a new symmetric session key when the power of an auction user his/her handheld device is low.
Explaining predicates	*Name*: SessionKeyAboutToExpire *Aspect*: Security/Integrity *Parameters*: Joinpoint *Definition*: SessionKeyAboutToExpire(Joinpoint jp) holds when the current session key is about to expire at join point jp. *Name*: DevicePowerLow *Aspect*: Context-awareness/Context monitoring *Parameters*: Joinpoint *Definition*: DevicePowerLow(Joinpoint jp) holds when the device power is low at join point jp.
Description	∀ Joinpoint jp: DevicePowerLow(jp) requires !SessionKeyAboutToExpire(jp)
Time of response	Architecture, design, middleware
Type of response	When the power of one's handheld device is low, the lifetime of the current symmetric session key should be automatically extended instead of creating a new key.

5. ONGOING AND FUTURE WORK

This approach is being taken within a larger investigation of AOSD applications[2] carried out in the context of AOSD-Europe [1]. We have begun to investigate aspect interactions within the context of the online auction system and the scope of three extra-functional concerns: persistence, security and context-awareness. Currently, we are manually documenting a list of aspect interactions between these aspects in the example auction application. Our previous example illustrating the template is one of these.

Based on this first experiment, we will evaluate the proposed documenting approach and extend it where needed. Extra attribute-value pairs may be needed. For instance, we do not consider instances of aspects yet. For now, we assumed there is only one global instance of each aspect. Moreover, we believe that other (sub)types of interaction may exist and will need to be incorporated. Our end goal is an exhaustive and non-overlapping classification of aspect interactions.

We realize that, because of the limited scope of our experiment, this merely is a first step towards our end goals. Nevertheless, we are persuaded of the relevance of finding a good approach to document aspect interactions explicitly and unambiguously. By means of documenting aspect interactions and, hence, make them more explicit, it will be possible to share this crucial development and execution knowledge. This results in an important form of help for other members of the team or over the course of evolution, when even a different team may be in place. Classifying and documenting aspect interactions enables us to address them in case of a conflict or a mutual exclusion or have them as part of the system documentation in case of a reinforcement or a dependency.

6. RELATED WORK

Interactions are a widely known problem, especially in the telecommunications domain. The relevance of interactions in the broader context of a middleware platform and its proliferated common services has been discussed in [2, 15]. A number of solutions have been proposed to deal with conflicting situations, such as for example in [17], where the authors propose the notion of a compositional intersection to enable the identification of suitable sets of concerns that can then be used in an early trade-off analysis. Some first results based on an analysis of a set of aspects are discussed by Douence et al. [7] and Rinard et al. [19].

We are convinced that our approach for classifying and explicitly documenting interactions is complementary to this body of existing work. To the best of our knowledge, there only exist a few approaches that are similar to our proposed approach and focus on describing the interactions themselves. In [18], Pawlak et al. propose CompAr, a language that allows programmers to abstractly define an execution domain, advice codes and their often implicit execution constraints. Their language enables the automatic detection and solving of aspect-composition issues (aspect interactions) of around advices. Major contribution of the work in [18] is the high level of abstraction the language offers to specify very generic aspect definitions. Batory et al. [14] propose

[2] More information about this larger investigation of AOSD applications can be found on the AOSD-Europe web portal (http://www.aosd-europe.net) within the applications research lab area.

an algebraic theory for modeling interactions in feature-oriented designs. In their theory, feature interactions are modeled as derivatives. The approach taken in [14] is very similar to the Distributed Feature Composition of Zave et al. [12].

7. CONCLUSION

In this short paper we have demonstrated our approach for classifying and documenting aspect interactions. While the research presented here is still in the preliminary phases, we believe that the direction taken offers a fresh perspective that is complementary to other research already undertaken in the field of AOSD. Indeed, we already envisage that the work could form the beginnings of what will be a catalogue of *interaction patterns* in a similar vein to that of design patterns [9]. A future document [20] will express this in more detail.

So far we have only discussed *production aspects* (i.e. aspects that are included in the final application). Of course, in a typical development scenario one could imagine many different kinds of *development aspects* (i.e. aspects that are used in the activities of testing, profiling, tracing, where the code is not included in the final release) causing interactions with our core concerns. Additionally different *versions* of concerns, which may exist in the software product line [5] context, where concerns are customized to differing requirements, can also further exacerbate the problem of interactions.

Interactions across many aspects add to our conviction that there is a need for the documentation, detection, modularization and, ultimately, the resolution of aspect interactions.

8. ACKNOWLEDGMENTS

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

9. REFERENCES

[1] AOSD-Europe. European Network of Excellence on Aspect-Oriented Software Development. European Commission grant IST-2-004349.

[2] L. Blair, G. Blair, and J. Pang. Feature interaction outside a telecom domain. *Workshop on Feature Interaction in Composed Systems*, 2001.

[3] L. Blair, and J. Pang. Feature interactions - Life beyond traditional telephony. *FIW 2000*, p. 83-93.

[4] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks: The International Journal of Computer and Telecommunications Networking archive*, Vol. 41 , Issue 1, p. 115-141, January 2003.

[5] P. Clements and L. Northrop, "Software product lines - Practices and patterns," Addison Wesley, 2002.

[6] J. A. Diaz Pace, F. Trilnik, and M. R. Campo. How to handle interacting concerns? *Workshop on Advanced for Separation of Concerns in OO Systems*, OOPSLA 2000, Minneapolis, USA.

[7] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects, *International Conference on Aspect-Oriented Software Development (AOSD04)*, 2004.

[8] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit (Eds.). Aspect-oriented software development. Addison-Wesley, 2005.

[9] E. Gamma, et al. Design patterns: Elements of reusable object-oriented software. Addison-Wesley, Reading, 1995.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995

[11] R. J. Hall. Feature interactions in electronic mail. *Proceedings of the 6th International Workshop on Feature Interactions in Telecommunications and Software Systems*, IOS Press, 2000.

[12] M. Jackson, and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering* XXIV(10):831-847, October 1998.

[13] D. O. Keck, and P. J. Kuehn. The teature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, Vol. 24, No. 10, October 1998.

[14] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented systems. *International Conference on Feature Interactions (ICFI)*, June 2005.

[15] X. Liu, G. Huang, W. Zhang, and H. Mei. Feature interaction problems in middleware services. *International Conference on Feature Interactions (ICFI)*, June 2005.

[16] E. Lupu, and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, Vol. 25 , Issue 6, p. 852-869, November 1999.

[17] A. Moreira, A. Rashid, J. Araujo. Multi-dimensional separation of concerns in requirements engineering. *International Conference on Requirements Engineering*, Paris, France, 2005.

[18] R. Pawlak, L. Duchien, and L. Seinturier. CompAr: Ensuring safe around advice composition. *7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS05)*, Athens, Greece, June 2005.

[19] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for AO programs. *Proceedings of the Twelfth International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, November 2004.

[20] F. Sanen, E. Truyen, W. Joosen, N. Loughran, A. Rashid, A. Jackson, A. Nedos, S. Clarke. Study on interaction issues. AOSD-Europe Deliverable 44. March 2006.

[21] F. Sanen, E. Truyen, B. De Win, W. Joosen, N. Boucke, T.Holvoet, N. Loughran, A. Rashid, R. Chitchyan, N. Leidenfrost, J. Fabry, N. Cacho, A. Garcia, A. Jackson, N. Hatton, J. Munnelly, S. Fritsch, S. Clarke, M. Amor, L. Fuentes, L. Fuentes, and C. Canal. A Domain Analysis Of Key Concerns – Known And New Candidates. AOSD-Europe Deliverable 45. March 2006.

Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces

Dean Wampler
Aspect Research Associates *and*
New Aspects of Software
dean@aspectprogramming.com

ABSTRACT

Recent trends in Aspect-oriented Design (AOD) have emphasized interface-based modularity constructs that support noninvasive advising of components by aspects in a robust and flexible way. We show how the AspectJ-based tool *Contract4J* supports *Design by Contract* in Java using two different forms of a design pattern-like protocol, one based on Java 5 annotations and the other based on a JavaBeans-like method naming convention. Neither form resembles conventional Java-style interfaces, yet each promotes the same goals of abstraction and minimal coupling between the cross-cutting concern of contract enforcement and the components affected by the contracts. Unlike traditional implementations of design patterns, which tend to be *ad hoc* and require manual enforcement by the developer, the Contract4J protocol offers at least partial support for compile time enforcement of proper usage. This example suggests that the concept of an interface in AOD should more fully encompass usage protocols and conversely, aspects give us ways to enforce proper usage programmatically.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming. Aspect-oriented Programming

General Terms

Design, Theory.

Keywords

Aspect-oriented software development, object-oriented software development, design, AspectJ, Java, Contract4J, Design by Contract.

1. INTRODUCTION

Much of the current research work on Aspect-Oriented Design (AOD) is focused on understanding the nature of component and aspect interfaces with the goal of improving modularity, maintainability, *etc.,* just as pure object-oriented systems emphasize interface-based design [1-2]. Because aspects are cross-cutting, they impose new challenges for design approaches that don't exist for objects, but they also offer new capabilities that can be exploited.

Griswold, *et al.* [2] have demonstrated one approach to designing interfaces, called *Crosscutting Programming Interfaces* (XPIs), which resemble conventional object-oriented interfaces with additions such as pointcut declarations (PCDs) that hide component join point details behind abstractions. The interfaces are implemented by components that wish to expose state or behavior to interested "clients", *e.g.,* aspects. The aspects then use the interfaces' PCDs only, rather than specifying PCDs that rely on specific details of the components. This approach effectively minimizes coupling between the components and the aspects, while still supporting nontrivial interactions. For our purposes, it is interesting to note that the XPI PCDs also impose contract-like constraints on what the components, which are the interface implementers, and the aspects are allowed to do. For example, a component may be required to name all state-changing methods with a prefix like "set" or "do", while an aspect may be required to make no state changes to the component.

Contract4J [3] is an open-source tool that supports *Design by Contract* (DbC) [4] in Java using AspectJ for the implementation. It demonstrates an alternative approach to aspect-aware interface design with two syntax forms supporting a common protocol that will be suitable for many situations. Instead of specifying methods to implement, like a traditional Java interface, one form uses Java 5 annotations and the other uses a method naming convention. Both approaches are essentially a design pattern [5] that must be followed by components that wish to expose key information, in this case a usage contract, and by clients interested in the usage contract, which they can "read" from the components if they understand the protocol, without having to know other details about the components.

2. DESIGN BY CONTRACT AND CONTRACT4J

Briefly, Design by Contract (DbC) [4] is a way of describing the contract of a component interface in a programmatically-testable way. *Preconditions* specify what inputs the client must provide in order for the component to successfully perform its work. *Postconditions* are guarantees made by the component on the results of that work, assuming the preconditions are satisfied. Finally, *invariant conditions* can be specified that hold before and after any publicly-visible component operation. The test conditions are written as executable code fragments to support runtime evaluation of the tests. When the tests fail, program execution stops immediately, forcing the developer to fix the problem immediately. Hence, DbC is a development tool for finding and fixing logic errors. It complements Test-Driven Development [6].

Contract4J supports DbC in Java as follows. The component developer annotates classes, fields, and methods with annotations that define the condition tests as Java boolean expressions. Aspects advice the join points of these annotations and evaluate the expressions at runtime to ensure that they pass. If not, program execution is halted.

Here is an example using a simplistic `BankAccount` interface.

```
@Contract
interface BankAccount {
  @Post("$return >= 0");
  float getBalance();

  @Pre("amount >= 0")
  @Post("$this.balance ==
         $old($this.balance)+amount
         && $return == $this.balance")
  float deposit(float amount);

  @Pre("amount >= 0 &&
        $this.balance - amount >= 0")
  @Post("$this.balance ==
         $old($this.balance)-amount
         && $return == $this.balance")
  float withdraw(float amount);
  …
}
```

Figure 1: BankAccount with Contract Details

The Contract4J annotations are shown in **bold.** The `@Contract` annotation signals that this class has a DbC specification defined. The other annotations are ignored unless this annotation is present. Contract4J also includes aspects that will generate compile-time warnings if the other annotations are present without the `@Contract` annotation, an example of partial programmatic enforcement of proper usage[1]. The `@Pre` annotation indicates a precondition test, which is evaluated before the join point executes. The `withdraw` method has a requirement that the input `amount` must be greater than or equal to zero and the `amount` must be less than or equal to the balance, so that no overdrafts occur. The expression `$this.balance` refers to an instance field that is implied by the JavaBean's accessor method `getBalance` defined in the interface. The `@Post` annotation indicates a postcondition test, which is evaluated after the join point executes. The `deposit` or `withdraw` method must return the correct new balance (specified with the `$return` keyword) and the new balance must be equal to the "old" balance (captured with the `$old(..)` expression) plus or minus the input amount, respectively. Not shown is an example `@Invar` annotation for invariant specifications, which can be applied to fields and classes, as well as methods. The field and class invariants are tested before and after every non-private method executes, except for field accessor methods and constructors, where the invariants are evaluated after execution, to permit lazy evaluation, *etc.* Method invariants are tested before and after the method executes.

The test expressions are strings, since Java 5 annotations can't contain arbitrary objects. Aspects match on join points where the annotations are present. The corresponding advice evaluates the test strings as Java expressions using a runtime evaluator, the Jakarta Jexl interpreter [7].

So, including the annotations specifies the behavior of the component more fully, by explicitly stating the expected behavior and eliminating ambiguities about what would happen if, for example, `amount` were less than 0.

A separate, experimental implementation of Contract4J, called *ContractBeans,* supports an alternative way of defining test conditions, where the component implementer writes the tests as special methods that follow a JavaBeans-like [8] signature convention,. Here is the same `BankAccount` interface expressed using this approach[2].

```
abstract class BankAccount {
  abstract public float getBalance();

  boolean postGetBalance(float result) {
    return result >= 0;
  }

  abstract public
  float deposit(float amount);

  public boolean preDeposit(float amount) {
    return amount >= 0;
  }
  public boolean postDeposit(float result,
                             float amount){
    return result >= 0 &&
           result == getBalance();
  }

  abstract public
  float withdraw(float amount);

  public boolean preWithdraw(
      float amount) {
    return amount >= 0 &&
           getBalance() - amount >= 0;
  }
  public boolean postWithdraw(
                           float result,
                           float amount) {
    return result >= 0 &&
           result == getBalance();
  }
  …
}
```

Figure 2: "ContractBeans" Format

An abstract class is used, rather than an interface, so that the tests, which are now defined as instance methods, can be defined "inline". (An alternative would be to use an aspect with intertype declarations to supply default implementations of the test methods for the original interface.)

Following a JavaBeans-like convention, the postcondition test for the `withdraw` method is named `postWithdraw.` (Compare with the JavaBeans convention for defining a `getFoo` method for a `foo` instance field.) This method has the same argument list as `withdraw`, except for a special argument at the beginning of the list that holds the return value from `withdraw`. The `preWithdraw` method is similar, except that its argument list is

[1] The @Contract annotation is not strictly necessary, but it makes the Contract4J implementation more efficient and it makes the code more self-documenting. A future release may drop the requirement for it to be present.

[2] Actually, this implementation doesn't support the "old" keyword for comparing against a previous value, so the tests shown reflect this limitation.

identical to the `withdraw` argument list. All the test methods must return `boolean`, indicating pass or fail. Invariant methods follow similar conventions.

This version of Contract4J advices *all* the fields and methods in a class, then uses runtime reflection to discover and invoke the tests, when present. Hence, this implementation has significant runtime overhead and the method based approach has several drawbacks compared to the annotation approach, including greater verbosity when writing tests and a less obvious connection between the tests and the methods or fields they constrain. It is discussed here because it demonstrates the approach of using naming patterns to convey meta-information of interest to other concerns. However, as a practical tool, the annotation form of Contract4J is more usable.

More details on the design and usage of Contract4J, as well as a discussion of general issues encountered during its development, can be found in [9] and [3].

