Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)

hrsg. von
Bram Adams, Michael Haupt, Daniel Lohmann

Technische Berichte Nr. 33

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 33

Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)

herausgegeben von

Bram Adams
Michael Haupt
Daniel Lohmann

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.d-nb.de abrufbar.

Universitätsverlag Potsdam 2010
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam**.**

Das Manuskript ist urheberrechtlich geschützt.

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2010/4122/
URN urn:nbn:de:kobv:517-opus-41221
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-41221

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-043-4

9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)

co-located with the 9th International Conference on Aspect-Oriented Software Development (AOSD) March 16, 2010, Rennes, France

Workshop Home Page: `http://aosd.net/workshops/acp4is/2010/`

Aspect oriented programming, component models, and design patterns are modern and actively evolving techniques for improving the modularization of complex software. In particular, these techniques hold great promise for the development of "systems infrastructure" software, e. g., application servers, middleware, virtual machines, compilers, operating systems, and other software that provides general services for higher-level applications. The developers of infrastructure software are faced with increasing demands from application programmers needing higher-level support for application development. Meeting these demands requires careful use of software modularization techniques, since infrastructural concerns are notoriously hard to modularize.

Aspects, components, and patterns provide very different means to deal with infrastructure software, but despite their differences, they have much in common. For instance, component models try to free the developer from the need to deal directly with services like security or transactions. These are primary examples of crosscutting concerns, and modularizing such concerns are the main target of aspect-oriented languages. Similarly, design patterns like Visitor and Interceptor facilitate the clean modularization of otherwise tangled concerns.

Building on the ACP4IS meetings at AOSD 2002–2009, ACP4IS '10 aims to provide a highly interactive forum for researchers and developers to discuss the application of and relationships between aspects, components, and patterns within modern infrastructure software. The goal is to put aspects, components, and patterns into a common reference frame and to build connections between the software engineering and systems communities.

Scope of the Workshop

The importance of "systems infrastructure" software—including application servers, virtual machines, middleware, compilers, and operating systems—is increasing as application programmers demand better and higher-level support for software development. Vendors that provide superior support for application development have a competitive advantage. The software industry as a whole benefits from an increased base level of abstraction, decreasing the need for application programmers to continually "reinvent the wheel".

These trends, however, mean that the demands on infrastructure software are increasing. More and more features and requirements are being "pushed down" into the

infrastructure, and the developers of systems software need better tools and techniques for handling these increased demands. The design and implementation of systems-level software presents unique opportunities and challenges for AOSD techniques. These challenges include the need to address the inherent complexity of infrastructure software, the need for strong assurances of correct and predictable behavior, the need for maximum run-time performance, and the necessity of dealing with the large body of existing systems software components.

This workshop aims to provide a highly interactive forum for researchers and developers to discuss the application of and relationships between aspects, components, and patterns within modern infrastructure software. The goal is to put aspects, components, and patterns into a common reference frame and to build connections between the software engineering and systems communities.

This year's workshop puts special focus on the challenges in system's programming introduced by multi-core platforms. As hardware-supported parallelization becomes mainstream, there is an increasing pressure on systems infrastructure to exploit this new parallelism to its fullest. However, the non-modular nature of parallel execution, and the numerous levels at which parallelism can be achieved (application, systems infrastructure, hardware or even a combination thereof) make it hard to come up with an intuitive, yet efficient parallel architecture. We solicited novel ideas and experience reports on this emerging research area.

Other topics in the scope of the workshop include, but are not restricted to:

- Approaches that combine or relate component-, pattern-, and aspect-based techniques
- Dimensions of infrastructure software quality including comprehensibility, configurability (by implementers), customizability (by users), reliability, evolvability, scalability, and run-time characteristics such as performance and code size
- Merits and downsides of container-, ORB-, and system-based separation of concerns
- Architectural techniques for particular system concerns, e.g., security, static and dynamic optimization, and real-time behaviour
- Design patterns for systems software
- Component, pattern, and aspect "mining" within systems code
- Application- or domain-specific optimization of systems
- Reasoning and optimization across architectural layers
- Quantitative and qualitative evaluations

The workshop is structured to encourage fruitful discussions and build connections between workshop participants. To this end, approximately half of the workshop time

will be devoted to short presentations of accepted papers, with the remaining half devoted to semi-structured discussion groups. Participants will be expected to have read the accepted papers prior to the workshop, to help ensure focused discussions.

A novelty at ACP4IS '10 is that we will invite workshop attendees to give "spontaneous" short presentations on their work if they see a relation to topics being presented and discussed at the workshop. These presentations will be limited to about ten minutes, and are intended to provide additional structured input to discussions. Spontaneous presentations will be asked for during the workshop; no paper needs to be submitted, and no publication is associated with them. There will be a session dedicated to them, just prior to discussion.

Organizers

Bram Adams is a post-doctoral fellow in the Software Analysis and Intelligence Lab at Queen's University (Canada), and is also affiliated with the SOFT lab at the Vrije Universiteit Brussel (Belgium). He obtained his PhD at the GH-SEL lab at Ghent University (Belgium). Bram has a wide range of research interests, ranging from software evolution in general, to the co-evolution of source code and the build system, and advanced separation of concerns. In his PhD, Bram developed a powerful aspect language for C (Aspicere), which he applied to large legacy C systems for reverse-engineering their behavior, re-engineering exception handling idioms and refactoring conditional compilation. Bram served in the program committees of WCRE, IWPSE and ACP4IS, and co-organized the first Practices of Linking Aspect Technology and Evolution workshop (associated with AOSD 2009). Bram is a member of the IEEE.

Michael Haupt is a post-doctoral researcher and lecturer in the Software Architecture Group at Hasso-Plattner-Institut in Potsdam. His research interests are in improving the modularity of complex software system architectures as well as in implementing programming languages, in which latter area his main focus is on faithfully regarding programming paradigms' core mechanisms as primary subjects of language implementation effort. Michael holds a doctoral degree from Technische Universität Darmstadt, where he has worked on the Steamloom virtual machine to provide run-time support for AOP languages. Michael has served as PC member for ECOOP 2008, as reviewer for TAOSD and IEEE TSE, and has been supporting reviewer for the AOSD, ECOOP, ICSE, FSE, MODELS, and VEE conference series. He has co-organized the Dynamic Aspects Workshop series in conjunction with the AOSD conferences, and the Virtual Machines and Intermediate Languages workshop series in conjunction with the AOSD and OOPSLA conferences. Michael is a member of the ACM.

Daniel Lohmann is an assistant professor at the Distributed Systems and Operating Systems group at Friedrich-Alexander University Erlangen-Nuremberg. He has been conducting research in the domain of (embedded) operating systems, software product lines, and aspect oriented programming since 2003. Daniel holds a doctoral degree from Friedrich-Alexander University; in his PhD he developed CiAO, the first purely aspect-oriented operating system. His current research activities are focused on applying AOP ideas for the fine-grained configuration of nonfunctional properties in system software and the new challenges of the many-core area. Daniel co-organized the MMB 2006 workshop on Nonfunctional Properties of Embedded Systems and the ACP4IS 2008 and 2009 workshops. Before joining the PhD programme at Friedrich-Alexander University he worked as a software developer, consultant and IT trainer. Daniel is a member of the ACM, GI, and EUROSYS.

Organization

Program Committee

Mehmet Aksit	University of Twente
Shigeru Chiba	Tokyo Institute of Technology
Eric Eide	University of Utah
Michael Engel	Technische Universität Dortmund
Franz Hauck	Ulm University
Julia Lawall	DIKU
Hidehiko Masuhara	University of Tokyo
Hridesh Rajan	Iowa State University
Doug Simon	Sun Microsystems Laboratories
Olaf Spinczyk	University of Dortmund
Eric Wohlstadter	University of British Columbia
Roel Wuyts	IMEC and K.U. Leuven

Steering Committee

Eric Eide	University of Utah
Olaf Spinczyk	University of Dortmund
Yvonne Coady	University of Victoria
David Lorenz	University of Virginia

Sponsor

The publication of this proceedings volume was sponsored by the AOSD-Europe Network of Excellence, `http://www.aosd-europe.net/`.

Table of Contents

Session 1: Adepts of Code and the Wizards of OS

Nicolas Palix, Julia L. Lawall, Gaël Thomas, Gilles Muller,
How Often Do Experts Make Mistakes? 9

Maarten Bynens, Dimitri Van Landuyt, Eddy Truyen, Wouter Joosen,
Towards Reusable Aspects: the Callback Mismatch Problem 17

Session 2: Scanners and Sensors for Components and Code

Abdelhakim Hannousse, Gilles Ardourel, Rémi Douence,
Views for Aspectualizing Component Models 21

Fan Yang, Hidehiko Masuhara, Tomoyuki Aotani, Flemming Nielson,
Hanne Riis Nielson,
AspectKE: Security Aspects with Program Analysis for Distributed Systems* 27

Session 3: Fantastic Frameworks and Infamous Infrastructures

Bholanathsingh Surajbali, Paul Grace, Geoff Coulson,
Preserving Dynamic Reconfiguration Consistency in Aspect Oriented Middleware ... 33

William Harrison,
Malleability, Obliviousness and Aspects for Broadcast Service Attachment 41

How Often do Experts Make Mistakes?

Nicolas Palix
DIKU-APL
University of Copenhagen
Denmark
npalix@diku.dk

Julia L. Lawall
INRIA Regal/LIP6
University of Copenhagen
France/Denmark
julia@diku.dk

Gaël Thomas   Gilles Muller
INRIA Regal/LIP6
France
{Gael.Thomas, Gilles.Muller}@lip6.fr

Abstract

Large open-source software projects involve developers with a wide variety of backgrounds and expertise. Such software projects furthermore include many internal APIs that developers must understand and use properly. According to the intended purpose of these APIs, they are more or less frequently used, and used by developers with more or less expertise. In this paper, we study the impact of usage patterns and developer expertise on the rate of defects occurring in the use of internal APIs. For this preliminary study, we focus on memory management APIs in the Linux kernel, as the use of these has been shown to be highly error prone in previous work. We study defect rates and developer expertise, to consider *e.g.*, whether widely used APIs are more defect prone because they are used by less experienced developers, or whether defects in widely used APIs are more likely to be fixed.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Process metrics, Product metrics; D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Measurement, Languages, Reliability

Keywords History of pattern occurrences, bug tracking, Herodotos, Coccinelle

1. Introduction

To ease development, large-scale software projects are often decomposed into multiple interdependent and coordinated modules. Software developers working on one module must then be aware of, and use properly, functions from the APIs of other modules. When a usage protocol is associated with these API functions, it must be carefully followed to ensure the reliability of the system. Large-scale software projects typically also impose coding conventions that should be followed throughout the software project and are not specific to any given API. These conventions ease code understanding, facilitate code review, and ease the maintenance process when many developers are involved in a particular piece of code and when new developers begin to work on the software project.

In this paper, we investigate the degree to which developers at different levels of expertise respect API usage protocols and coding conventions. We focus on the Linux operating system, which as an open source system makes its complete development history available. Furthermore, we focus on memory management APIs, as their use has been found to be highly error prone [3]. The Linux kernel indeed provides both a general-purpose memory management API and highly specialized variants. Thus, it is possible to compare defect rates in APIs that have a related functionality but that require different degrees of expertise to use correctly. We specifically assess the following hypotheses that may be considered to be generally relevant to open-source software:

1. Defects are introduced by less experienced developers.
2. Frequently used APIs are used by developers at all levels of experience, and thus have a high rate of defect occurrences. Nevertheless, these defects are likely to be fixed.
3. Rarely used APIs are used by only highly experienced developers, and thus have a low rate of defect occurrences. Nevertheless, these defects are less likely to be fixed.
4. Coding style conventions are well known to experienced developers.
5. The frequency of a defect varies inversely with its visible impact, *i.e.*, defects causing crashes or hangs occur less often, while defects that have a delayed or cumulative effect such as memory leaks occur more frequently.

To assess these hypotheses, several challenges must be addressed. First, we need to mine the Linux code base to find the occurrences of defective code across the history of the different versions of the software project. Next, we need to identify the developer who introduced each defect. Finally, we need a means to evaluate the level of expertise of the developer at the time the defect was introduced. To address these issues, we use the Coccinelle source code matching tool to detect defects in the uses of memory management functions, focusing specifically on code that violates the usage protocol of the memory management API, code that does not satisfy the global Linux kernel coding style, and code that uses memory management functions inefficiently. We then use the Herodotos tool [7] to correlate the defect occurrences across the different versions. Finally, we use the git [4] source code manager used by Linux kernel developers for version control in order to extract information about developer expertise.

2. Linux Memory Management APIs

In user-level code, the most common memory management functions are `malloc` and `free`. These functions are, however, not available at the kernel level. Instead, the kernel provides a variety of memory management APIs, some generic and others more special-purpose. We first describe the commonalities in these APIs and then present four Linux kernel memory management APIs in detail.

2.1 Common behavior and potential defects

All of the memory management APIs defined in the Linux kernel impose essentially the same usage protocol, as shown in Figure 1 and illustrated by the following code:

```
x = alloc(size,flag);
if (x == NULL) { ... return -ENOMEM; }
...
free(x);
```

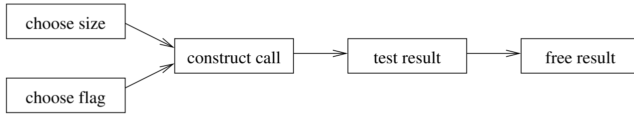

Figure 1. Usage protocol for Linux kernel memory management functions

Name	Description	Potential Impact
sizeof	Size argument expressed as the size of a type rather than the result of dereferencing the destination location.	coding style
noderef	Size argument expressed as the size of a pointer, rather than the pointed type.	buffer overflow
flag	Flag that allows locking when a lock is already held.	hang
cast	Cast on the result of an allocation function.	coding style
null test	Missing NULL test on the result of an allocation function (inverted when NULL test is not required).	crash
free	Missing deallocation of a pointer that is only accessible locally.	memory leak
memset	Explicit zeroing of the allocated memory rather than allocating using a zeroing allocation function.	inefficient
array alloc	Allocation of an array without using a dedicated array-allocating function.	buffer overflow

Table 1. Defect kinds studied

In this usage protocol, the allocation function takes two arguments: a size indicating the number of bytes to allocate and a flag indicating how the allocation may be carried out. The allocation function then returns either a pointer to the allocated region of memory or `NULL`, indicating that some failure has occurred. This result must thus be tested for `NULL` before using the allocated memory. Finally, the allocated memory should be freed when it is not useful any more, using the corresponding deallocation function.

Each step in this usage protocol introduces possibilities for defects. These defects may be violations of the Linux kernel coding style, that at best only have an impact on the maintainability of the code, or they may induce runtime errors, such as buffer overflows, hangs, crashes, or memory leaks. These defects are summarized in Table 1 and are described in detail below, for each step of the usage protocol:

Choose size The size argument to a memory allocation function is typically determined by the type of the location that stores the result of the call. One possibility is to express the size explicitly in terms of the type of this location (defect "sizeof"):

```
x = kmalloc(sizeof(struct foo),...);
```

The Linux kernel **coding style**, however, suggests to express the size as the size of the result of dereferencing the location itself:

```
x = kmalloc(sizeof(*x),...);
```

This strategy makes the size computation robust to changes in the type of `x`.

The approach preferred by the Linux kernel coding style, however, introduces the possibility of another kind of defect, in which the size is computed in terms of the pointer itself, instead of what it references, *e.g.*:

```
x = kmalloc(sizeof(x),...);
```

In this case, only a few bytes are allocated, leading to a likely subsequent **buffer overflow** (defect "noderef").

Pattern	Memory Management API			
	Standard	Node	Cache	Bootmem
basic	4 240	52	264	105
array	363	N/A	N/A	N/A
zeroing	5 125	25	96	N/A
TOTAL	9 728	77	360	105

Table 2. Number of occurrences of the memory allocation functions in Linux 2.6.32 (released December 2009)

Choose flag The flag argument indicates some constraints on the memory allocation process. The most common values are `GFP_KERNEL`, indicating that the memory allocation process may sleep if adequate memory is not immediately available, and `GFP_ATOMIC`, indicating that such sleeping is not allowed, typically because the function is called in a context in which interrupts are turned off. Using `GFP_ATOMIC` where `GFP_KERNEL` could be used can cause the memory allocation to fail unnecessarily, while using `GFP_KERNEL` where `GFP_ATOMIC` is required can **hang** the kernel (defect "flag").

Construct call The Linux kernel memory allocation functions have return type `void *`, while the location that stores the result typically has some other type, such as that of a pointer to some structure. Some programmers thus cast the result of the memory allocation to the destination type. Such a cast is, however, not required by the C standard and is against the Linux kernel **coding style** (defect "cast").

Test result If the pointer resulting from a call to an allocation function is not immediately tested for being NULL, then the first dereference of a NULL result will normally **crash** the kernel (defect "null test"). This dereference may be far from the allocation site, making the problem difficult to diagnose.

Free result In Linux kernel code, a common pattern is for one function to allocate multiple resources. Each of these allocations may fail, in which case all of the previously allocated resources must be freed. Neglecting to free allocated memory in the case of such a failure causes a **memory leak** (defect "free").

2.2 The specific APIs

The Linux kernel provides a number of different memory management APIs for different purposes. These differ in when they can be invoked and the features they provide. The APIs we consider are described below. Table 2 summarizes the usage frequency of their allocation functions.

Standard `kmalloc` is the standard memory allocation function in the Linux kernel, comparable to `malloc` at the user level. Two variants have recently been introduced. `kcalloc` was introduced in Linux 2.6.9 (October 2004) for allocating arrays. This function takes the number of elements and the size of each element as separate arguments, and protects against the case where their product overflows the size of an integer. The elements of the array are also initialized to 0. `kzalloc` was introduced in Linux 2.6.14 (October 2005) for allocating a region of memory in which all elements are initialized to 0 but that is not an array. Memory allocated with all of these functions is freed using `kfree`.

Node `kmalloc_node` targets NUMA architectures, where memory may be local to a processor or shared between a subset of the processors, and access to non-local memory is very expensive. This function thus takes an extra argument that specifies the node that should be associated with the allocated memory. `kzalloc_node` is `kmalloc_node`'s zeroing counterpart. Memory allocated with both of these functions is freed using `kfree`. Some other variants

of these functions exist that aid in debugging, but these are rarely used and we do not consider them further.

Cache `kmem_cache_alloc` allocates memory from a previously allocated memory cache. `kmem_cache_zalloc` is its zeroing counterpart. Memory allocated with both of these functions is freed using `kmem_cache_free`.

Boot These functions must be used to allocate memory during the booting process. They are analogous to `kzalloc` in that the memory is already zeroed. They furthermore always return a valid pointer, never `NULL`; in the case of an allocation error, the kernel panics. These functions do not take a flag argument. We consider only the allocation functions `alloc_bootmem`, `alloc_bootmem_low`, `alloc_bootmem_pages`, and `alloc_bootmem_low_pages`. Memory allocated with all of these functions is freed using `free_bootmem`.

In addition to the defect types outlined in Section 2.1, the different features of the memory management functions within each API introduce the possibility of using one of these functions in the wrong situation. In terms of defects, we consider cases where the zeroing and array allocating variants, if available, are not used and the corresponding code is inlined into the call site (defects "memset" and "array alloc", respectively). For the `alloc_bootmem` functions, which do zero the memory and do not return `NULL`, we consider code that performs unnecessary zeroing and `NULL` test operations. These mistakes essentially only impact the efficiency of the code, but may also impact readability, and thus subsequent code maintenance.

3. Tools

To carry out our study, we use the following tools: 1) Coccinelle to find occurrences of defects in recent versions of the Linux source tree, 2) Herodotos to correlate these occurrences across multiple versions, and 3) git to identify the developer responsible for introducing each defect occurrence and to obtain information about the other patches submitted by this developer. Coccinelle is applicable to any software implemented in C. Herodotos is applicable to any software at all, as it is language-independent. Git can also be used to access developer information for any software, as long as it or some compatible tool has been used as the version control manager during the software's development.

3.1 Coccinelle

Coccinelle is a tool for performing control-flow based pattern searches and transformations in C code [2, 6]. It provides a language, SmPL, for specifying searches and transformations and an engine for performing them. In this work, we use SmPL to create patterns representing defects and then use Coccinelle to search for these patterns across different versions of the Linux source tree. Patterns are expressed using a notation close to source code, but may contain *metavariables* to represent arbitrary sub-terms. A metavariable may occur multiple times, in which case all occurrences must match the same term. SmPL furthermore provides the operator "`...`", which connects separate patterns that should be matched within a single control-flow path. This feature allows, for example, matching an execution path in which there is first a call to a memory allocation function and then a return with no intervening save or free of the allocated data, amounting to a memory leak. More details about Coccinelle, including numerous examples, are found in previous work [2, 5, 6].

3.2 Herodotos

To understand how defects have been introduced in the Linux kernel, we have to *correlate* the defect occurrences found by Coccinelle across multiple versions. Indeed, the position of a defect may change across versions due to the addition or removal of other code in the same file. To correlate defect occurrences, we use the Herodotos tool [7]. Herodotos uses Unix diff to identify the differences in each affected file from one version to the next and thereby predicts the change in position of a defect. If a defect of the same type is reported in the predicted position in the next version, they are considered to be the same defect. Otherwise, the defect is considered to have been corrected. Herodotos also can be configured to produce a wide variety of graphs and statistics representing the defect history.

Pattern	Memory Management API			
	Standard	Node	Cache	Bootmem
sizeof	30.64%	28.57%	N/A	16.19%
noderef	0	0	N/A	0.95%
flag	0.01%	0	N/A	N/A
cast	0.79%	2.60%	6.39%	29.52%
null test	1.04%	6.49%	3.06%	8.57%
free	0.10%	0	0.56%	0
memset	2.92%	1.92%	3.03%	1.90%
array alloc	3.32%	2.60%	0.28%	0.95%

Table 3. Comparison for Linux 2.6.32

3.3 Git

Since version 2.6.12 (June 2005) Linux has used the `git` version control system [4]. Git maintains a graph representing the project history, including commits, forks and merges. Each of these operations is referenced by a SHA1 hash code. This hash code gives access to the changes in the repository and some related meta-information. For instance, Git registers the name and the email of the author and the committer of a change, short and long descriptions of the change, and the date on which the change was committed.

Git includes various options for browsing the commit history. In this work, we use git to trace the contributions of each developer. Starting from the earliest version in which Coccinelle finds a given defect, we use the *blame* option to find the name of the developer who has most recently edited the defective line in a prior commit. To evaluate the level of expertise of this developer, we then count the number of patches from this developer that were accepted prior to the one introducing the defect and the number of days between the developer's first accepted patch and the defective one. We consider the level of expertise of the developer to be the product of these two quantities.

4. Assessment

We now assess the hypotheses presented in Section 1 for the memory management APIs. To support our assessment, we have collected various statistics. Table 3 presents the percentage of defect occurrences as compared to all occurrences of each kind of memory allocation function for Linux 2.6.32, which is the most recent version. Figure 2 presents the same information, but for all versions since Linux 2.6.12. The defect Flag is omitted, because its frequency is very close to 0. Figure 3 presents the average lifespan of these defects. Finally, Figure 4 presents the number of developers introducing each kind of defect (on the X axis) and their average level of expertise, calculated as described in Section 3.3.

Our assessment of each of the five hypotheses is as follows:

Defects are introduced by less experienced developers Figure 4 shows that in most cases, the expertise of the developers who introduce defects is indeed low, *i.e.*, they have participated in kernel development for only a short time and have submitted only a few patches. But for two defect types for the Node API and for one

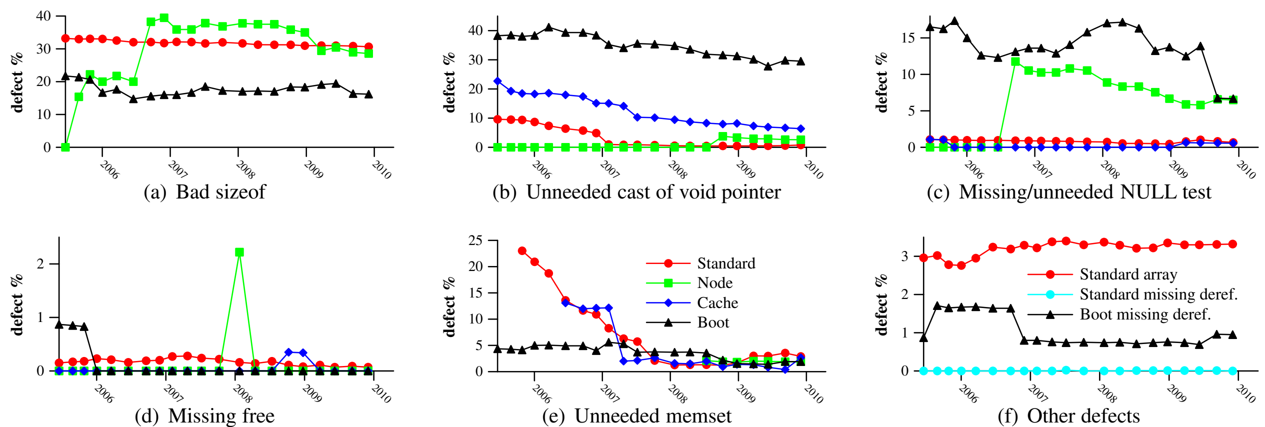

Figure 2. Defect ratio per uses for each defect kind

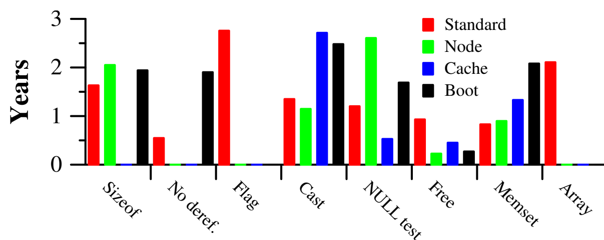

Figure 3. Average defect lifespan

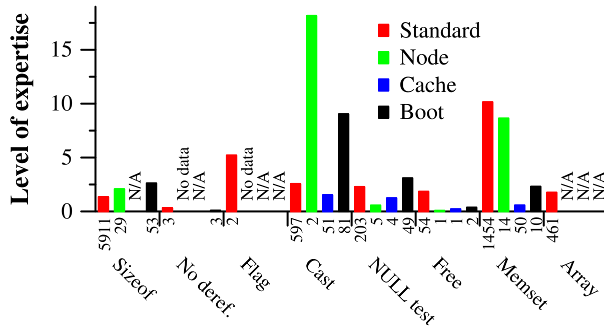

Figure 4. Developer expertise per allocator kind. No data means no defects of the given type. N/A means the defect type is irrelevant to the given API.

defect type for the Boot API, the average level of expertise is relatively high. We conjecture that these APIs are mostly used by experienced developers, and thus only experienced developers introduce the defects. For Node, at least, the number of developers in these cases is also very small.

Frequently used APIs have a high rate of defects, but these defects are quickly fixed The Standard API is used much more frequently than the others, and in some cases, notably sizeof and array, it has a higher defect frequency as well. But it also has a lower defect frequency for the remaining defect kinds. Some defect kinds show a slight or substantial decrease in their frequency for the Standard API, notably the use of `memset` rather than the zeroing function `kzalloc`. Such defects have an average lifespan of around one year, while the lifespan of the comparable defect for the Boot API is two years. Other defect kinds essentially remain steady, notably the non-use of the array allocation function `kcalloc`, which has an average lifespan of two years. The frequency of this defect, however, is consistently very low.

Rarely used APIs have a low rate of defects, but these defects are rarely fixed The data in Figure 2 does not support the hypothesis of a low rate of defects, as it is often the rarely used APIs Node, Cache, and Boot that have the highest frequency of defects. For example, Boot has the highest rate of improper NULL tests. In this case, the API has the special property that a NULL test is not needed, and thus the results show that developers are not fully familiar with the features of this API. The defect rates are relatively stable, without the substantial drops over time seen in the case of the much more frequently used Standard. Defects also have a long lifespan, notably of typically two years or more for Boot.