3. THE CONTRACT4J PROTOCOL AND ASPECT INTERFACES

The two approaches are different syntax forms for a design pattern, in the sense that they define a protocol for interaction between a component and clients of its contract, with minimal coupling between them. In this case, the protocol is used by aspects to find tests and execute them before and/or after a method invocation or field access. Compare this to the well-known *Observer* pattern where a state change in a subject triggers notifications to observers. In fact, many of the better-known patterns could be implemented, at least in part, using similar techniques. A simple implementation of *Observer* would be to annotate state-changing methods in potential subjects with an annotation, *e.g.,* @ChangesState. Aspects could advice these methods with `after` advice to retrieve the new state and react accordingly.

Using either form of the Contract4J protocol in an interface or class enhances the declaration with additional usage constraints that complete the specification of the component's contract, both the usage requirements that clients must satisfy, and the results the component guarantees to produce. DbC also promotes runtime evaluation of these constraints.

Consistent with the general goals of interface-based design, including the recent work on aspect-aware interfaces, the component remains agnostic about how the contract information is used. Of course, the two syntax forms were designed with Contract4J's needs in mind. However, other aspect-based tools could exploit these protocols for other purposes. Just as IDEs exploit JavaBeans conventions, the Contract4J protocol could be used, *e.g.,* by code analysis tools, automated test generation tools, and the code-completion features offered by most IDEs (*e.g.,* to warn when method arguments are clearly invalid). Hence, for a class of situations like DbC, a protocol like the two in Contract4J effectively provides an interface used by aspects for cross-cutting concerns. Specifically, if an aspect needs to locate all methods, constructors and fields meeting certain criteria that can be indicated through an annotation (or naming convention) and some action needs to be taken in the corresponding advice and that action can be expressed through the same syntax conventions, then this concern can be implemented in a similar way.

For example, the EJB 3 specification defines annotations that indicate how to persist the state of the annotated "Plain Old Java Objects" (POJOs) [12]. Actually, this example could be considered a case of tangling an explicit concern in the POJO; the annotations should ideally be more generic, perhaps conveying general lifetime information (*e.g.,* must live beyond the life of the process) and indicating what properties are required for uniquely specifying the state (*vs.* those that are transient properties). They should convey enough information that a persistence aspect could "infer" the correct the behavior (and similarly, other, unrelated aspects could infer what they need to know for their needs). Finding the right balance between explicitness and generality is clearly an art that needs to be developed.

Similar examples of using annotations to integrate with infrastructure services and containers include annotations that flag "sensitive" methods that trigger authentication processes before execution, annotations that provide "hints" to caching systems, and annotations that indicate potentially long-running activities that could be wrapped in a separate thread, *etc*.

4. DISCUSSION AND FUTURE WORK

The two Contract4J protocol forms demonstrate an idiomatic "interface"-based approach to aspect-component collaboration that are more like a design pattern than conventional interfaces, including the XPI form discussed by Griswold, *et al.* [2]. It is interesting that [2] calls for an explicit statement of the contract of behavior implied for both the components that implement the XPI and the aspects that use them to advise components. The unique characteristics of aspects make an explicit contract specification more important, because the risk of breaking existing behavior is greater, yet no programmatic enforcement mechanism currently exists. In contrast, Contract4J is specifically focused on contract definition *and* enforcement. Hence, Contract4J could be used to enforce the contract section of an XPI. Conversely, Contract4J could be extended to support the XPI-style of contract specification.

In general, Contract4J's protocol and the inclusion of contract details in XPI both suggest that good aspect-aware interfaces can go beyond the limitations of conventional interfaces, which provide a list of available methods and public state information, but usually offer no guidance on proper usage and never offer programmatic enforcement of that usage. Aspects offer the ability to monitor and enforce proper usage, which can then be specified as part of the interface.

With this support, design patterns can be programmatically enforceable and not just *ad hoc* collaboration conventions that require manual enforcement. The enforcement can be implemented as aspects and specified using a new kind of interface [10]. This would raise the level of abstraction in aspect design from focusing on points in a program's execution to designing nontrivial component collaborations at a higher level. Another example of "aspects as protocols" is the work of Jacobson and Ng on use cases as aspects [13]. Use cases are units of work involving collaborations of components. It is necessary for use case implementers to understand the usage constraints of the components and for components to expose suitable programmatic abstractions that permit them to be used in use cases. Currently, use cases are more like "extended" design patterns, relying on *ad hoc* conventions and manually-enforced usage compliance.

For Contract4J itself, future work will focus on completing the partial support for enforcing proper usage of the protocol. For example, while Contract4J now warns the user if contract annotations are used without the class @Contract annotation, it contains few additional enforcement mechanisms. Enhancing this support will further clarify how design pattern-like protocols can be specified in aspect-aware interfaces in a way that is programmatically executable and enforceable. This capability will be necessary to remove the current *ad hoc* nature of design patterns and make them as rigorously precise as classes and aspects themselves.

Possible extensions to Contract4J's functionality include extending the notion of a "contract" beyond the domain of DbC. For example, there is no built-in support currently for enforcing contracts *between* components, as opposed to enforcing a component's contract in isolation. Another possible enhancement is to support time-ordered event specifications, similar to [11].

Intercomponent contracts could be supported in several ways. First, it is already possible for one component to make assertions about another one, if the latter component is a bean property of the former or invocations of static (class) methods are sufficient. The test expression can simply call methods on the other component and assert conditions on the results. Without an explicit property connection, it would be necessary for Contract4J to support some sort of lookup scheme for locating instances. The bean management facilities of a framework like Spring could be leveraged, for example.

Annotations could be added to Contract4J to support temporal constructs [11]. Constraints on the order of invocation of API calls are perhaps the most common usage scenario for intercomponent contracts. However, temporal annotations could also be used to drive events in a state machine, notify observers, *etc*., which would extend these annotations beyond the contract role to more general purposes.

An alternative approach is to bypass Contract4J and to write *ad hoc* aspects that test the component relationships. The AspectJ literature is full of practical examples. This approach gives the developer maximum flexibility, but it requires AspectJ expertise and the developer doesn't get the other advantages of Contract4J, discussed previously and elsewhere [9].

The annotation approach has its limits. It is certainly not a "complete" conception of aspect-aware interfaces. It provides a convenient and terse mechanism for expressing information, which can be used to drive nontrivial aspects behind the scenes. However, it has limited expressive power and "hard-coding" an annotation in source code undermines the separation of concerns advantage of aspects, if not used judiciously.

5. CONCLUSIONS

Contract4J defines a design pattern-like protocol with two different syntax forms for specifying the contract of a component. The protocol is essentially a non-conventional "interface" that could be used by other tools as well, such as code analysis tools and IDEs. Like good interfaces, the protocol minimizes coupling between aspects and components. For some situations, this approach provides a terse, yet intuitive and reasonably-expressive way to specify metainformation about the component that can be exploited by aspects behind the scenes to support nontrivial component interactions. However, this approach has its limits and it does not replace a more complete concept of aspect interfaces.

Consistent with the work on crosscutting programming interfaces (XPIs) [2], Contract4J shows that aspect-aware interfaces, whatever their form, should contain more precise contract specifications for proper use and not just lists of methods and state information. Otherwise, the risk of breaking either the aspects or the components is great, especially as they evolve. However, programmatic enforcement of the interfaces is also essential. Fortunately, aspects also make such enforcement possible in ways that reduce the *ad hoc* nature of most pattern implementations seen today. Taken together, these facts suggest that a fruitful direction for AOP research is to explore how aspects can elevate *ad hoc* patterns of collaboration to programmatic constructs.

6. ACKNOWLEDGMENTS

My thanks to Eric Bodden, Kevin Sullivan, and Ron Bodkin for stimulating discussions. The comments of the anonymous reviewers were very helpful in clarifying the ideas in this paper.

7. REFERENCES

[1] G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM Press, 2005, pp. 49-58.

[2] W. G. Griswold, *et al.,* "Modular Software Design with Crosscutting Interfaces", *IEEE Software*, vol. 23, no. 1, 2006, 51-60.

[3] http://www.contract4j.org

[4] B. Meyer, *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, Saddle River, NJ, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns; Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[6] http://en.wikipedia.org/wiki/Test_driven_development

[7] http://jakarta.apache.org/commons/jexl/

[8] http://java.sun.com/products/javabeans/index.jsp

[9] D. Wampler, "The Challenges of Writing Portable and Reusable Aspects in AspectJ: Lessons from *Contract4J*", Industry Track, AOSD 2006, *forthcoming.*

[10] J. Hannemann and G. Kiczales, Design Pattern Implementation in Java and AspectJ. *Proc. OOPSLA '02,* ACM Press, 2002, pp. 161-173.

[11] http://www.bodden.de/studies/publications/DiplomaThesis/body_diplomathesis.php

[12] JSR-220: http://java.sun.com/products/ejb/docs.html

[13] I. Jacobson and Pan-Wei Ng, *Aspect-Oriented Software Development with Use Cases,* Addison-Wesley, Upper Saddle River, NJ, 2005.

It is Time to Get Real with Real-Time: In Search of the Balance between Tools, Patterns and Aspects

Celina Gibbs, Yvonne Coady, Jan Vitek, Tian Zhao, James Noble, and Chris Andreae

Abstract

Increasing demands for real-time systems are vastly outstripping the ability for developers to robustly design, implement, compose, integrate, validate, and enforce real-time constraints. It is essential that the production of real-time systems take advantage of approaches that enable higher software productivity. Though many accidental complexities have proven to be time consuming and problematic – type errors, memory management, and steep learning curves – we believe a disciplined approach using tools, patterns, and aspects can help. But the question as to how to best strike a balance between these approaches remains.

This paper previews Scoped Types and Aspects for Real-Time Systems (STARS), an approach aiming to guide real-time software development. In this paper, the balance between tools, patterns and aspects in this programming model is explored, and tradeoffs associated with an early prototype are identified.

1 Introduction

The Real-Time Specification for Java (RTSJ) introduces abstractions through which developers must manage resources, such as non-garbage collected regions of memory [2]. The difficulty associated with managing the inherent complexity associated with these concerns ultimately compromises the development, maintenance and evolution of safety critical code bases and increases the likelihood of fatal memory access errors at runtime.

This paper considers key tradeoffs in the design of a programming model for real-time systems. The model integrates the RTSJ abstractions with tools for verification, a disciplined approach including patterns, and language features from AOP. *Scoped Types and Aspects for Real-Time Systems* (STARS), offers a programming environment we believe that is conducive to modern software development for real-time systems. Building on the work of Scoped Types [6], STARS both guides real-time development with a simplified Scoped Type discipline and provides much needed support for the modularization and verification of real-time constraints.

1.1 Background on RTSJ

RTSJ extends the Java memory management model to include dynamically checked region-based memory known as *scoped memory areas*. A scoped memory area is an allocation context, which provides a pool of memory for threads executing in it. Individual objects allocated in a scoped memory area cannot be deallocated. Instead, an entire scoped memory area can be collected as soon as all threads exit that area.

The RTSJ defines two predefined areas for *immortal* and *heap* memory represented by the Java classes `ImmortalMemory` and `HeapMemory`, respectively, for objects with unbounded lifetimes and objects that must be garbage collected. Scoped memory areas can be nested to form a dynamic, tree-shaped hierarchy, where child memory areas have strictly shorter lifetimes than their parent. Though this structure can be well defined in terms of design, it can be easily overlooked in an implementation, resulting in a dangling reference. Since a scoped memory area could be reclaimed at any time, dynamically enforced safety rules must include checks to ensure a memory area with a longer lifetime does not hold a reference to an object allocated in a memory area with a shorter lifetime.

1.2 Related Work: Scoped Types

Scoped types are one of the latest developments in the general area of type systems for controlled sharing of references [6]. The key insight of the scoped type work was the necessity to make the scope structure of the program explicit in order to have a tractable verification procedure. STARS builds on the contribution of scoped types and proposes that every time the programmer writes an allocation expression of the form `new Object()`, it should be possible to know statically (i.e. at verification time) where the object fits in the scope structure of the program.

This paper investigates some of the tradeoffs involved in the design and implementation of the STARS programming model. The core decisions involve striking the right balance between aspects, patterns and tools in the development of real-time applications.

2 The STARS Programming Model

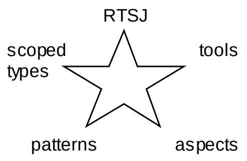


Figure 1: Elements of the STARS programming model.

STARS guides the design and implementation of real-time systems with a well defined, explicit programming model. Key elements of the model are overviewed here. Figure 1 shows that STARS consists of five elements: RTSJ, Scoped Types, aspects, patterns and tools.

When Scoped Types is coupled with a cleanly defined programming discipline, it provides static constraints and declarative specification. Patterns and aspects can provide controlled and explicit memory management. Tools of course can enhance each of these features as well as provide additional infrastructural support, such as static verification.

Static verification is important in order to save costs. For example, in a comparison between (1) a plain Java implementation, (2) an RTSJ implementation and (3) a Scoped Type implementation with static checking, on average the plain Java performs the best (due to the fact that it does not have to incur the cost of dynamic scope checks and does not have code to manage, enter and reclaim scoped memory areas). However, results show interference by the data reporting thread and garbage collection[1]. The RTSJ version is more predictable but markedly, slower. The Scoped version retains the predictability of the RTSJ version but is faster due to the elimination of most runtime checks. The following subsections overview the current Scoped Typed model for memory scopes, and the static constraints that can be established based on that model.

2.1 Modeling memory scopes

STARS looks to leverage the abstractions proposed in the Scoped Types discipline to explicitly enforce as well as modularly support reasoning about the RTSJ.

We start by considering a key simplifying feature of Scoped Types. Rather than relying on RTSJ's implicit, dynamic notion of allocation context, i.e. the last entered memory area by the current thread, we need to enforce an explicit lexically-scoped discipline which guarantees that the relative location of any object is obvious from the program text. Equally as important, we must establish a simple, clear, and *static* memory scope hierarchy in the program's code. That is, developers need a clear structural view of the memory area hierarchy of an application.

In an attempt to model the Scoped Types abstractions a restructuring of the programs package structure is required. Essentially, we equate Java packages to memory scopes. Nested packages model nested scopes. Because real-time code needs to coexist with standard Java code, we require all real-time code to be in packages nested within a package for *immortal* memory called *imm* — all the instances of classes in this package are in permanent, are not garbage collected, but are accessible from the standard Java system. Then, classes to be in scoped memory reside in scopes nested inside *imm*, where the static package nesting reflects the dynamic scope nesting at runtime.

Further, Scoped Types categorizes every class that will execute in immortal memory as either a *gate* or a *scoped class*. A scoped class is assigned to the memory scope it executes in, whereas a gate class turns scopes into first class entities, facilitating the ability for threads to enter and exit nested scopes explicitly. Each memory scope and consequently each package has just one gate class and all references from parent to child scope must proceed through it. Each gate class is therefore associated with a thread of execution and a memory scope that execution and further allocations will occur in explicitly dictated by the package it resides in.

Intuitively, this model enforces the invariant that every instance of gate class maps to a uniquely scoped memory area. Furthermore, every instance of a class defined in a scoped package *P* is allocated in the memory area of a gate defined in *P*. Operationally, whenever a method of a gate is invoked, the allocation context is switched to the scope associated with that gate. Objects allocated within a scoped package are allowed to refer to objects defined in a parent package (just as in the RTSJ objects allocated in a scope are allowed to refer to a parent scope). But as expected the converse is forbidden. At runtime, there is one scoped memory area (immortal memory) corresponding to package *imm*, and then as many scoped memory areas nested inside it as there are instances of the gate classes defined in *imm's* immediate subpackages (and so on, down through other packages nested more deeply within *imm*).

2.2 Static Constraints

We now outline the rules that ensure static correctness of STARS programs. In the following we assume that a scoped package contains at least one gate class and zero or more scoped classes. The descendant relation on package is a partial order on packages.

[1] This cannot occur in the RTSJ version as the real-time thread cannot be interrupted by a lower priority thread.