Coding style conventions are well known to experienced developers This hypothesis is also not supported by Figure 2. The coding style defects Sizeof and Cast indeed have the highest frequency of all of the considered defects, and the frequencies are highest for Node and Boot, respectively. These are are highly specialized APIs and thus are only likely to be used by experienced developers. It may be that such developers have a more specialized focus, and are thus not aware of these conventions. Or these APIs may be considered less often when doing coding style cleanups.

The frequency of a defect varies inversely with its impact Noderef is likely to lead to a kernel crash, as much less memory is allocated than intended. Flag can cause a kernel hang. A missing free can cause a memory leak. All of these defect kinds do indeed occur very infrequently, as shown by Table 3. Missing NULL tests, however, are relatively frequent, found in up to 10% of all occurrences for the Node API. We conjecture that the memory allocation functions do not return `NULL` very often, and thus the impact of the defect is not seen very often in practice.

5. Related work

The closest work to this one is our paper at AOSD 2010 [7], which presents Herodotos. The experiments in that paper have a larger scope, as they consider four open source projects and a wider range of defects. In this paper, on the other hand, we consider in more depth the defects in the use of a single type of API, in one software project. We have also added the assessment of developer expertise to the collected statistics. In future work, we will apply the analyses presented here to the wider set of examples considered in the AOSD paper.

Zhou and Davis assess the appropriateness of statistical models for describing the patterns of bug reports for eight open source projects [9]. They do not, however, distinguish between different defect types, nor do they study the level expertise of the developers who introduce the bugs. Chou *et al.* [3] do consider specific bug types in earlier versions of Linux, but do not study developer expertise.

Anvik and Murphy [1], and Schuler and Zimmermann [8] propose approaches to determine implementation expertise based on mining bug reports and code repositories. However, they determine who has expertise on a particular piece of code while we want to investigate the expertise of a developer who commits code containing a particular defect.

6. Conclusion

In this paper, we have studied the history of a set of defect types affecting a range of memory management APIs in the Linux kernel code, considering both the percentage of defects as compared to the total number of occurrences of each considered function and the expertise of developers that introduced these defects. Based on this information, we have assessed a collection of hypotheses, differentiating between widely used APIs and more specialized ones. The hypotheses that the developers who introduce defects have less experience, that defects in the use of widely used APIs are fixed quickly, and that defects in the use of rarely used APIs tend to linger are largely substantiated. On the other hand, the percentage of defect occurrences does not appear to be correlated to the frequency of use of the API, expert developers do not seem to be more aware of coding conventions than less expert ones, and the frequency of a defect is not always inversely related to its potential impact. More work will be required to assess these hypotheses in the context of Linux and the memory management APIs, as well as other software projects and APIs.

Availability
The data used in this paper are available at
`http://www.diku.dk/~npalix/acp4is10/`.

References

[1] J. Anvik and G. C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, Minneapolis, USA, May 2007. IEEE Computer Society.

[2] J. Brunel, D. Doligez, R. R. Hansen, J. Lawall, and G. Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, Jan. 2009.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, Oct. 2001.

[4] Git: The fast version control system. `http://git-scm.com/`.

[5] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN 2009)*, pages 43–52, Estoril, Portugal, June 2009.

[6] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[7] N. Palix, J. Lawall, and G. Muller. Tracking code patterns over multiple software versions with Herodotos. In *Proc. of the ACM International Conference on Aspect-Oriented Software Development, AOSD'10*, Rennes and Saint Malo, France, Mar. 2010. To appear.

[8] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, Leipzig, Germany, May 2008.

[9] Y. Zhou and J. Davis. Open source software reliability model: an empirical approach. In *5-WOSSE: Proceedings of the fifth workshop on Open source software engineering*, pages 1–6, St. Louis, MO, USA, 2005. ACM.

A. SmPL files

A.1 alloc_size

```
virtual org

@ r depends on org disable sizeof_type_expr @
type T,T1;
T *x;
expression n;
position p;
@@

(
x@p = (T1)\(kmalloc \| kzalloc \)(<+... sizeof(T) ...+>, ...)
|
x@p = (T1)kcalloc(n, <+... sizeof(T) ...+>, ...)
)

@script:python@
p << r.p;
x << r.x;
xtype << r.T;
@@

msg = "var: %s type: %s " % (x,xtype)
coccilib.org.print_safe_todo(p[0],msg)
cocci.include_match(False)
```

A.2 alloc_noderef

```
virtual org

@ r depends on org @
expression *x;
expression n;
position p;
@@

(
 \(kmalloc \| kzalloc \)(<+... sizeof@p(x) ...+>, ...)
|
 kcalloc(n, <+... sizeof@p(x) ...+>, ...)
)

@bad_deref@
position r.p;
expression e;
constant c;
@@

\( sizeof@p(*e) \| sizeof@p(c) \)

@script:python depends on !bad_deref@
p << r.p;
x << r.x;
@@

msg = "var: %s" % (x)
coccilib.org.print_safe_todo(p[0],msg)
cocci.include_match(False)
```

A.3 gfp_kernel

```
virtual org

@r depends on org@
expression E1;
position p;
@@

(
read_lock_irq
|
write_lock_irq
|
read_lock_irqsave
|
write_lock_irqsave
|
local_irq_save
|
spin_lock_irq
|
spin_lock_irqsave
) (E1,...);
... when != E1
    when any
 \(kmalloc\|kcalloc\|kzalloc\)(...,<+... GFP_KERNEL@p ...+>)

@script:python@
p << r.p;
@@

msg="%s::%s" % (p[0].file, p[0].line)
coccilib.org.print_todo(p[0], msg)
cocci.include_match(False)
```

A.4 cast_alloc

```
virtual org
virtual patch

@ depends on patch && !org disable drop_cast @
type T;
@@

- (T *)
 \(kcalloc \| kmalloc \| kzalloc \)(...)

@r depends on !patch && org disable drop_cast @
type T;
position p;
@@

 (T@p)\(kmalloc\|kzalloc\|kcalloc\)(...)

@script:python@
p << r.p;
t << r.T;
@@

msg = "%s" % (t)
coccilib.org.print_todo(p[0], msg)
cocci.include_match(False)
```

A.5 alloc_nulltest

```
virtual org

@r depends on org@
type T;
expression *x;
identifier f;
constant char *C;
position p1,p2;
@@

x@p1 = (T)\(kmalloc\|kcalloc\|kzalloc\)(...);
... when != x == NULL
    when != x != NULL
    when != (x || ...)
(
kfree(x)
|
f(...,C,...,x,...)
|
f@p2(...,x,...)
|
x->f@p2
)

@script:python@
x << r.x;
p1 << r.p1;
p2 << r.p2;
@@

msg = "%s" % (x)
coccilib.org.print_todo(p1[0],msg)
coccilib.org.print_link(p2[0])
cocci.include_match(False)
```

A.6 kmalloc

```
virtual org

@r exists@
type T;
local idexpression x;
statement S;
expression E;
identifier f,l;
position p1,p2,p3;
expression *ptr != NULL;
@@

(
if ((x@p1 = (T)\(kmalloc\|kzalloc\|kcalloc\)(...)) == NULL) S
|
x@p1 = (T)\(kmalloc\|kzalloc\|kcalloc\)(...);
...
if (x == NULL) S
)
<... when != x
     when != if (...) { <+...x...+> }
(
goto@p3 l;
|
x->f = E
)
...>
(
 return \(0\|<+...x...+>\|ptr\);
|
 return@p2 ...;
)

@script:python@
x << r.x;
p1 << r.p1;
p2 << r.p2;
@@

coccilib.org.print_todo(p1[0], x)
for p in p2:
        coccilib.org.print_link(p)
cocci.include_match(False)
```

A.7 kzalloc

```
// Options: -no_includes -include_headers
virtual org

@r depends on org@
type T;
expression x;
expression E1,E2;
position p1,p2;
statement S;
iterator I;
@@

x = (T)kmalloc@p1(E1,E2);
... when != x
if (x == NULL || ...) { ... return ...; }
... when != x
    when != for(...;...;...) S
    when != while(...) S
    when != I(...) S
memset@p2(x,0,...);

@s depends on org exists@
position r.p1,r.p2;
@@

... when != kmalloc@p1
memset@p2(...);

@script:python depends on !s@
p1 << r.p1;
x << r.x;
@@

msg="%s" % (x)
coccilib.org.print_safe_todo(p1[0], msg)
cocci.include_match(False)
```

A.8 kcalloc

```
// Options: -no_includes -include_headers
virtual org

@r depends on org exists@
expression E1, E2,E3;
position p;
constant C1, C2;
@@

(
 kmalloc(C1 * C2, E3)
|
 kzalloc(C1 * C2, E3)
|
 kmalloc@p(E1 * E2, E3)
|
 kzalloc@p(E1 * E2, E3)
)

@script:python@
p << r.p;
E1 << r.E1;
E2 << r.E2;
@@

msg="%s ## %s" % (E1, E2)
coccilib.org.print_safe_todo(p[0],msg)
cocci.include_match(False)
```

B. Excerpt of the HCL file

```
prefix="."
patterns="./cocci"
projects="/var"
results="./results"
website="./web"
findcmd="spatch.opt %f -sp_file %p -dir %d > %o"
flags="-timeout 60 -use_glimpse -D org"

project Linux {
 scm = "git:linux.git"
 dir = "linuxes"
 color = 1 0 0
 linestyle = solid
 marktype = circle
 versions = {
  ("linux-2.6.12", 06/17/2005, 4155826)
  [...]
  ("linux-2.6.32", 12/02/2009, 7663555)
 }
}

[...]

pattern std_cast {
 file = "std/cast_alloc.cocci"
 color = 1 0 0
}

[...]

graph gr/std.jgr {
 xaxis = date
 xlegend = ""
 yaxis = sum
 ylegend = "# of defects"
 project = Linux
 curve pattern std_size
 curve pattern std_noderef
 curve pattern std_gfp
 curve pattern std_cast
 curve pattern std_nulltest
 curve pattern std_free
 curve pattern std_zalloc
 curve pattern std_calloc
}

graph gr/cumul-exp.jgr {
 xaxis = groups
 xlegend = ""
 yaxis = expertise
 ylegend = "Level of expertise"

 legend = "defaults fontsize 6 x 25 y 15"

 project=Standard
 project=Node
 project=Cache
 project=Boot

 [...]

 group "Cast" {
  curve project Standard pattern std_cast
  curve project Node     pattern node_cast
  curve project Cache    pattern cache_cast
  curve project Boot     pattern boot_cast
 }

 [...]
}
```

Towards Reusable Aspects: the Callback Mismatch Problem

Maarten Bynens, Dimitri Van Landuyt,
Eddy Truyen and Wouter Joosen

DistriNet, Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{maarten.bynens,dimitri.vanlanduyt,
eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract

Because software development is increasingly expensive and time-consuming, software reuse gains importance. Aspect-oriented software development modularizes crosscutting concerns which enables their systematic reuse. Literature provides a number of AOP patterns and best practices for developing reusable aspects based on compelling examples for concerns like tracing, transactions and persistence. However, such best practices are lacking for systematically reusing invasive aspects.

In this paper, we present the 'callback mismatch problem'. This problem arises in the context of abstraction mismatch, in which the aspect is required to issue a callback to the base application. As a consequence, the composition of invasive aspects is cumbersome to implement, difficult to maintain and impossible to reuse.

We motivate this problem in a real-world example, show that it persists in the current state-of-the-art, and outline the need for advanced aspectual composition mechanisms to deal with this.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.2.11 [*Software Engineering*]: Software architectures—Information hiding,Languages,Patterns

General Terms Design, Documentation

Keywords reusable aspects, invasive aspects, aspect adapter

1. Introduction

Current AOP languages and approaches often result in aspects that are tightly coupled to the base classes they act upon. For example, it is a common technique to write advice code that involves join point reflection to find out the necessary contextual information [11]. Such advice code typically hard-codes assumptions about the structure and behavior of the base classes. This has a number of negative consequences: the aspect must be maintained together with the base, which makes it difficult to develop aspects and base in parallel, and leads to fragility of aspectual composition (lack of robustness). Additionally, the resulting aspects and their compositions are very specific to the scope of one application, and thus not reusable, for example in an aspect library.

To address these problems, the current state-of-the-art provides a number of techniques and patterns that involve introducing an abstraction layer between the base and the aspect. Examples of this are pointcut interfaces [6], annotations and marker interfaces. Introducing an abstraction layer enables the design of reusable aspects, in the sense that the required interface of the aspect (the elements it needs from the base to perform its function) can be specified uniquely in terms of abstractions that are relevant in the scope of the aspect itself. For example, the required interface of a reusable *authentication* aspect would be defined in terms of aspect-specific abstractions such as the *principal*, *credentials*, etc. To compose this *authentication* aspect to the base application, the developer must implement and provide these abstractions, by mapping elements of the base application (e.g. a *customer* in a web shop) to the aspect abstractions (the principal). Because the aspect is less tightly coupled to the base application, it can be reused more easily across applications.

A common problem in the design of reusable aspects is that of *abstraction mismatch*. This occurs when the elements of the base are not fully compatible with the abstraction required by the aspect. For example, the *credentials* abstraction may consist of a password that is encoded in MD5 —meaning that the *authentication* aspect expects passwords to be provided in MD5—, while the base offers the password in plain text. The solution to this is to introduce an adapter [5] that converts the base abstraction into the aspect abstraction. In the example, the adapter would be responsible for applying the MD5 hash function to the password that is provided by the base and providing the result to the aspect.

These techniques are sufficient to realize a loose coupling between aspect and base for both *spectative* and *regulative* aspects [9]; i.e. aspects that respectively observe the base application without affecting its functionality, or observe the base application and redirect or block the thread of execution in some cases. However, there is a lack of similar patterns or solutions for *invasive aspects* that issue callbacks to the base application to change its state or its behavior.

In this paper, we highlight this problem, which we call the *callback mismatch* problem. This problem arises (i) in the occurrence of abstraction mismatch, and (ii) when the aspect is required to issue a callback to the base application. As a consequence, the composition specification of such aspects becomes cumbersome to implement, difficult to maintain and impossible to reuse.

The structure of this paper is as follows. First, we define and illustrate the *callback mismatch* problem in a case study and we show that this is a realistic problem in the context of parallel development and reuse of aspectual modules. Then, we show that in the current state-of-the-art in aspect-oriented programming (AOP) and related techniques, patterns and notations, this problem persists and there is a need for advanced aspectual composition mechanisms to deal with this issue.

2. The callback mismatch problem

2.1 Problem definition

Pointcuts abstract not only from interesting join points in the base program but also expose relevant context data available at these join points. Abstraction mismatch is the problem where the representation of these abstractions in the base program is not compatible with the representation in the aspect. Dealing with abstraction mismatches is easy by employing a binding aspect that extracts the necessary information from the available base abstractions.

In the presence of callbacks however, specifying such a binding aspect becomes problematic. Callbacks happen when the reusable aspect uses the data and/or the behavior of the base application exposed by a pointcut to intervene in the normal control flow. As presented in Section 1, callbacks are mostly used to realize invasive aspects. To bind the callback to the base program, the binding aspect needs to include adapter functionality that routes the callback to the same base object that triggered the reusable aspect in the first place.

This problem is more complex to overcome than traditional problems with object-oriented libraries and frameworks (e.g. API mismatch). As the reusable aspect is never explicitly called from the base program, the adapter (or in this case the binding aspect) needs to adapt in both directions. It has to make sure that the relevant join points are translated to the aspect abstractions and that callbacks refer back to the original object. As a result, dealing with the callback mismatch problem takes more than solving the mismatch separately in both directions.

In summary, the *callback mismatch problem* leads to the following:

Problem summary. In the current state-of-the art of AOP languages, patterns and best practices, the required composition logic for dealing with both (1) *abstraction mismatch* and (2) *callbacks* is cumbersome to implement, difficult to maintain and impossible to reuse.

2.2 Motivating Example

To illustrate the problem outlined in this paper, we present a simplified example from the car crash management system (CCMS) [10, 16]. This is a large-scale and realistic distributed application that helps the authorities dealing with car crashes more efficiently by (i) centralizing all information related to a car crash, (ii) proposing a suitable crash resolution strategy, (iii) dispatching resource workers (e.g. first-aid workers) to the crash site, and (iv) reassessing the strategy in real-time when new information comes in.

To avoid wasting resources on prank calls and witnesses assuming a false identity, the correct and efficient functioning of the CCMS depends highly on *witness identity validation*, which is implemented in the CCMS as an aspect. More specifically, as long as the system has not successfully validated the identity of the witness, the CCMS will operate in *limited mode*, meaning that only a restricted set of resources can be assigned to that particular car crash.

Figure 1 presents this aspect in detail. The sequence starts when a witness calls the crisis center to report a car crash. The coordinator answering the call enters the name and phone number of the witness into the CCMS.

In this example, `Witness` represents the base abstraction: it provides the information needed by the aspect.

The *witness identity validation* aspect is provided in the form of a reusable identity validation aspect `IdentityValidation`. The required aspect abstractions in this example are `Person` and `ValidationReport`.

This illustrates *abstraction mismatch* in this example: the provided abstraction of the base application is the `Witness` which encapsulates the name, the phone number, and the validity state of the witness. On the other hand, the required interface of the aspect consists of (1) the `Person` abstraction which encapsulates first and last name and phone number, and (2) `ValidationReport` abstraction which encapsulates the validity state.

As pointed out in Section 2.1, this issue can be resolved by specifying a binding aspect with adapter functionality. In this example, we have implemented a *class adapter* which adapts the interface of the `Witness` object to match those of `Person` and `ValidationReport` (message 2).

After this, the aspectual composition with the *witness identity validation* aspect is realized. More specifically, the pointcut for this aspect is specified in terms of the `Person` interface (message 3). Both the `Person` and the `ValidationReport` are exposed through these join points.

Finally, the `IdentityValidation` component contacts a third-party telecom operator to check whether the presented person is indeed listed under the given phone number. The result of this verification activity is set via the `ValidationReport` interface. Because the `Witness` object has previously been adapted to this interface by the adapter, the callback ends up at the witness (message 4).

Section 2.3 illustrates in further detail how the adapter code is affected by the callback mismatch problem.

If the adapter is implemented incorrectly, the CCMS itself will remain in limited mode, and thus addresses the car crash inefficiently, if at all. The fact that the correct functioning of the entire application depends fully on the correct realization of the callback stresses the importance of writing an adapter that realizes the desired behavior in a comprehensible, maintainable and reusable manner.

2.3 Minimal solution in AspectJ

An example implementation of the scenario is included in the appendix. The pointcut `personIdNeedsChecking` in aspect `IdentityValidation` is defined in terms of types `Person` and `ValidationReport`. `Person` contains the data that needs to be checked and `ValidationReport` captures the result of the validation. Since these types are not directly supported by the base code, an adapter needs to be written to bind the aspect to the application. Listing 1 shows the adapter.

```
public aspect Adapter extends IdentityValidation{            1
                                                             2
  declare parents: Witness implements                        3
      ValidationReport;
  public void Witness.validation(boolean b){                 4
    validate(b);                                             5
  }                                                          6
                                                             7
  declare parents: Witness implements Person;                8
  public void Witness.setFirstName(String s){}               9
  public void Witness.setLastName(String s){}                10
  public String Witness.getFirstName(){                      11
    return getName().split(" ")[0];                          12
  }                                                          13
  public String Witness.getLastName(){                       14
    return getName().split(" ")[1];                          15
  }                                                          16
  void around(Person w): execution(*                         17
      Witness.setName(String)) && this(w){
    proceed(w);                                              18
    w.setFirstName(w.getFirstName());                        19
    w.setLastName(w.getLastName());                          20
  }                                                          21
                                                             22
  pointcut report(ValidationReport report):                  23
      this(report);
}                                                            24
```

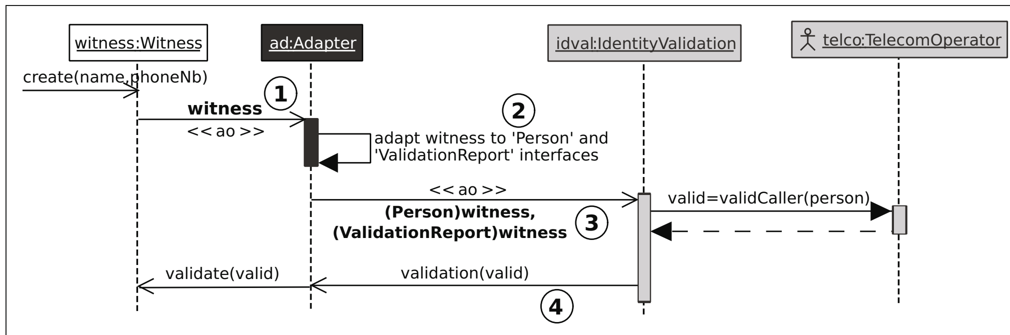

Figure 1. UML sequence diagram to illustrate the role of the adapter (in dark gray), and the *witness identity validation* aspect (in grey).

Listing 1. Example implementation of the adapter

In this scenario, the adapter has two responsibilities. Firstly, it needs to make sure that the callbacks through `ValidationReport` and `Person` are reified in the witness object. Therefore, the class `Witness` is made to implement the types `ValidationReport` (lines 3–6) and `Person` (lines 8–16) by means of *declare parents* and inter-type declarations (methods `setFirstName` and `setLastName` do not need an actual implementation because they are not used as a callback). Secondly, it needs to propagate the relevant join points on `Witness` as required join points on `Person`. This is achieved by around advice that calls *proceed* and additionally calls the appropriate methods (lines 17–21). Because there is a mismatch in the sense that `Person` has separate concepts for first name and last name, extra mapping functionality is required.

This example shows that even in this simple (almost trivial) case, defining the adapter is already a cumbersome task. One that needs to be repeated for every mismatch.

3. Approaches

This section gives an overview of existing AOP languages, techniques and patterns that are related to the problem and briefly argues that none of them sufficiently addresses the callback mismatch problem.

3.1 Explicit Pointcut Interfaces

Approaches like pointcut interface[6], XPI[15] and explicit join points[8] do not help to define bidirectional adapters more easily. The problem is that the aspect will always use a type description to be able to issue callbacks. This type should then be mapped to a concrete type in the base code. The approaches mentioned describe join points and not types and thus cannot be used in this mapping.

In the simplified case, the aspect specifies an abstract pointcut and the callback is issued on one of the exposed parameters. An explicit pointcut interface can help with implementing this abstract pointcut, but the mapping of the callback to the base code still needs to be done. A standard unidirectional adapter is sufficient in this case.

3.2 Type parameters

At first sight, type parameters seem to solve the callback mismatch problem, since an instantiated type parameter will behave as an alias for a concrete type of the base code. Unfortunately, for the aspect to be able to issue callbacks, it needs to refer to an actual type (and e.g. use it as a bound for the type parameter). As a result we end up with the same problems as before.

3.3 Caesar

Caesar supports on-demand remodularization to integrate independent components. Its model is object-based and uses virtual types, mixin composition and the concept of wrapper recycling [12]. As a result, Caesar provides a means to specify expressive, reusable adapters. However, Caesar does not support remodularization of aspect abstractions. In Caesar, the aspect composition is part of the binding and requires manual object wrapping (assisted by dynamic wrapper selection and wrapper recycling) [1, 13]. We can conclude that Caesar doesn't offer a solution to the callback mismatch problem as it not aims to bind abstract aspect compositions.

3.4 Subject-oriented programming

Subject-oriented programming [7] and its descendants Hyper/J and Theme[4] (which all involve Multi-Dimensional Separation of Concerns (MDSOC)) represent a more symmetrical approach to AOSD, meaning that each concern is developed independently. One of the key features of these approaches is *declarative completeness*, meaning that each concern explicitly defines the structure and behavior of the classes it depends on. To assemble an application, these concerns are composed using composition rules. Composition directives includes mechanisms for name-based merging of classes and methods, and support for renaming, overriding, . . .

Because these mechanisms are nondirectional, they are inherently adequate for specifying callbacks. However, the composition mechanisms are not expressive enough to resolve sophisticated abstraction mismatches that can only be resolved with complex adapters involving more than renaming, overriding and merging classes and methods. Therefore these approaches do not solve the abstraction mismatch problem.

4. Conclusion

This paper introduces the *callback mismatch* problem. In essence, this problem is triggered by two key elements: (i) *abstraction mismatch* which is resolved by applying the Adapter design pattern [5], and (ii) *invasive* aspects [9], i.e. aspects that issue a callback to the base application to change its state or behavior. This situation leads to composition logic that is cumbersome to implement, difficult to maintain and impossible to reuse.

We have illustrated the problem in a minimal example from a realistic case study. Additionally, we have presented a number of factors that deteriorate this problem. Finally, we outline a number of related approaches in which this problem persists.

From this, we conclude that the current state-of-the-art is currently is not capable of solving the callback mismatch problem adequately. In our opinion, there are three distinct research directions

to be explored for a solution to this problem: (i) next-generation language constructs that allow the described adapters to be defined more elegantly and concisely (e.g. inspired by Caesar and SOP that provide disjoint sets of constructs that solve the problem partly), (ii) middleware-based solutions and framework-specific services that are capable of hiding most of the described adapter complexity, or (iii) AOP design patterns that provide reference solutions to this problem.

Logging, tracing and authentication are aspects addressed pervasively throughout AOSD research. Based on the impact that these aspects have on the base application, they are characterized as either *spectative* or *regulative* [9]. The large body of research into these particular aspects classes suggests that they are well-known, and they can sufficiently be dealt with by current AOP techniques. As AOP matures, it is our opinion that the research focus should shift from *spectative* and *regulative* aspects towards more *invasive* aspects, which represent the most challenging class of crosscutting concerns. We believe that the problem brought to the forefront in this paper is a key hurdle in the road towards advanced AO languages, middleware and patterns that deal with these types of aspects in an efficient, maintainable, and reusable manner.

Acknowledgments

This research is supported by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven.

References

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. [14], pages 135–173.

[2] Maarten Bynens and Wouter Joosen. Towards a pattern language for aspect-based design. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 13–15, New York, NY, USA, 2009. ACM.

[3] Maarten Bynens, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. The elementary pointcut pattern. In *BPAOSD'07: Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development*, pages 1–2, 2007.

[4] Siobhán Clarke and Robert J. Walker. Generic aspect-oriented design with Theme/UML. pages 425–458.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[6] Stephan Gudmundson and Gregor Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[7] William H. Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA*, pages 411–428, 1993.

[8] Kevin Hoffman and Patrick Eugster. Bridging java and aspectj through explicit join points. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 63–72, New York, NY, USA, 2007. ACM.

[9] Shmuel Katz. Aspect categories and classes of temporal properties. [14], pages 106–134.

[10] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis management systems: A case study for aspect-oriented modeling. Technical Report SOCS-TR-2009-3, School of Computer Science, McGill University, 2009. `http://www.cs.mcgill.ca/research/reports/2009/socs-tr-2009-3.pdf`.

[11] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[12] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 52–67, New York, NY, USA, 2002. ACM.

[13] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[14] Awais Rashid and Mehmet Aksit, editors. *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*. Springer, 2006.

[15] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/SIGSOFT FSE*, pages 166–175, 2005.

[16] Dimitri Van Landuyt, Eddy Truyen, and Wouter Joosen. Discovery of stable domain abstractions for reusable pointcut interfaces: common case study for ao modeling. Technical report, Department of Computer Science, K.U.Leuven, 2009. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW560.abs.html`.

A. Entire example

The full source code is available at `http://www.cs.kuleuven.be/~dimitri/callbackmismatch.zip`.

```
public abstract aspect IdentityValidation {

  pointcut personIdNeedsChecking(Person
      person, ValidationReport report):
    (execution(new(..)) || execution(void
        Person.set*(..))) && this(person) &&
        report(report);
  abstract pointcut report(ValidationReport
      report);

  Object around(Person person, ValidationReport
      report):personIdNeedsChecking(person, report){
    Object res = proceed(person, report);
    report.validation(TelecomOperator.validCaller(
        person.getFirstName() + " " +
        person.getLastName(),
        person.getPhone()));
    return res;
  }
}
```

Listing 2. IdentityValidation.aj

```
public interface Person {

  public String getFirstName();
  public String getLastName();
  public String getPhone();
  public void setFirstName(String fname);
  public void setLastName(String lname);
  public void setPhone(String phone);

}
```

Listing 3. Person.java

Views for Aspectualizing Component Models

Abdelhakim Hannousse *
Ascola Team, Ecole des Mines de Nantes, Inria, Lina
abdel-hakim.hannousse@emn.fr

Gilles Ardourel
Coloss Team, Université de Nantes, Lina CNRS UMR62441
Gilles.Ardourel@univ-nantes.fr

Rémi Douence
Ascola Team, Ecole des Mines de Nantes, Inria, Lina
Remi.Douence@emn.fr

Abstract

Component based software development (CBSD) and aspect-oriented software development (AOSD) are two complementary approaches. However, existing proposals for integrating aspects into component models are direct transposition of object-oriented AOSD techniques to components. In this article, we propose a new approach based on views. Our proposal introduces crosscutting components quite naturally and can be integrated into different component models.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures–*Languages*

General Terms Aspect-Oriented Software Development, Component Based Software Development.