Rule 1. *An expression of a type T defined in scoped package P is visible to classes defined in P or to class defined in some package P' that is a descendant of P. An expression of a gate type G defined in package P is visible to classes defined in P' where P is a direct descendant of P'.*

This first rule encodes the essence of the RTSJ access rules. Scoped packages are those that are descendents of the *imm* package. *Gate* classes are treated differently as they are handles used from a parent scope to access a memory area. They must be accessible (visible) to the code defined in the parent scope.

Rule 2. *An expression of type T defined in scoped package P can be widened to type T' iff T' is defined in P.*

Rule 2 is used to enforce structural properties on the object graph. By preventing types to be cast to arbitrary supertypes (in particular `Object`), it is possible to verify Rule 1 statically.

Rule 3. *An method invocation of method m on an expression of scoped (gate) type T is valid iff type T implements m.*

Rule 3 prevents reference leaks, within an inherited method the receiver (i.e. `this`) is implicitly cast to the method's defining class – this could lead to a leak if the method is defined in another package.

Rule 4. *The constructor of class C defined in scope package P can only be invoked by a method defined in P.*

Rule 4 prevents a subpackage from invoking `new` on a class defined in a parent package. To do this, programmers should provide a factory method in the parent package.

Rule 5. *A class C defined in a scoped package P may not have static reference fields.*

Rule 5 prevents objects of classes defined in the same scoped package (but with different gates at runtime) from communicating via static variables. This can result in dangling references as the gates have disjoint lifetimes.

2.3 Sample Scoped Types Structure

To get a sense of what the structure a scoped type program looks like, we overview an example here. The package structure includes *ImmExample* at the top level, and two subpackages *a* and *b*. The classes in this simple example are *Many*, *RunnerA* and *RunnerB.*

```
Imm/
ImmExpample/
        ImmExample.java              🡐 Gate
        …
        RunnerA.java
        RunnerB.java
        a/
          A.java                     🡐 Gate
          …
         b/
          B.java                     🡐 Gate
         …
```

Which corresponds to the following scoped memory area structure, where the subpackages *a* and *b* are child scopes at the same level, and *ImmExample*, *A* and *B* are gates:

$$\begin{tikzcd} & \boxed{\text{heap}} \arrow[d] & \\ & \boxed{\text{imm}} \arrow[dl] \arrow[dr] & \\ \boxed{\text{a}} & & \boxed{\text{b}} \end{tikzcd}$$

3 The STARS Prototype

A STARS prototype is currently being developed. Three key elements of the STARS model considered here are the tool that verifies the static constraints, patterns and idioms to manipulate scopes, and finally, an aspect-oriented translation that takes the scoped-type Java code and weaves in the necessary low-level real-time calls to execute the code on a real-time virtual machine. The following subsections consider these in turn, with an emphasis on the aspect-oriented issues. The software system used to demonstrate the STARS approach is modeling a real-time *collision detector* (or CD) consisting of two threads: (1) a real-time thread which periodically acquires data on position of aircraft from simulated sensors, and (2) a second thread that is low priority non-real-time thread responsible for updating the display. The system must detect collisions before they happen. The number of planes, airports, and nature of flight restrictions are all variables to the system.

The CD algorithm and base implementation was written by Filip Pizlo and Jason Fox using libraries written by Ben Titzer. The system is about 25K LOC and contains a mixture of plain Java and real-time Java. To provide a proof-of-concept for our proposal, we refactored the CD to abide by the previously described static constraints. The refactoring was done in three stages. First, we designed a scope structure for the program based on the `ScopedMemory` areas used in the CD. Second, we redistributed classes between packages so that the Scoped CD package structure matches the scope structure. Third, we removed or replaced explicit RTSJ memory management idioms with equivalent constructs of our model.

3.1 Tools: Checking the Scoped Types Discipline

We must ensure that only programs following the Scoped Types discipline are accepted by STARS. The JavaCop "pluggable types" checker [1] verifies Scoped Types as a pluggable type. The key idea is that pluggable types layer a new static system over an existing lanaguage, and JavaCop can then check syntax-directed rules at compile time. JavaCop made it possible to leverage package structure in order to construct and check a relatively high-level description of the Scoped Type definitions.

3.2 Patterns and Idioms

RTSJ programmers have adopted a number of programming idioms to manipulate scopes. After changing the structure of the original CD, we needed to convert these idioms into corresponding idioms that abide by the rules established in Section 2.2. In almost every case, the resulting code was simpler and more general, because it could directly manipulate standard Java objects rather than having to create and manage special RTSJ scope metaobjects explicitly. In short, the patterns included two of those identified in [4] (*scoped run loop* and *multiscoped object*).

3.3 Aspect-Oriented Memory Management

Though the design of memory management in a real-time system may be clear, its implementation typically is not because it is inherently tangled throughout the code. For this reason we chose an aspect-oriented approach for modularizing scope management in a STARS program. This part of STARS is implemented using a subset of aspect-oriented programming extensions provided by AspectJ [5, 3].

After the program has been statically verified, aspects are composed with the plain Java base-level application. The aspects weave necessary elements of the RTSJ API into the system, and invoke some virtual machine specific extensions to ensure efficient management. This translation (and the aspects) depend critically upon the program following the Scoped Type discipline. If the rules are broken, the resulting program will no longer obey the RTSJ scoped memory discipline. As a result, either the program will fail at runtime with just the kind of an exception we aim to prevent; or worse, if running on a virtual machine that omits runtime checks, fail in some unchecked manner.

Memory management aspects in STARS can be largely generated from information provided by the declarative specification for gates and scoped packages. For each gate class, the aspect introduces two fields: `memory` and `thread`, for the memory area in which the gate executes and the thread to which the gate is bound. Though these introductions can be automatically generated by having all gates implement the following interface:

```
public interface Gate {
      private MemoryArea memoryArea;
      private Thread thread;
}
```

the programmer must specifically customize some functionality, as the RTSJ memory hierarchy provides many choices for memory areas.

The `MemoryArea` class is the abstract parent of all classes representing memory. Its subclasses include `HeapMemory`, `ImmortalMemory`, and `ScopedMemory`. Both `HeapMemory` and `ImmortalMemory` have a singleton instance obtained by invoking the `instance()` method. The `ScopedMemory` class is also abstract and its subclasses `LTMemory` and `VTMemory` provide linear time and variable time allocation of objects in scoped memory areas respectively.

All memory area classes implement the `enter()` and `executeInArea()` methods which permit application code to execute within the allocation context of the chosen memory area. Furthermore, the `getMemoryArea()` method lets one obtain the allocation context of an object – an instance of a subclass of `MemoryArea`. Finally, all memory areas support methods to reflectively allocate objects.

For an example of the ways in which the precise functionality must be customized, in one case memory may be set simply as immortal without a bound thread:

```
this.memory = ImmortalMemory.instance();
```

whereas another gate may require both a memory area and a real-time thread:

```
this.memory =
  new LTMemory(Constants.MEMSIZE,
    Constants.MEMSIZE);
this.thread =
  new RealtimeThread(
    new PriorityParameters(Constants.priority),
    null,null,null,null, this);
```

In order to efficiently manage the setting and getting of appropriate scoped memory allocation context during program execution, the STARS infrastructure relies upon two virtual machine methods customized for this purpose:

```
VM.setAllocForThread(ScopedMemoryArea memory);
ScopedMemoryArea VM.getAllocForThread();
```

More specifically, before any object is created (i.e., before all calls to *new*) or before any call to a gate method from a parent, the allocation context must be set accordingly. Subsequently, within any object initializer, the right context must be used. It is important to note that this is the only added VM support needed for STARS, and its use is localized within one STARS memory management aspect. The STARS infrastructure does provide a class `STARS` with some helper methods (such as `waitForNextPeriod` and `start()`) available for general purpose use.