Keywords Aspectualization, VIL, Views, Crosscutting wrappers

1. Introduction

Component based software development (CBSD) and aspect-oriented software development (AOSD) are two complementary approaches: while CBSD focuses on the modularity and the reusability of software systems by assembling components [10], AOSD focuses on the modularity of crosscutting concerns [7]. However, existing proposals for integrating aspects into component models are direct transposition of object-oriented AOSD techniques to components. Moreover, current proposals consider only specific component models and do not address the issue on its general form. Furthermore, most of them are unable to handle both integration and interaction of aspects. In this article, we contribute by proposing a new approach based on views. A view is defined as a reconfigured component architecture by introducing new composites encapsulating some of its original components. These new composites can then be wrapped to alter the behavior of their inner components. Views can be integrated into different component models. In this paper we show how views can be used for Fractal component model [3]. We also introduce a language for views, we call VIL, that makes integrating views and wrappers into a component architecture more expressive. However, integrating aspects following views consideration introduces crosscutting wrappers (i.e. crosscutting aspects). In this paper we highlight crosscutting wrappers issue and discuss the need of a formal specification of both components and wrappers behavior in order to detect and tackle their interaction issue.

The rest of this paper is organized as follows: section 2 describes a motivating example that we use to demonstrate how views are powerful enough to describe aspects. Section 3 introduces our language for views VIL. Section 4 shows how VIL can be integrated into Fractal component model. Section 5 discusses wrappers interactions and how VIL could contribute for detecting conflicting views. Section 6 reviews related work and section 7 concludes and discusses our key perspectives.

* Partially funded by the Miles Project

2. Motivating Example

In this section, we show with an example how views enable the integration of aspects into component architectures. Our example is a revised version of the one given in [2]. It describes a software controller of a crane that can lift and carry containers from arriving trucks to a buffer area or vice versa. The crane system is composed of an engine that moves the crane left to the truck and right to the buffer area, a mechanical arm that moves up and down and a magnet for latching and releasing containers by activating and deactivating its magnetic field. The engine and the arm may run in two different modes: *slow* and *fast*. Users interact with the crane using a control board. The control board allows users to choose a running mode for the crane and start crane loading or unloading containers. Figure 1 and figure 2 depict, respectively, a schematic overview and a possible component architecture of the crane system.

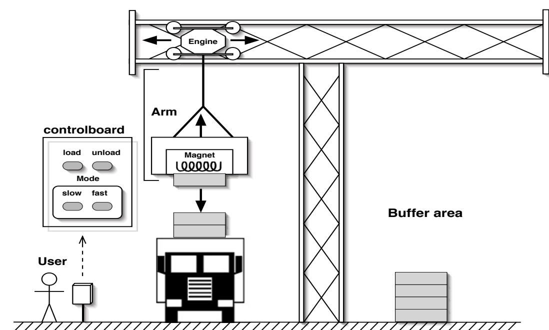

Figure 1. A Schematic Overview of the Crane System

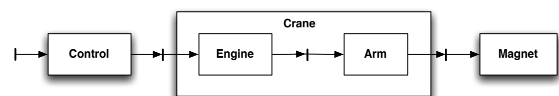

Figure 2. The Crane System Architecture

In Figure 2, components are depicted by rectangles and provided and required interfaces are represented by input and output arrows, respectively. Figure 2 models the crane system as a component architecture with three main components: *controller, crane* and *magnet*. The controller component provides an interface that permits to

set the running mode of the crane and start loading and unloading containers. Upon receipt of user commands, the control component transforms those commands into signals and requires the crane to act following those signals through its required interface. The crane component is a composite of the *engine* and the *arm* components. The engine component provides an interface that permits to move the crane left and right following a running mode and requires an interface to call the arm to move up and down. The arm, in turn, provides an interface for moving up and down following a running mode and requires an interface to ask the magnet to latch or release a container. Finally, the magnet component provides merely an interface for latching and releasing containers.

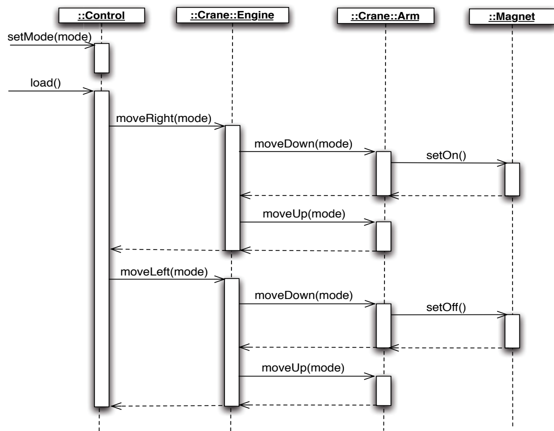

Figure 3. Loading Process for the Crane System

Figure 3 shows the UML sequence diagram of loading a single container. The process of loading a container starts when the user sets the running mode for the crane and presses the load button on the control board. These two actions are transformed into calling *setMode* and *load* operations, respectively, on the provided interface of the control component. When the control component receives a load call, it requires the engine component to move right by calling *moveRight* operation on its required interface. Upon receipt of *moveRight* call, the engine does the action and requires the arm to move down by calling *moveDown* operation. The arm accepts the call, moves down and asks the magnet to latch a container from the buffer area by calling *setOn* action on the arm. When the container is latched, the engine calls the arm to move up throwing a *moveUp* call. When all this done, the control requires the engine to move left to the truck by calling *moveLeft* operation. The engine receives the call, asks the arm to move down which in turn asks the magnet to release the latched container by calling *setOff* action.

2.1 An Optimized Crane System

Now we want to enhance the functionality of the crane system by forcing it to fulfill the following constraints:

C1 *When the arm is not carrying a container, the crane should run in fast mode.*

C2 *When the crane is loading a container on the truck, the arm should move down slowly.*

It is obvious that running the crane in fast mode when the arm is not carrying a container enhances the performance of the crane. Moreover, moving the arm slowly when it is carrying a container to be released on the truck ensures the safety of the truck. We call the above constraints *performance* and *safety* constraints, respectively.

In the following, we show how views can be used in order to force the crane system to fulfill the above constraints.

In this article, we use the term *view* to refer to a component architecture with additional composites encapsulating some its original components. We also use the term *wrapper* to refer to each entity that surrounds a component, intercepts calls on its provided and required interfaces and may alter its behavior.

Views implementation differs from one component model to another. As an example, a view in Fractal component model can be implemented as a controller associated to a composite that acts when calls are intercepted on its interfaces.

2.1.1 Fulfilling Performance Constraint

The crane system can be forced to fulfill the performance constraint by adding a wrapper around the engine and the arm components. The added wrapper intercepts calls on the provided interfaces of the engine and required interfaces of the arm. The wrapper stores and updates the state of the magnet whenever *setOn* and *setOff* operations are called. Thus, whenever the wrapper intercepts *moveLeft* and *moveRight* calls, it first checks the stored state of the magnet; if the state of the magnet is off it forces the engine to run in fast mode by proceeding the intercepted call with fast as a value of its parameter.

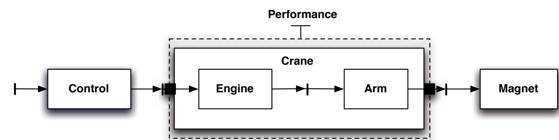

Figure 4. Performance View

Since the engine and the arm already belong to the same composite, the performance wrapper can be integrated at the crane composite level, which gives the first view of the crane system as shown in figure 4. This view is equivalent to the basic architecture with the exception of adding a wrapper to a composite level. The wrapper in figure 4 is presented as dashed border rectangle around the crane component. The small dark squares in the figure indicate the intercepted interfaces. We use the same notation for all the wrappers described in this paper.

2.1.2 Fulfilling Truck Safety Constraint

Considering truck safety in the crane system can be made by integrating a wrapper around the control and the engine components. This way, the integrated wrapper will intercept calls on provided interfaces of the control and required interfaces of the engine. The wrapper stores and updates the state on which the control is under loading or unloading a container. So that, whenever the second call of *moveDown* is intercepted, on the required interface of the engine, and the control is being loading a container it proceeds the *moveDown* call in slow mode.

In this case, we need another view of the component architecture of the crane where the control and the engine are encapsulated in the same composite. Figure 5 shows this required view.

Views make it simple to fulfill either constraints **C1** and **C2** shown above. However, when we consider both constraints, wrappers crosscut each other as shown in figure 6. It is obvious that the structure of the component system must be transformed in order to enable both wrappers at the same time. In the following, we introduce a specialized language for views definitions and show how it can be integrated with fractal in order to weave crosscutting wrappers.

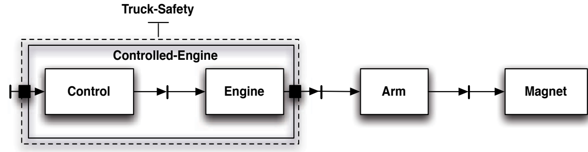

Figure 5. Truck Safety View

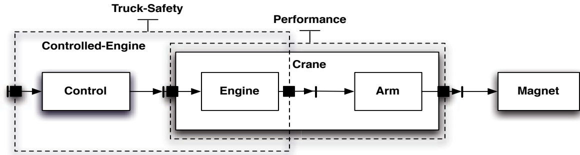

Figure 6. Wrappers Crosscut Phenomenon

3. VIL: Views Language

In this section we introduce a specialized language we call VIL for managing views in component models. Views can be specified using VIL to deal with the integration of wrappers into component architectures. We start by reviewing FPath language [5], used in VIL to access the required components which are going to be integrated into the same view.

3.1 FPath Query Language

FPath is a query language developed to deal with the introspection of the Fractal component architectures [5]. FPath uses declarative path expressions to introspect Fractal elements: components, interfaces and attributes.

$$engPath = \$root/child :: *[name(.) = crane]/child :: *[name(.) = engine]$$

For example, $engPath$ is an FPath expression that provides an access to the engine component in the architecture given by figure 2. This expression is divided into three steps separated by "/". The first step "$\$root$" indicates a value of an FPath variable to denote the component representing the root of the crane system. This later is considered as an input to the next step. The second step "$child :: *[name(.) = crane]$" takes the root component, denoted by the previous step, checks all its inner components "$child :: *$" and selects the one who has the name crane "$[name(.) = crane]$". The third step "$child :: *[name(.) = engine]$", which is similar to the second step, starts from the crane, denoted by the previous step, and provides an access to the engine component by checking all its inner components and selects the one who has the name engine. Similarly, $crnPath$ and $ctrPath$ provide accesses to the crane and the control components in the crane system architecture, respectively.

$$\begin{aligned} crnPath &= \$root/child :: *[name(.) = crane] \\ ctrPath &= \$root/child :: *[name(.) = control] \end{aligned}$$

3.2 VIL Language

Now we describe the views introduced in section 2.1 using VIL. As described in section 2.1.1, performance view wraps the crane component, intercepts all its provided and required interfaces. This can be expressed in VIL as follows:

$$V_1 = \textbf{view}\ crnPath$$

In VIL, the **view** keyword defines a view for a component architecture by wrapping the component described by $crnPath$ expression and intercepts all its provided and required interfaces.

Besides **view** keyword, **req** and **prov** keywords are used to define views by wrapping a component and intercept all its required and provided interfaces, respectively. Moreover, a wrapper may be interested to intercept calls on only some interfaces of a component, in this case, we use the "c **except** s" expression to indicate that the corresponding wrapper intercepts all the interfaces of the component c except those defined in s where s is a set of interface names.

In the case where the components that are going to be wrapped do not belong originally to the same composite, different sub-views should be defined each of which wraps one component and intercepts only its concerned interfaces. For example, in the truck safety case, the control and the engine components do not belong to the same composite; so, we need to define two sub-views, one to wrap the control and intercept all its provided interfaces and a second to wrap the engine and intercept all its required interfaces. These two sub-views can be defined in VIL as "**prov** $ctrPath$" and "**req** $engPath$" respectively. The complete view can be defined by composing sub-views using predefined views composition operators. For truck safety case, the two above sub-views can be composed using the "$\sqcup$" (i.e. *union*) operator. The result view describes the act of introducing a composite that wraps all the components defined by all its sub-views and intercepts all the interfaces intercepted by all its sub-views. The following is the complete VIL expression describing the truck safety view:

$$V_2 = \textbf{prov}\ ctrPath \sqcup \textbf{req}\ engPath$$

Besides "$\sqcup$" operator, "$\sqcap$" and "$-$" operators are used to describe intersection and difference operations on views. These three operators are used to extend the scope of wrappers, to determine conflicts on wrappers and to separate the scope of one wrapper from another in views, respectively. These operators are inspired by those defined in set theory. The following is the complete syntax we propose for the VIL language:

$$\begin{aligned} v \in View \quad ::= &\quad \textbf{view}\ e \mid \textbf{req}\ e \mid \textbf{prov}\ e \mid v_1\ \textbf{except}\ s \\ \mid &\quad v_1 \sqcup v_2 \mid v_1 \sqcap v_2 \mid v_1 - v_2 \end{aligned}$$

VIL is portable, declarative and robust language. VIL is portable because it does not depend on a specific component model, it is an independent language which can be integrated into different component models. We will show later in the next section how VIL can be integrated into Fractal component model. VIL inherits its declarative property from the FPath language [5]. Moreover, views can be composed using a set of declarative operators which enable programmers to define new abstractions (such as *controlled-engine*) on component architectures. Finally, when a component architecture is reconfigured, some views definitions may remain valid. For example, adding a new component between the engine and the arm components on the architecture depicted in figure 6 does not alter neither the performance nor the truck-safety views. Of course, arbitrary modifications of component architectures may also break views.