In order to provide a high-level example of a memory management aspect of STARS, consider the sample code in Figure 2. In lines 6-9 the declare parents construct is used to identify gate classes and to extend the specialized thread class. The new fields required for each Gate class, memory and thread, are introduced starting on line 11. The around advice beginning on line 15 simply demonstrates the kind of functionality that the aspect can associate with all calls to new within the *detector* package. This code simply shows the new call proceeding to create an instance of a new object, and if that object is a gate, the memory field of the gate is set accordingly. As previously mentioned, this code relating to the specifics of the scoped memory area would need to be customized (not automatically generated) on a per gate basis.

```
1  import javax.realtime.*;
2
3  public aspect ScopeAspect {
4
5  // gate class
6  declare parents: App implements Gate;
7
8  // thread concerns
9  declare parents: App extends NoHeapRealtimeThread;
10
11 // fields for gate context
12 private ScopedMemoryArea Gate.memory;
13 private Thread Gate.thread;
14
15 // functionality associated with all calls to new
16 Object around(Object o):
       call(detector.*.new(..)) && this(o) {
17       Object newObj = proceed(o);
18       if (newObj instanceof Gate) {
19         Gate tmp = (Gate)newObj;
20         tmp.memory = new LTMemory(124000, 124000);
21       }
22       return newObj;
23  }
24 }
```

Figure 2. Sample memory management aspect.

4 Tradeoffs

In its current form the STARS approach impacts the logical structure of Real-time Java programs. By giving an additional meaning to the package construct, we, *de facto*, extend the language. This form of overloading of language constructs has the same rationale as the definition of the RTSJ itself, namely to extend a language without changing its syntax, compiler, or intermediate format. As for the architectural changes, this discipline imposes a different kind of functional decomposition on programs. Rather than grouping classes on the basis of some logical criteria, we group them by lifetime and function.

```
1  import javax.realtime.*;
2
3  public aspect ScopeAspect {
4  declare parents:  Mem extends IMM
5  declare parents:  Cdmem extends Mem
6  declare parents: Main implements IMM;
7
8  // classes to be created in Mem Scope
9  declare parents: (App || Detector ||
10               StateTable || Aircraft ||
11               Position) implements Mem;
12 // class to be created in CDMem Scope
13 declare parents: Frame implements CDMem;
14
15 // gate class
16 declare parents: App implements Gate;
17
18 // thread concerns
19 declare parents: App extends NoHeapRealtimeThread;
20
21 // fields for gate context
22 private ScopedMemoryArea Gate.memory;
23 private Thread Gate.thread;
24
25 // functionality associated with all calls to new
26 Object around(Object o): call(detector.*.new(..))
      && this(o) {
27  Object newObj = proceed(o);
28  if (newObj instanceof Gate) {
29    Gate tmp = (Gate)newObj;
30    tmp.memory = new LTMemory(124000, 124000);
31  }
32  return newObj;
33 }
```

Figure 3. New memory management aspect, introducing memory hierarchy for scope management.

It can be argued that this decomposition is natural; RTSJ programmers must think in terms of scopes and locations in their design. Thus it is not surprising to see that classes that end up allocated in the same scope are closely coupled, and grouping them in the same package is not unrealistic. But from a first principles perspective, the importance of a logical package structure cannot be underestimated in application development, maintenance and evolution. Could we offset this impact by escalating tool support, aspects, or both, in order to still claim the associated static guarantees?

One alternative to this overloading of package structure is the use of empty interfaces that would behave as low level flags to model the Scoped Types abstractions. That is, empty interfaces corresponding to each of an application's memory scope hierarchy are introduced, with the hierarchal relationship between those memory scopes paralleled with the *extends* relationship shown in line 4 of Figure 3. Intuitively, this enforces the single parent rule for memory areas establishing a root memory scope of *immortal memory* (imm). Further we can associate each class with a specific memory area illustrated in lines 9-13 of Figure 3, statically reflecting the dynamic scope hierarchy similar to the result of the package restructuring described in Section 2.1.

Arguably, this alternative to package restructuring does not explicitly force the same safety guarantees, such as the assignment of a class to a memory scope. Just as with the implicit requirement of non-garbage collected classes to be assigned to packages within the immortal memory package, the requirement for those classes to implement the corresponding interface can possibly be overlooked.

5 Conclusions and Future Work

It is essential that the production of real-time systems take advantage of approaches that enable higher software productivity. Through a combined approach of tools, patterns and aspects, STARS attempts to strike a balance necessary to combat real problems associated with real-time, such as type errors, memory management, and steep learning curves.

The question of how to strike a balance between tools, patterns and aspects is a matter we are currently exploring. Depending on the balance, aspects can be used solely to introduce low level RTSJ mechanisms, or their role could be more comprehensive, as part of tool support and also pattern implementations. One of the determining factors in this work will be the performance impact of aspects in RT applications. We are currently exploring these metrics.

6 References

1. Chris Andreae, Shane Markstrum, Todd Millstein, and James Noble. Practical pluggable types for java. Submitted, December 2005.
2. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.
3. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
4. Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 2004.
5. AspectJ Project. IBM, http://www.eclipse.org/aspectj/.
6. Tian Zhao, James Noble, and Jan Vitek. Scoped Types for Realtime Java. In *International Real-Time Systems Symposium (RTSS 2004)*, Lisbon, Portugal, December 2004. IEEE.

Aspect-Oriented Support for Modular Parallel Computing

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

Miguel P. Monteiro
Escola Superior de Tecnologia
Instit. Politécnico de Castelo Branco
Avenida do Empresário
6000-767 Castelo Branco 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

In this paper, we discuss the benefits of using aspect-oriented programming to develop parallel applications. We use aspects to separate parallelisation concerns into three categories: partition, concurrency and distribution. The achieved modularisation enables us to assemble a variety of platform specific parallel applications, by composing combinations of (reusable) aspect modules into domain-specific core functionality. The approach makes it feasible to develop parallel applications of a higher complexity than that achieved with traditional concurrent object oriented languages.

Keywords

Parallel computing, skeletons, aspect composition.

1. INTRODUCTION

There is a growing demand for parallel applications. The increasingly popular multi-core CPU architectures require concurrent programming to effectively leverage the underlying parallel processing capabilities. However, concurrent programming may introduce overhead in application execution time when running on a single core system. Grid systems [6] connect worldwide computing resources, delivering significant computing power. However, the systems are extremely complex to program, due to the intrinsic heterogeneity of computing resources, network latencies and bandwidths. Grid systems are built as clusters of clusters of multiprocessors machines, whose processors can be multi-core CPUs.

Traditional parallel applications suffer from classic tangling problems [10], as parallelisation concerns cut across multiple application modules. Core functionality (i.e., domain specific logic) is usually mixed with parallelisation concerns. Such concerns include work partition into parallel tasks, concurrent execution of these tasks; synchronisation of parallel accesses to shared data structures (to avoid data races) and distributed execution of the tasks. Code related to these concerns is tangled in application code, which makes it harder to understand, reuse and evolve core functionality and parallel code. Traditional parallel programming is mainly focused on performance issues. Aspect-oriented programming (AOP) [10] contributes to conciliate between high level and high performance computing, by modularising the above concerns more effectively [9].

This paper presents an approach to develop parallel applications in which parallelisation concerns are modularised into various aspects. Section 2 overviews the approach. Section 3 focuses on the development of reusable parallelisation concerns and section 4 discusses benefits and how to support composition of the aspects. Section 5 concludes the paper.

2. MODULAR PARALLELISATION CONCERNS

Our approach entails modularising parallelisation concerns with AOP, to increase general modularity and reuse potential in parallel applications. The base application must be amenable for parallelisation, since our focus is more on modularising *existing* parallel applications than parallelising sequential applications. It is harder to transform sequential applications into parallel ones than to modularise current parallel applications. The greater suitability of AOP to achieve such transformations of sequential applications relative to other approaches is as yet unproven.

Our approach involves using traditional object oriented mechanisms to implement application core functionality and implementing parallelisation concerns with AOP. Parallelisation concerns are grouped in thee categories: functional or/and data partition, concurrency and distribution. Each concern is implemented in its own aspect module that can be (un)plugged from application core functionality.

Core functionality expresses domain specific logic; by specifying what the application is supposed to do. Partition modules specify how work is performed in an efficient way, using several processing elements. Concurrency modules manage execution of parallel tasks, including synchronisation requirements. Distribution modules assign objects to available resources and manage remote method invocations.

The programmer is in charge of dividing parallel code into these concerns. In some parallel applications, core functionality code is the same as the sequential code. However, in intrinsically parallel applications (i.e., in parallel applications where there is no sequential equivalent) our core functionality module contains the domain-specific logic.

The partition module transparently replicates objects and manages method calls to the replicated objects. Replicated objects are managed by the partition aspect, which also manages their life-cycle (these are called aspect managed objects). The partition aspect controls the way a method call is executed into such objects. Usual partitions include *farming*, *divide and conquer*, *pipeline* and *heartbeat* [2]. For instance, in a pipeline partition, aspect-managed objects are organised in a pipeline sequence and each method call is successively executed by all pipeline objects. Partition module can introduce joinpoints that can be intercepted by other aspects.

The concurrency module specifies asynchronous method invocations: the caller object is allowed to proceed while the called object executes the requested method. In Java, this can be achieved by using a new thread to perform the requested method. Asynchronous method invocations may also require synchronisation to protect shared objects, to avoid data races and to ensure a specific execution order. Synchronisation code is also placed in the concurrency aspect and it resorts to synchronised block constructs and monitors provided by Java.

Partition and concurrency code is deployed in separate modules. The main idea is first to develop a partition module and next to develop the concurrency module. This way it is possible to (un)plug concurrency for debugging purposes and it also helps to avoid the inheritance anomaly problem [12], as partition code can be reused independently of concurrency constraints.

Our approach is based on distributed objects, which can be deployed across machines. Object distribution concerns are implemented in their own module as well. We identify two main benefits: (1) partition and concurrency modules can be developed without taking into account object distribution issues (i.e., they are developed for a single processor/shared memory machine) and (2) it is easier to switch between underlying middleware implementations for distribution concerns, such as CORBA, Java RMI and MPI.

A simple example (using Java and AspectJ) of the intended separation of the above concerns is described next. The Java Grande Forum (JGF) RayTracer [14] is a benchmarking application that renders an image of sixty spheres. This benchmark is already amenable for parallelisation. Actually, JGF provides both sequential and parallel versions of this application. Core functionality is very close to the JGF sequential version. It creates a *RayTracer* object, initialises it with a scene to render and then sends it a message to render the scene, specifying image size by means of an *Interval* object:

```
RayTracer rt = new RayTracer();
Scene sc = ... // create scene to render
rt.initialise(sc);
Interval interval = new Interval(0,500);
Image result = rt.render(interval);
```

The concrete partition aspect for this example intercepts the creation of the *RayTracer* object and creates a set of aspect-managed *RayTracer* objects. The concrete partition aspect broadcasts the call to method *initialise* to all the new *RayTracer* objects. It also broadcasts the call of method *render,* using a different interval for each *RayTracer* object and joins partial results produced by each one (this is a typical implementation of a simple farm partition):

```
RayTracer farm[]=new RayTracer[numberOfWorkers];

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

void around(/*scene*/) :
               call (RayTracer.initialise(..)) {
   for(int i=0; i<numberOfWorkers; i++)
      farm[i].initialise(/*scene*/);
}
```

```
Image around(/* interval */) :
                    call (RayTracer.render(..)) {
   for(int i=0; i<numberOfWorkers; i++)
      res[i] = farm[i].render(/* subinterval*/);
   ... //join sub-images saved in res array
   return(/*merged subimages*/);
}
```

In the above example, array *farm* stores the references to aspect-managed objects. Element zero of the array is used as the group front-end: it executes calls to *RayTracer* objects that are not intercepted by the partition aspect. We chose to perform explicit calls (i.e., *new*, *initialise* and *render* calls), instead of calling *proceed* to allow specific advising of joinpoints introduced by this partition aspect. Also note that all advices also must include *!within(...)* to avoid recursive advices (the *within* pointcut designator (PCD) is omitted above for simplicity). This aspect can be deployed with *pertarget* instantiation to support advices on multiple *RayTracer* instances.

The concurrency aspect spawns a new thread for each call to *initialise* (code for an asynchronous call of method *render* is also possible but trickier, as it involves the creation of a future object):

```
void around() : call (RayTracer.initialise()) {
   (new Thread() {
      public void run() {
         proceed();
      }
   }).start();
}
```

Distribution aspects redirect local object creations/calls to remote object instances. The caller local aspect plays the role of traditional proxies. However, a fake local object is required due to type system compliance. The following code presents a sketch of the code for the ray tracer benchmark:

```
RayTracer around() : call (RayTracer.new()) {
   // request object creation to remote factory
   // associate remote object to local fake
   return(/*fake local object*/);
}

void around() : call (RayTracer.initialise()) {
   // redirect call to remote object
}

Image around() : call (RayTracer.render(...)) {
   // redirect call to remote object
   return(/*remotely rendered image*/);
}
```

Table 1 presents combinations of these modules and their purpose. By modularising parallelisation concerns into multiple aspects it is possible to manage multiple configurations of a parallel application and to deploy the one that more adequately matches the target platform. When the target platform is a single processor machine, only core functionality is deployed. On multiprocessor machines, the concurrency module is included as well. However, we keep the choice over whether we include the partition module, depending on the type of parallel application (e.g., in branch and bound applications partition module usually is not required).

Table 1. Deployable parallel applications

Partition Module	Concurrency Module	Distribution module	Purpose
No	No	No	Tidy up core functionality, debugging, single processor machines
Yes	No	No	Tidy up partition strategy, debugging
No / Yes	Yes	No	Shared memory parallel machines (SMP/Multi-core)
Yes	Yes	Yes	Distributed memory machines/Grids
No	No / Yes	Yes	Distributed application

Aspect precedence is of particular importance in this approach. Partition code has the highest precedence. Contrary to the effect achieved by AspectJ's precedence mechanism, advice of partition code is *always* the first to execute, including after advice. This is required since partition code may introduce new objects and method calls that can be intercepted by other aspects. In the previous ray tracer benchmark, both concurrency and distribution aspects should be applied to aspect-managed ray tracer objects.

Order of precedence of concurrency and distribution determines the difference between client activated and server activated threads [11]. Both precedences achieve asynchronous remote method invocations. When concurrency has higher priority, threads are created to perform all remote communication in a separate local thread running on a client node, to hide network latencies and to increase network bandwidth usage.

The precedence rules ensure that parallelisation concerns are composed in the right order. However, in this composition model, each aspect only applies to joinpoints introduced by one previous module (i.e., with the next higher precedence, in the precedence list). For instance, it does not make sense to apply the distribution module to jointpoints from the core functionality and from the partition aspect. Replacing the explicit method calls in aspects by calls to *proceed* can enforce this behaviour; however, our experience reveals that this becomes trickier, since it depends on the particular implementation of the aspect weaver. Another alternative is to use the *within* PCD to specifically advise a single module, but it negatively affects the flexibility to assemble parallel applications, as it reduces the range of possible compositions.

A description of the approach, to modularise parallelisation concerns, from a parallel computing perspective, as well as a detailed case study, including performance figures, can be found in [17].

3. REUSABLE MODULES

Having modularised partition, concurrency and distribution concerns, the next step is to build reusable modules for these concerns. The modules are abstract aspects that are developed on the basis of abstract pointcuts and marker interfaces.

Reusable implementations based on abstract pointcuts follow the template advice idiom [7] that is extensively used in [8]. An abstract aspect defines the reusable crosscutting implementation. Reusable code is applied to a case-specific situation by creating a concrete aspect that inherits the logic from the abstract aspect, usually a set of abstract pointcuts and methods. The pointcuts are defined by the concrete aspects to specify the case-specific joinpoints. Inherited methods configure the logic defined in the abstract aspect by defining case-specific logic that binds the reusable part to the case-specific part.

Marker interfaces are used to implement mixin composition [3]. Concrete aspects introduce implementation of marker interfaces declared in the abstract aspect to case-specific classes. Joinpoints originating from implementing classes are captured by the aspects to apply the concern logic as with other. Marker interfaces are preferable when the aspect implements a class based role.

The reusable partition module combines the two above approaches. Each reusable aspect declares marker interfaces to specify the classes whose instances are replicated (i.e., which classes give rise to aspect-managed instances). Abstract pointcuts are concretised to specify how method calls are executed by the aspect-managed instances.