4. VIL Mapping to Fractal

In this section, we show how VIL can be integrated into Fractal Component Model [3]. We suppose here that the reader is familiar with Julia implementation of Fractal and Fractal-ADL. Fractal uses an Architecture Description Language (ADL) to describe component architectures. It supports hierarchies, introspection and com-

ponent sharing. We distinguish two cases for views mapping: the first case is when the components to be wrapped are directly related to each other and already belong to the same composite. Here we need just to associate a controller to that composite in order to intercept its interfaces and implement the wrapper behavior.

The second and more interesting case is when the components to be wrapped do not belong to the same composite or their are not directly related to each other. In this case view mapping is divided into two steps. The first step consists in finding the closest common parent of the components to be wrapped. This can be done using FPath language: Consider c_1 and c_2 two different components, the following FPath expression provides a set of all their common parents including the root component:

$$e = c_1/ancestor :: *[in(c_2/ancestor :: *)]$$

The "$c_1/ancestor :: *$" sub-expression returns the set of all the ancestors of c_1 including the root component. With the predicate "in" presented between square brackets, only the ancestors of c_1 that belong to the set of ancestors of c_2 will be returned. The closest parent c belongs to that set and has the following particularity: $descendant(c) \cap e = \phi$ which means that the descendants of the closest parent do not belong to the set returned by e.

The second step consists in adding a new composite as an inner component of the common parent of c_1 and c_2 found by the previous step. The new composite declares c_1 and c_2 as its inner components sharing them with their common parent. This way, the original architecture is not affected by views integration. Integrating a view means associating a controller to each shared component. The added controller intercepts calls and route them to the nesting composite. Figure 7 shows how the performance and truck safety views are integrated into the Fractal component architecture of the crane system. In this figure, the performance view is integrated following the first case and truck safety view is integrated following the second case. The component architecture transformation becomes a tedious and error prone task when the architecture grows. Our approach makes it possible to automatize this task.

Figure 7. Views in Fractal Component Model

Figure 8 shows the equivalent Fractal-ADL code of the architecture given in Figure 7. The underlined lines of code is the ones that can be generated automatically as a result of analyzing the following VIL expression that describes truck safety view:

$$V_3 = \textbf{prov}\ (\$root/control) \sqcup \textbf{req}\ (\$root/child :: crane/child :: engine)$$

5. Wrappers Interactions

We have shown how C1 and C2 constraints can be satisfied by introducing wrappers. We have also shown how both wrappers implementing C1 and C2 can be introduced at the same time in an automatically transformed architecture. In this case, the intercepted

```
<component name=root>
  <component name=control>
    ...........
    <controller name=prov>
  </component>
  <component name=crane>
    <component name=engine>
      ...........
    <controller name=req>
    </component>
    ...........
    <component name=arm>
      ...........
    </component>
    <binding ...........>
    <controller name = performance>
  </component>
  <component name=controlled-engine>
    <component name=control definition=/control>
    <component name=engine definition=/crane/engine>
    <controller name=truck-safety>
  </component>
  <binding ......>
</component>
```

Figure 8. Views Integration into Fractal-ADL

interfaces by both wrappers are disjoint and they are not in conflict with each other. However, this is not a general rule. So, we cannot consider that two wrappers are not in conflict just because they do not intercept common interfaces. As counterexample, let us consider the following *saving energy* constraint:

C3 *After carrying a thousand of containers in a day the arm should run in slow mode.*

Saving energy view requires the control and the engine components to be in the same composite. This time, the wrapper intercepts the provided interfaces of the control and the required interfaces of the engine. When load and unload calls are intercepted, the wrapper updates the number of carried containers. When the threshold is reached, the wrapper forces all the subsequent calls of moving the arm up and down to be in slow mode.

Consider the intercepted calls by the wrappers implementing C2 and C3, respectively. They are not disjoint, but when the wrappers intercept common calls (i.e. *moveUp* and *moveDown*) they agree to run them in slow mode. Indeed, C2 forces the arm to move down slowly in some cases (i.e. when it is loading the truck) and C3 forces the arm to move up and down slowly in some cases (i.e. when the threshold is reached). So, when the wrappers implementing C2 and C3 are applied at the same time, both constraints are once again satisfied.

Now consider the case of C1 and C3. The intercepted calls by their wrappers are disjoint. However, when both wrappers are applied at the same time, both constraints are not satisfied. In fact, both constraints can not be satisfied. Indeed, while the performance view forces the crane (and the arm) to run in fast mode when the arm is empty, the saving energy view forces the arm to run in slow mode once the threshold number of carried containers is reached. The exact behavior at run time depends on the implementation. Possible outcomes are:

- only one constraint is satisfied, because the first wrapper to be applied overrides the second one
- only one constraint is satisfied, because the second wrapper to be applied overrides the first one
- or worse, none of the two constraints is satisfied, because the implementation interleaves wrappers code.

Unsurprisingly, these conflicts are similar to aspect interactions. We believe that a support for conflicts detection and resolution is mandatory for aspectualizing component models. It is simple in VIL to detect views intersections. But as we have seen, this information is not sufficient in general to detect conflicts. Related work on aspect interactions [12] is a good starting point for future study. We also believe that component models offer properties such as protocols or contracts that could help in conflict detection. Finally, the notion of views could also help to specify what a conflict is and how it can be solved. For instance if a wrapper introduces transactions, we could specify that nested wrappers (*i.e.* nested transactions) are not allowed, or we could also declare that it is allowed to automatically extend the scope of a wrapper (*i.e.* it wraps more components) in order to expand the corresponding transaction.

6. Related Work

Many works are dedicated to aspectualize component models. However, most of them are interested in a specific component model and all of those works have failed to satisfy the two following requirements: (1) integrate aspects into component models in a natural way and (2) handle aspects interactions. In our opinion, their failure is due their lack of expressiveness as well as their lack of a formal model to analyze and verify properties on the result aspectualized architectures.

Some of the proposals to aspectualize component models (*e.g.*, FAC [8], FRACTAL-AOP [6], SAFRAN [4]) propose to extend component models with aspect-oriented concepts. Others (*e.g.*, FuseJ [9] and CaesarJ [1]) introduce new component models. To the best of our knowledge, all of them directly transpose object-oriented AOP concepts into existing CBSE. In particular, they rely on AspectJ-like pointcut expressions to define where aspects weave components. Our approach relies on alternative views to get rid of the tyranny of the primary decomposition and naturally introduces crosscutting at the level of components.

In all models but JAsCo, aspects are components. Currently in our proposal a wrapper is not always a component. When an aspect is a component, this promotes aspects reuse and enable to consider aspects of aspects. It should be studied how our approach can be extended in order to consider aspects of aspects. In the other hand, no aspectualized component model but JAsCo, proposes conflict detection support (beyond AspectJ-like detection of overlapping crosscut). JAsCo offers an API dedicated to compose aspects in a programmatic way. Our approach introduces crosscutting at the component level and could help to study interaction (*e.g.*; detect when two wrappers intersect, or when a wrapper is nested into another).

Unlike AspectJ-like pointcut expressions [7], VIL expressions are declarative and AspectJ pointcuts are imperative. This can be shown through the ability of VIL expressions to specify a pointcut for different joinpoints without so much care about the actions to be executed for each joinpoint. In the case of AspectJ, pointcuts and advices are strongly related. Moreover, VIL expressions are not used only to specify joinpoints but also to reconfigure component architectures in a way that wrappers can be integrated at the right positions. Our proposal can also be compared with Composition Filter model (CF) [2, 11] in the sense that each wrapper can be shown as an interface layer with input and output filters surrounding a component. However, views address more general concerns than those specified as filters. Moreover, according to the CF model presented in [11], filters can only be associated to only one component where a wrapper may alter more than one component. Furthermore, even if filters can be generalized to wrap many components it will be difficult to those filters to wrap components at different levels of hierarchies and share states on those components.

7. Conclusion and Future Work

In this paper we proposed VIL. A specialized language for aspectualizing component models. It relies on the concept of views that alter the basic component architecture by introducing new composite components. These extra composites can then be wrapped in order to intercept their interfaces and alter their basic behaviors for satisfy extra constraints. We have proposed a declarative language to define views. Our language do not rely on a specific component model. We have shown how to implement VIL in Fractal component model. Finally, we have discussed views interactions. Indeed, several views may share components and interact at common intercepted interfaces. This may lead to a conflict between views and violation of their satisfied constraints. However, views that do not share components may also interact. As future work, we are interested in providing a mechanism for conflicts detection and resolution. For conflict detection, both components and views behaviors should be considered. Each view should be associated with one or more constrains, then the compatibility of constraints associated to each pair of views should be checked to see whether or not they are in conflict with each other. For conflict resolution many strategies can be considered. We can mention as examples: associate priorities to views and define rules for views applications (e.g. when *v1* is applied *v2* cannot be applied).

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I (TAOSD I)*, vol. 3880 of LNCS, pages 135-173. Springer, 2006.

[2] L. Bergmans and M. Akşit. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Computing*, 36(1): 32-52, 1996.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software- Practice and Experience*, 36(11-12):1257-1284, 2006.

[4] P. C. David and T. Ledoux. Towards a Framework for self-adaptive component-based applications. In *Distributed Applications and Interoperable Systems*, vol. 2893 of LNCS, pages 1-14. Springer, 2003.

[5] P. C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45-63, 2009.

[6] H. Fakih, N. Bouraqadi, and L. Duchien. Aspects and Software Components: A case study of the Fractal Component Model. In *Proceedings of the International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, 2004.

[7] G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE'05)*, pages 49-58. ACM, 2005.

[8] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A Component-based and Aspect-oriented model for software evolution. *International Journal of Computer Applications in Technology*, 31(1/2):94-105, 2008.

[9] D. Suvée, B. D. Fraine, and W. Vanderperren. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In *Component-Based Software Engineering (CBSE)*, vol. 4063 of LNCS, pages 114-122. Springer, 2006.

[10] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-Oriented Programming. *Component Software Series*. ACM Press and Addison-Wesley, 2nd edition, 2002.

[11] L. Bergmans and M. Akşit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol. 44, No. 10, pp. 51-57, October 2001.

[12] R. Douence, P. Fradet, and M. Sudhot. A framework for the detection and the resolution of aspect interaction. In *GPCE'06: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative programming and component engineering*, pages 173-188, Springer-Verlag, 2002.

AspectKE*: Security Aspects with Program Analysis for Distributed Systems

Fan Yang
DTU Informatics,Technical University of Denmark
fy@imm.dtu.dk

Hidehiko Masuhara
Graduate School of Arts and Sciences,University of Tokyo
masuhara@acm.org

Tomoyuki Aotani
Graduate School of Arts and Sciences,University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Flemming Nielson
DTU Informatics,Technical University of Denmark
nielson@imm.dtu.dk

Hanne Riis Nielson
DTU Informatics,Technical University of Denmark
riis@imm.dtu.dk

Abstract

Enforcing security policies to distributed systems is difficult, in particular, when a system contains untrusted components. We designed AspectKE*, a distributed AOP language based on a tuple space, to tackle this issue. In AspectKE*, aspects can enforce access control policies that depend on future behavior of running processes. One of the key language features is the predicates and functions that extract results of static program analysis, which are useful for defining security aspects that have to know about future behavior of a program. AspectKE* also provides a novel variable binding mechanism for pointcuts, so that pointcuts can uniformly specify join points based on both static and dynamic information about the program. Our implementation strategy performs fundamental static analysis at load-time, so as to retain runtime overheads minimal. We implemented a compiler for AspectKE*, and demonstrate usefulness of AspectKE* through a security aspect for a distributed chat system.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; D.4.6 [*Security and Protection*]: Access controls; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Design, Languages, Security

Keywords Aspect Oriented Programming, Program Analysis, Security Policies, Distributed Systems, Tuple Spaces

1. Introduction

Enforcing security policies to a distributed system is challenging, especially when trusted components of a system have to work with untrusted components. In such a case, we need to ensure that untrusted components do not break security policies of the system. A common approach is to statically check security properties of the untrusted components before their execution [7, 9]. For example, Java type checks downloaded code before execution.

The approach has two problems. The first is lack of flexibility: the programmers cannot easily (re)define security policies, as they are normally integrated with a compiler and runtime system of the language. The second is expressiveness: static analyses are sometimes too restrictive to accurately enforce security policies in practice, as they have to approximate properties of a program, and cannot be combined with runtime information.

In order to address those problems, we designed and implemented AspectKE*, an aspect-oriented programming (AOP) language based on a tuple space system. AspectKE* has the following key characteristics.

- It provides high-level program analysis predicates and functions that can be used as pointcuts in aspects. Since those predicates and functions give information on future behavior of processes, the programmers can easily apply aspects (e.g., security aspects) to processes that are defined by untrusted parties.
- It also provides a novel variable binding mechanism for pointcuts, so that the programmers can specify static and dynamic conditions in a uniform manner.
- Its implementation strategy realizes runtime evaluation of program analysis predicates and functions with minimal runtime overheads, which is achieved by analyzing the required static information beforehand at the load-time, and merely looking it up at runtime.
- It is the first AOP system that is based on a tuple space. Even though tuple space based systems are not predominant in the industry, we believe that our techniques can be applied to other distributed systems as well.

The rest of this paper is organized as follows. Section 2 describes the problems that we address. Section 3 outlines our design principles for solving the problem. Section 4 proposes our AOP language. Section 5 shows our solution to the problems in Section 2. Section 6 sketches implementation issues. Sections 7 discusses the related work and Section 8 concludes the paper.

2. Motivating Problem

Imagine a company that runs a chat system for exchanging messages among its employees. In order for the employees to access the system from outside the company, the chat system allows client programs (a third-party software) to be executed on untrusted computers. Now the challenge is how to ensure secrecy and integrity of data exchanged between company and employees, especially when we cannot trust client processes running on a computer with lower-security.