The reusable partition module implements the functionality to transparently replicate objects and redirect/broadcast method calls to aspect-managed objects. Method calls that are not captured by the aspect run on a special object, the group proxy. The structure of the reusable aspect (not shown here) is similar to the partition code as presented in previous section. The main difference is that in this case we replace references to class *RayTracer* by interfaces, abstract pointcuts and calls to *proceed*. Figure 1 presents an example of the reuse of the partition aspect. Objects of class *RayTracer* are replicated into all available processing elements. Calls to method *initialise* are broadcasted to all objects in this set (pointcut *broadcastCall*). Calls to method *render* are also executed by all elements in the set but each object receives a different argument (computed by method *scatter*, which is called by aspect *ObjectGridProtocol*). For simplicity, code implementing the merging of results of method *render* is not shown.

```
aspect Partition extends ObjectGridProtocol {

   declare parents: RayTracer implements Grid1D;

   pointcut void broadcastCall() :
               call(* RayTracer.initialise()));

   // calculates parameters of each scatterCall
   Vector scatter(Object arg) {
      Vector v = new Vector();
      ... // splits arg into sub-intervals
      return(v);
   }
   pointcut scatterCall(..) :
            call (* RayTracer.render(..)) ...;
}
```

Figure 1 – Example of reusing of partition concerns

A complete description of the reusable aspects for partition can be found in [16].

Reusable components for concurrency concerns provide functionality to perform calls in separate threads and to manage synchronisation among running threads [5]. Figure 2 presents an example in which separate threads call method *initialise*.

```
public aspect Oneway extends OnewayProtocol {
   protected pointcut onewayMethodExecution() :
      (execution(* RayTracer.initialise(..)));
```

Figure 2 Reuse of OnewayProtocol

Distribution concerns can be modularised using aspects [15][17]. However, building *reusable* abstract aspects for distribution seems to be harder. Traditional approaches require additional tools to automate the generation of distribution code. Aspects can improve on such tools by (1) reducing the amount of generated code, (2) avoiding invasive changes on the original classes [4] and (3) by using code templates [19]. Development of reusable modules in the form of abstract aspects may require an extensive usage of introspection features of Java and AspectJ, due to the specificity of Java RMI.

4. COMPOSING (REUSABLE) MODULES

Aspect composition provides a particularly interesting subject to parallel computing. Composition of partition strategies has been well studied on skeleton-based approaches [13]. The idea is to combine partition strategies to achieve more sophisticated parallelisations. One common strategy is to have a two-level farm parallelisation, where each worker is also a farmer (with its own workers). Such a partition strategy is of particular importance to large-scale parallel applications, in which a single level farm risks becoming a bottleneck. With aspect-oriented (AO) modules this composition can be implemented by specifying aspects (e.g., a partition) that act on joinpoints introduced by another aspect. In our example in Figure 1, this would mean that each *RayTracer* object in the set would also be a set of objects. Composing partition aspects can be done by capturing just the joinpoints originating from a specific aspect (e.g., the first-level partition aspect), which can be implemented using the AspectJ *within* PCD. A simple implementation based on non-reusable aspects requires the duplication of partition code presented in section 2:

```
aspect partitionLevel1 {
   RayTracer around(): call (RayTracer.new()) {
      // same as before
   }
   void around(): call (RayTracer.initialise()){
      // same as before
   }
   Image around(): call (RayTracer.render(...)){
      // same as before
   }
}

aspect partitionLevel2 {
   RayTracer around(): call (RayTracer.new())
                  && within(partitionLevel1) {
      // same as before
   }
   void around(): call RayTracer.initialise())
                  && within(partitionLevel1) {
      // same as before
   }
   Image around(): call (RayTracer.render(...)){
                  && within(partitionLevel1) {
      // same as before
   }
```

Composing concurrency and distribution aspects is also of interest. When using a two-level farming it may be more efficient to only distribute objects of the first level of the farm. This type of composition closely matches the architecture of clusters of SMP machines. To achieve such composition, the distribution aspect should be applied only to the first-level partition aspect. This can be also achieved including *within(partitionLevel1)* in all pointcuts.

Composing reusable aspects is harder. For instance, when using marker interfaces it is not possible to distinguish from aspect-managed instances of level1 and level2 farm. This requires the programmatic support of associations between objects and aspects. A similar problem occurs when explicit method calls are replaced by *proceed* to develop reusable aspects, no longer being possible to advise a particular aspect by means of the *within* designator.

5. RELATED WORK

Skeletons and templates are alternative ways to achieve the separation between core functionally and parallelisation strategies. In functional languages, the parallelisation strategy can be modelled by higher-order functions that accept functions as parameters [13]. Templates and generative patterns provide generic classes for the parallelisation strategy that can be refined to include the core functionality [1]. CO_2P_3S [18] provides an example of such a system: code that models the parallelisation strategy is generated and the user must provide application-dependent sequential hook methods.

Skeletons and templates are very close to our AO approach in that both have a similar goal: to modularise the parallelisation strategy from core functionally. One main difference between skeletons and reusable AOP modules is how parallelisation strategies and core functionality are composed together to yield a parallel application. In the former approach, core functionality must be decomposed into code fragments to fill the hooks provided by the skeleton/template. In AOP approaches this composition is based on joinpoints, which results in less invasive changes to the core functionality. The advantage of AO parallel programming is due to the richer set of mechanisms available to perform compositions between core functionality and parallelisation strategies. On the other way, templates and skeletons have the advantage to enforce stricter rules for compositions and the correct (syntactic) composition can be checked at compile-time (for example ensuring that parallelisation modules are stacked in the right order).

6. CONCLUSION

This paper discusses an AO approach to modularise parallelisation concerns, namely object partitioning, concurrency management and distribution. The approach leverages the superior compositional capabilities of AOP to obtain a higher reuse potential from parallelisation concerns.

Composing non-reusable aspects can be performed using current AOP capabilities. However, an adequate model to compose reusable aspects in a general way is left for future work.

7. ACKNOWLEDGMENTS

This work is supported by PPC-VM (Portable Parallel Computing based on Virtual Machines) project POSI/CHS/47158/2002, funded by Portuguese FCT (POSI) and by European funds (FEDER). Miguel P. Monteiro is partially supported by FCT under project SOFTAS (POSI/EIA/60189/2004).

8. REFERENCES

[1] Aldinucci, M., Danelutto, M., Teti, P., *An advanced environment supporting structured parallel programming in Java*, Future Generation Computing Systems, vol. 19, 2003.

[2] Andrews, G., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, 2000.

[3] Bracha G., Cook W., *Mixin-Based Inheritance*. ECOOP/ OOPSLA 1990, Ottawa, Canada, October 1990.

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

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

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

[7] Hanenberg, S., Schmidmeier, A., Unland, R., *AspectJ Idioms for Aspect-Oriented Software Construction*, 8th EuroPLoP, Irsee, Germany, June 2003.

[8] Hannemann, J., Kiczales, G., *Design Pattern implementation in Java and in AspectJ*, OOPSLA 2002, Seattle, USA, November 2002.

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

[10] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., *Aspect-Oriented Programming*. ECOOP'97, Jyväskylä, Finland, June 1997.

[11] Lea, D., *Concurrent Programming in Java*, Second edition, Addison-Wesley, 1999.

[12] Matsuoka S., Yonezawa A., *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*. In Research Directions in Concurrent Object-Oriented Programming (Agha G., Wegner P., et al., editors), pp. 107-150, MIT press, 1993.

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

[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., Cunha, C., Monteiro, M., *Aspect-Oriented Pluggable Support for Parallel Computing*, to be presented at VecPar'06, Rio de Janeiro, Brasil, June 2006.

[17] Sobral, J., *Incrementally Developing Parallel Applications with AspectJ*, to be presented at IEEE IPDPS'06, Rhodes, Greece, April 2006.

[18] Tan, K., Szafron, D., Schaeffer, J., Anvik, J. MacDonald, S., *Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment*, PPoPP'03, San Diego, California, USA, June 2003.

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