First, let us see a chat system without any security mechanism. Figure 1 illustrates a simplified distributed chat system that consists of six nodes. The nodes ALICE and BOB represent two users' chat functionalities inside the chat system (trusted). The nodes GUI1 and GUI2 represent the users who use the chat system (trusted). The nodes CLIENT1 and CLIENT2 represent third-party software

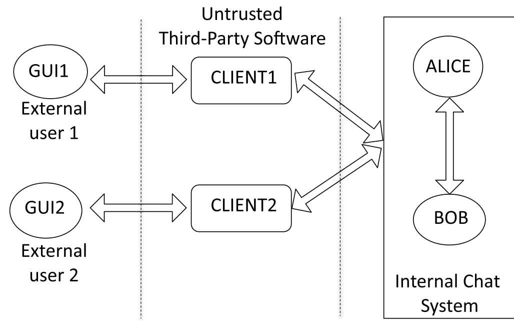

Figure 1. An Overview of a Simplified Chat System

running on untrusted computers that relay messages between users and chat function nodes (untrusted).

Let us focus on CLIENT1 and CLIENT2, as they are the only untrusted parts. Besides performing intended operations, a third-party client might contain malicious code that performs unintended operations. Listings 1 and 2 show a code fragment of node CLIENT1, written in AspectKE*, implementing a user login procedure. Line 8 of Listing 2 is added to the original client definition so that it will leak password information to an eavesdropper.

```
1  node CLIENT1{
2      process clientlogin(CLIENT1,GUI1);
3  }
```

Listing 1. Node CLIENT1

```
1  proc clientlogin(location self,location gui){
2      location user;
3      symbol password;
4
5      in(OUTPUTG,LOGIN,user,password)@gui;
6      out(INPUTU,LOGIN,password,self)@user;
7
8      out(LOGIN,user,password)@EAVESDROPPER;
9
10     in(OUTPUTU,LOGIN,user)@self;
11     out(INPUTG,LOGIN,user,SUCCESS)@gui;
12
13     parallel{
14         process clientsendmsg(self,user,gui);
15         process clientreceivemsg(self,user,gui);
16     }
17 }
```

Listing 2. Process clientlogin

Listing 1 defines the node CLIENT1, which instantiates a process clientlogin, with CLIENT1 and GUI1 as parameters. Constants are capitalized in this paper, whose declarations are omitted in this paper.

Listing 2 defines process clientlogin. Lines 2-3 define local variables. Line 5 waits for an input of a user name and a password information from a user. For example, when Alice (using GUI1) inputs a login request, Line 5 binds the variable user to the user name (i.e., ALICE), and the variable password to the password typed in (e.g., ALICEPW). Line 6 sends the password information to the corresponding user node at the server computer by referencing the two variables. A process (omitted here) at the chat function node will send a confirmation message if the password is correct. Lines 10-11 receive the confirmation message and notify GUI1 of successful login. Lines 13-15 start processes for handling messages between users (details of this step are not discussed in the paper).

The definition of CLIENT1 except for Line 8 is intended; i.e., it performs no malicious operations. The operation at Line 8 is additional malicious code that sends user and password to node EAVESDROPPER.

To ensure the secrecy and integrity of users' data which pass through untrusted components, we pose the following security policy: CLIENT1 is allowed to get data from GUI1 only when the obtained data is sent to the specified trusted nodes. For the program above, the policy means that the input action at Line 5 is permitted only if its continuation process does not send password to any node other than user. This security policy essentially demands to perform static analysis of the continuation process (Lines 6-17) before actually performing the input action (Line 5). In this paper, we show how to integrate the static analysis techniques into a security aspect with minimal runtime overheads.

3. Design Principles

3.1 Static Analysis for Security Aspect

Some security policies need information on future events. An example is the security policy mentioned in Section 2, where we cannot decide whether to permit an input action performed by an untrusted process without inspecting how the password information will be used in the future. In this paper, we integrate static analysis techniques into aspect definition, and provide an expressive way of specifying security aspects that refer to future events.

3.2 Program Analysis Predicates and Functions

The language for composing security aspects should have comprehensive interface for using static analysis techniques. We provide several high-level program analysis predicates and functions that extract static analysis results of a program, so that the users can easily specify security policies in aspects. In addition, our novel variable binding mechanism for pointcuts enables the programmers to specify static and dynamic conditions in a uniform manner.

3.3 Load-Time Static Analysis

We perform static analysis at load-time because it fits a distributed setting and can retain runtime overheads minimal as well. In principle, static analysis can be performed at either compile time, load-time, or run-time. However, compile-time analysis requires the definition of processes which is not realistic in a distributed system with mobile processes. Run-time analysis is not feasible either as it causes huge runtime overheads.

4. AspectKE*

We designed and implemented AspectKE* programming language, an aspect extension to the Klava tuple space system[2]. Since Klava is a distributed tuple space system (DTS), we briefly introduce basic concept in DTS.

A DTS consists of *nodes*, *processes*, *tuple spaces* and *tuples*. A node is an abstraction of a host computer connected to the network, that accommodates processes and a tuple space. A tuple space is a repository of tuples that can be accessed concurrently from processes. A process is a thread of execution that can output (through out action) its data as a tuple to a tuple space, and can retrieve (through read or in action) data from a tuple space by matching a pattern. Unlike classical tuple space systems such as Linda [5] that assume a globally shared tuple space, a DTS contains shared tuple spaces distributed over a network. Besides the standard actions about retrieving and outputting tuples on a local or remote nodes, a Klava process can also create new processes on a local or remote

node (through eval action), can create a new remote node (through newloc action).

In AspectKE*, aspects are global activities that monitor actions performed by all processes in a Klava system.

4.1 The Hello World Example

Listing 3 shows a Hello World program that demonstrates the basic usage of nodes and processes. In the program, a process at node N1 reads HELLO and WORLD from its own tuple space and create a process at node N2 that outputs these words in a different order.

```
1  location N1,N2;
2  symbol W1,W2,HELLO,WORLD;
3
4  node N1{
5      data (N2,W1,HELLO);
6      data (N2,W2,WORLD);
7      process p1(N2);
8  }
9
10 node N2{
11 }
12
13 proc p1(location baz){
14     symbol foo, bar;
15     read(baz,W1,foo)@N1;
16     in(baz,W2,bar)@N1;
17     eval(process p2(foo,bar,baz))@baz
18 }
19
20 proc p2(symbol foo,symbol bar,location baz){
21     out(foo,bar)@N2;
22     out(bar,foo)@baz;
23 }
```

Listing 3. Hello World Program

Lines 1 and 2 declare constants. The type *location* is a set of logical node locations. The type *symbol* is a set of globally distinguishable data. Lines 4-11 define initial states of node N1 and N2. Node N1 consists of two tuples and one process. Node N2 is empty. Lines 13-18 define a process p1. Line 14 declares two local variables, which are bound to values by an input action. For example, the tuple ⟨baz,W1,foo⟩ at Line 15 matches the tuple ⟨N2,W1,HELLO⟩ at node N1, and binds foo to HELLO. Line 16 performs an in action, which reads a tuple ⟨N2,W2,WORLD⟩ from N1 in a similar manner to read actions, and then removes the read tuple. Line 17 creates a process p2 with parameters HELLO, WORLD and N2 at node N2. The process p2 then executes two out actions that output HELLO and WORLD onto the node N2 in a different orders.

4.2 A Simple Aspect for Hello World

Listing 4 defines a simple aspect that prevents read actions in the Hello World program from executing. Note that in AspectKE*, all actions are joint points.

```
1  aspect a1(N1)
2  {  read(location VAR n, symbol VAR w,
3          symbol FORMAL word)@(N1)
4      -> process z;
5      { case(!w=W1) break;
6        case(!beused(word,z)) break;
7        case(forall(x,targeted(OUT,z))<x=n>) break;
8        default proceed;
9      }
10 }
```

Listing 4. A Simple Aspect

4.2.1 Pointcut

Lines 1-4 define a pointcut that captures a read action (which reads N1's tuple space) performed at node N1. Parameters of the pointcut specify types (either location or symbol) and kinds (either VAR or FORMAL). When the joint point (Line 15 in Listing 3) is to be executed, variables n and w are bound to values N2 and W1, respectively. The variable word, whose kind is *formal*, is bound to a variable foo in the process.

Note that a *formal* variable is bound to a variable in a process, unlike the binding mechanism of *var* variable and formal parameters of an advice declaration in AspectJ, which are bound to values. This idea is originally proposed in our previous work [6] in order to deal with open joinpoints that extensively occur in tuple space systems. We adopt this mechanism for specifying usage of variables in a process that is not yet bound to any value when an action is performed.

The description at Line 4 binds the variable z to a continuation process right after the captured action. When the pointcut matches the read action at Line 15 in Listing 3, z denotes actions performed by Lines 16-18 and 20-23.

In addition, our variable binding mechanism can internally link static information to each variable, thus enables programmers to specify static and dynamic conditions regarding the bound variables in a uniform manner.

4.2.2 Advice

Lines 5-9 define a piece of advice that terminates an executing process if one of the following three conditions holds. (1) Its second parameter is not equal to W1. (2) Its third parameter will not be used in the rest of the process. (3) All out actions in the rest of the process target at the location specified by the first parameter n. Each case statement consists of a condition and suggestion (*break* or *proceed*). If *break* is executed, the current process stops. If *proceed* is executed, the current process continues. When pointcut matches the join point as mentioned in Section 4.2.1, the action is terminated by the third case.

From the advice definition, it is obvious that the first case condition does not hold. The second case condition uses a *program analysis predicate* beused, which does not hold as well. In AspectKE*, *program analysis predicates (and functions)* are language constructs for aspects that predict future behavior of a program. Here we only explain them by examples, but their definition will be discussed in the next subsection. This beused predicate checks future behavior of an executing process, namely, whether variable foo (captured by word) is not referenced in any action of the continuation processes. Since it uses foo in the out actions at Line 21 and 22 of Listing 3, the beused predicate holds, which in turn makes the overall condition false. Note that the aspect has to check the condition before executing those out actions. This means that we need to analyze the future behavior of a program.

The third case is complicated, although the expression itself looks quite simple thanks to our novel binding mechanism. It checks whether the first argument is used as the destination of all out actions in the continuation process by a predicate forall and a *program analysis function* targeted.

All destination locations of out actions in the process z are collected and returned as a set by the function targeted(OUT,z). For example, if z contains two out actions out(...)@c and out(...)@v, targeted(OUT,z) returns a set {c,v}. Each element in the set is either a constant (e.g., N2 at Line 21 in Listing 3) or a variable name (e.g., baz at Line 22 in Listing 3).

Predicate & Function	*The Return Value*
performed(z)	returns the set of all actions that are performed in z
targeted(OUT,z)	returns the set of all destination locations of out actions in z
beused(foo,z)	returns true if variable foo is used in any actions of z
beused(foo,OUT,z)	returns true if variable foo is used in out actions of z
beusedsafe(foo,OUT,A,z)	returns true if variable foo either will not be used in out actions of z at all, or used in out actions of z, but only be performed to locations within set A.

Table 1. Program Analysis Predicates and Functions

The predicate forall(x,A)<x=n> holds when all the elements in A is equal to n. Note that equality is checked in a different ways depending on what x denotes. If x denotes a concrete value, x=n is true when n equals to the value. If x denotes a variable v, x=n is true when v will be bound to n if proceeded. When matching the joint point at Line 15 in Listing 3, A is a set that contains the constant N2 and the variable baz whose runtime value is also N2, which in turn lets the pointcut binds n to N2 as well. Thus forall(x,A)<x=n> is true according to the definition of the forall predicate and equality. The aspect will then suggests *break* in order to terminate the read action. Since both runtime data and static information are needed to evaluate this condition, it goes beyond a static property of n. It also shows that the programmers can specify static and dynamic conditions of n in a uniform manner.

4.3 Program Analysis Predicates and Functions

Table 1 summarizes the program analysis predicates and functions in AspectKE*, where foo is a bound formal variable; OUT is a type of actions (can be replaced with other types of actions); z is a the continuation process of the captured action; and A is a collection of locations which includes *locations* in two forms: constants and bound formal (or var) variables. These predicates and functions are designed to specify different properties of the continuation program.

5. A Security Aspect for the Distributed Chat System

Listing 5 presents a security aspect with program analysis predicates to enforce the security policy presented in Section 2: the input action at Line 5 in Listing 2 is permitted only if its continuation process will never output password to any node other than user.

```
1  aspect in_loginpw(location VAR s){
2     in(OUTPUTG,LOGIN, location FORMAL uid,
3       symbol FORMAL pw)@(location VAR gui)
4       -> process z;
5     { case(element_of(gui,{GUI1,GUI2})&&
6              !beusedsafe(pw,OUT,{uid},z))
7              break;
8       default
9              proceed;
10    }
11 }
```

Listing 5. Aspect for Protecting Password Usage

Upon the joint point at Line 5 in Listing 2, the var variables s and gui are bound to CLIENT1 and GUI1, respectively; the formal variables uid and pw are bound to the variable user and password, respectively. At Line 5 in Listing 5, the predicate element_of returns true since GUI1 is in the set {GUI1,GUI2}. At Line 6, the program analysis predicate beusedsafe checks if the continuation process z outputs password only to location user. Since the underlying static analysis detects that password is output to EAVESDROPPER, this predicate returns false. Thus the overall suggestion from the advice is to *break*, which results in termination of the malicious client.

6. Implementation Issues

Our AOP system consists of a translator from AspectKE* to Java and a runtime system that supports tuple space and AOP operations. The translator translates a source program in AspectKE* into a Java program that exploits distributed operations in a runtime library. The runtime system matches and executes aspects dynamically so that new security policies can be applied to a running system.

In order to efficiently evaluate a program analysis predicate and function in an aspect, our system performs context insensitive interprocedural dataflow analyses when it dynamically loads process definitions. The analyzer takes process definitions at the Java bytecode level, in order to apply aspects to a system without source code, which is the common approach in the distributed mobile processes.

Figure 2 shows how the program analysis predicates and functions work with advice. The runtime system matches each action with pointcut descriptions. When matches, it evaluates a program analysis predicate (or function) by looking up the result performed at load-time. We developed a mapping mechanism that associates bound variables in aspects to static information of the bound value from analysis results. For example, when evaluating the beused(word,z) predicate at Line 6 of Listing 4, it picks up the relevant static analysis results (variable foo at Line 15 links with variables foo appeared in Line 21 and 22 of Listing 3) to evaluate whether word (mapping to foo at Line 15 of Listing 3) is used in z (the continuation process of Line 15 in Listing 3).

Regarding the performance, since the load-time static analysis does not incur much runtime cost, we believe that our approach is practical and can be used in other AOP systems that need to check the future behavior of programs.

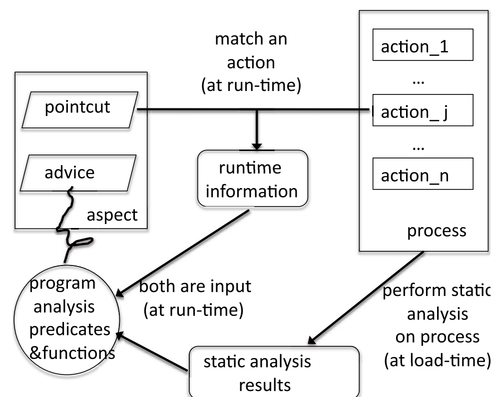

Figure 2. Evaluation of Predicates and Functions

7. Related Work

There are several tuple space systems that provide a certain security mechanism. For example, KLAIM[10] (with Klava[2] as its implementation) uses a static type system to realize access control. SecOS[12] provides a low-level security mechanism that protects every tuple field with a lock. JavaSpaces[4], which is used

in industrial contexts, has a security mechanism based on the Java security framework. Our work is different from these in combining aspects with program analysis techniques, hence provides more flexible and precise ways to specify and enforce security policies.

There are several AOP systems in which pointcuts can specify relationships between join points. AspectJ's cflow pointcut captures join points based on a control flow in a program, which can be used for implementing access control mechanisms. Dataflow pointcut [8] identifies joint points based on flow of the information, which can be used for enforcement of secrecy and integrity. However, both pointcuts capture control or data flow that have happened before, rather than in the future. Some advanced AOP languages [1, 3, 11] allow the programmers to define their own pointcut primitives, including those that exploit program analysis results. In theory, it is also possible for those languages to define security aspects based on the future behavior of a program by defining pointcuts that statically analyze the program. However, those languages offer accesses to the programs at bytecode or AST-level, which makes it hard to implement correct and efficient static analyses.

Our approach, in contrast, provides predicates and functions that give relatively high-level information about future behavior, which makes it much easier to implement security aspects. Additionally, due to the novel binding mechanism of variables in pointcuts, our language is more expressive for specifying analysis properties.

8. Conclusions

We designed and implemented a prototype of AspectKE* that can retrofit existing, or even running distributed systems by applying security aspects. As an AOP system, our contributions can be summarized as follows. (1) AspectKE* can straightforwardly express a large set of security policies, especially those based on future behavior of executing processes. (2) The high-level program analysis predicates and functions allow the programmers to directly specify security policies without defining complicated program analysis. (3) The novel variable binding mechanism for pointcuts enables aspects to express dynamic properties of an executing process in combination with static properties derived by the static analysis predicates and functions. (4) We proposed an efficient implementation strategy that combines load-time static analysis and runtime checking, so as to keep runtime overheads minimal while keeping expressiveness of aspects.

Current AspectKE* can merely monitor processes and command the processes to *break* or *proceed* from its advice. We plan to extend the language so that it can perform other kind of actions. The challenge is how to formulate static analysis as aspects can introduce extra data- and control-flows into processes that should also be monitored by static analysis predicates and functions.

Acknowledgments

This work is partly supported by the Danish Strategic Research Council (project 2106-06-0028) "Aspects of Security for Citizens". We would like to thank Lorenzo Bettini for discussing about the Klava system, Christian Probst, Hubert Baumeister, and Sebastian Nanz for their early comments, and the members of the PPP group at the University of Tokyo, Robert Hirschfeld and his research group members for their comments on the work.

References

[1] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD'07*, pages 161–172. ACM, 2007. ISBN 1-59593-615-7.

[2] L. Bettini, R. D. Nicola, and R. Pugliese. Klava: a Java package for distributed and mobile applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[3] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. In *AOSD'04*, pages 102–111. ACM, 2004.

[4] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces principles, patterns, and practice*. Addison-Wesley Longman Ltd. Essex, UK, UK, 1999.

[5] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. ISSN 0164-0925.

[6] C. Hankin, F. Nielson, H. R. Nielson, and F. Yang. Advice for coordination. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 153–168. Springer, 2008.

[7] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification*. Addison-Wesley Professional, 1999.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS'03*, volume 2895 of *LNCS*, pages 105–121. Springer, 2003.

[9] G. C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119. ACM, 1997.

[10] R. D. Nicola, G. L. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[11] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05*, volume 3586 of *LNCS*, pages 214–240. Springer, 2005.

[12] J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Sci. Comput. Program.*, 46(1-2):163–193, 2003.

Preserving Dynamic Reconfiguration Consistency in Aspect Oriented Middleware

Bholanathsingh Surajbali, Paul Grace and Geoff Coulson

Computing Department
Lancaster University
Lancaster, UK
{b.surajbali, p.grace, geoff} @comp.lancs.ac.uk

Abstract

Aspect-oriented middleware is a promising technology for the realisation of dynamic reconfiguration in heterogeneous distributed systems. However, like other dynamic reconfiguration approaches, AO-middleware-based reconfiguration requires that the consistency of the system is maintained across reconfigurations. AO-middleware-based reconfiguration is an ongoing research topic and several consistency approaches have been proposed. However, most of these approaches tend to be targeted at specific contexts, whereas for distributed systems it is crucial to cover a wide range of operating conditions. In this paper we propose an approach that offers distributed, dynamic reconfiguration in a consistent manner, and features a flexible framework-based consistency management approach to cover a wide range of operating conditions. We evaluate our approach by investigating the configurability and transparency of our approach and also quantify the performance overheads of the associated consistency mechanisms.

Categories and Subject Descriptors D.2.7 11 [**Software Engineering**]: Distribution, Maintenance, and Enhancement.

General Terms: Algorithms, Design, Management.

Keywords: middleware; reflection; aspects; dynamic reconfiguration; consistency.

1. Introduction

A key and growing challenge for distributed systems is their need to support *dynamic reconfiguration* in order to maintain optimal levels of service in diverse and changing environments. In response to this challenge, *aspect-oriented middleware* [10, 12, 13, 14, 16, 19] has recently emerged as a promising basis on which to build reconfigurable distributed systems. The core concept of AO middleware is that of an *aspect*: a module that deals with one specific concern and can be changed independently of other modules. Aspects are made up of individual code elements that implement the concern (*advices*). Advices are deployed at multiple positions in a system (*join points*) which are expressed by *pointcuts*—a particular form of composition language.

Dynamic reconfiguration of distributed systems requires assurances that the reconfiguration does not leave the system in an inconsistent state that can potentially lead to incorrect execution or even complete system failure. In AO middleware environments reconfiguration inconsistencies arise from a range of characteristic sources which we classify under two broad headings: *system environment related* sources and *composition related* sources. System environment related inconsistencies occur due to the runtime system environment (e.g. message loss or node crash); whereas composition related inconsistencies refer to application-specific semantic relationships between modules or aspects (e.g. if one aspect is dependent on another than removing the first will result in inconsistency; or if two aspects are mutually exclusive then deploying both simultaneously will result in inconsistency).

In general, avoiding these sources of inconsistency is a difficult task due to the diversity of distributed applications (e.g. centralised/decentralised, static/mobile, small scale/large scale etc) and also because of diverse application-specific factors (e.g. varying dependability requirements, or varying trade-offs between consistency and scalability). Relying on the application developer to ensure the consistency of the system is not feasible under such heterogeneous conditions. Moreover, a *one-size-fits-all* approach to consistency management is not feasible either. Instead, *multiple* consistency strategies should be supported within a framework-based approach so that appropriate strategies can be applied to each set of arising circumstances.

Supporting multiple consistency strategies entails meeting the following key requirements:

- *Configurability*. It must be possible to configure and even reconfigure the consistency-related functionality of the system.
- *Transparency*. Managing reconfiguration across each node is a complex and error prone task for the application programmer. Achieving consistency must therefore involve minimum programmer effort.

To address the above issues and requirements we propose in this paper a *distributed consistency framework* that ensures consistent AO-based dynamic reconfiguration while being tailorable to specific conditions and environments.

The rest of the paper is organised as follows. Section 2 provides a detailed discussion of the various threats to consistency to which distributed applications are prone. In Section 3 we present necessary background on the AO composition technology on which we base our proposal (i.e. our AO-OpenCom platform). Section 4 then presents our distributed consistency framework, which is then evaluated in Section 5. Finally, Section 6 discusses related work, and we offer our conclusions in Section 7.

2. Threats to Consistency

To illustrate threats to consistency under dynamic reconfiguration in distributed systems we now present a simple case-study (see

figure 1) which comprises a multimedia peer to peer network in which heterogeneous peers share data files and interact among themselves. The peers (laptops, PCs, and PDAs) can also operate in different network domains (Internet, Wi-Fi, ad-hoc wireless networks, etc.). Given this environment a wide range of dynamic reconfiguration scenario are feasible. For example:

(i) when a new video codec become available we may want to encapsulate it as encoder and decoder aspects and dynamically deploy it on all nodes with video capabilities;

(ii) when nodes move from a fixed to a wireless network environment we may want to deploy fragmentation and reassembly of the video and audio media frames;

(iii) when application performance degrades at a given node we want to deploy a cache aspect while ensuring that cache consistency is maintained across nodes.

We now present important threats to the consistency of such reconfiguration scenarios. While we do not claim this to be an exhaustive list, we believe it to be strongly indicative of the challenges that must be addressed.

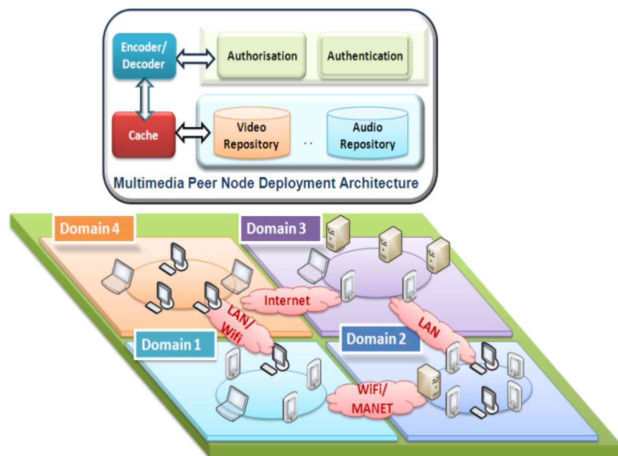

Figure 1. Multimedia application case study scenario

2.1 System environment threats

These relate to reconfiguration inconsistencies that occur due to the instability of the underlying distributed environment in which the reconfiguration takes place. The inherently unstable characteristics of the networks and nodes employed in the scenario increase the chances that a reconfiguration will be compromised. These threat include:

Protocol message disruptions. If reconfiguration-related messages are lost, re-ordered, duplicated or delayed, the consistency of the reconfiguration is clearly compromised. For example, as messages get lost, the initiating node (referred as the coordinator) of the reconfiguration can be mislead into waiting for the reconfiguration to complete.

Local node disruptions. The reconfiguration requests (i) to (iii) sent by the initiator of the reconfiguration may not reach some of the peer nodes. Even if the messaging is unproblematic, individual nodes may still fail to apply a requested reconfiguration. For example:

- the node may be overloaded or may crash;
- a aspect composition request may fail because of resource scarcity on the target node or because the node's local policy forbids it to make the requested change;
- modules or aspects may still be performing computations when an attempt is made to remove or recompose them.

Again, such factors can lead to parts of the intended reconfiguration not being carried out, and consequent inconsistency.

Infrastructure service failures. Aspects to be reconfigured into the system are typically stored in repositories which may get congested with requests, or crash, meaning that aspects may not be available to be deployed (or may perhaps be only deployable in parts of the system). Additionally, different repository instances may have different versions of the aspects: e.g. different versions of the encryption aspects may be produced over time, so that different nodes configure different codec versions and be inconsistent with one another.

Simultaneous reconfigurations. Different reconfiguration requests may arise simultaneously so that reconfiguration-related messages relating to distinct requests may be interleaved and potentially be received in different orders at different nodes. For example, one request might ask for a fragmentation aspect to be replaced, while another asks for it to be removed. There will clearly be different outcomes depending on the execution order of these two requests—and furthermore the outcomes might be different at different nodes.

Unauthorised nodes initiating reconfiguration. Reconfiguration messages may be spoofed by malicious nodes in an attempt to directly and deliberately compromise consistency.

2.2 Compositional threats

These relate to faulty interactions, following reconfiguration, between the newly-reconfigured entities and prior non-reconfigured entities. The associated threats typically involve conflicts and dependencies: conflicts are threats causing negative interactions between system entities; while a dependency threat relates to a 'required' relationship that needs to be associated with the reconfiguration for the system to operate correctly. The different compositional threats are:

Unsynchronised weaving of dependent aspects. Some aspects are inherently dependent on each other; for example, decryption is dependent on encryption, and a cache may be dependent on a remote cache manager. Therefore the order in which aspects are woven is crucial: e.g., we must ensure that an assembler aspect is put in place *before* its associated fragmenter, otherwise fragmented messages may be received which cannot be handled.

Unsynchronised binding of distributed aspects. Some distributed aspect systems employ 'remote aspects' which are used by several distributed client nodes. If such an aspect, e.g. a cache manager is removed without the consent or even the awareness of its client nodes, errors can arise when clients attempt to communicate with the aspect.

Mutual exclusion of aspects. Behavioural conflicts can occur as new aspects are woven. For example adding a logging aspect into our scenario at the same join points as an encryption aspect can result in behavioural conflicts, because the system is open to read the logged, decrypted messages.

3. The AO-OpenCom Framework

Before discussing our proposed distributed consistency framework, we briefly introduce the software composition technology that underlies our work. *AO-OpenCom* is an extension of the OpenCom component model [5] and provides a distributed AO composition service while allowing aspectual compositions to be dynamically reconfigured. An earlier version of AO-OpenCom was the subject of a prior workshop paper [16]. We revisit it here because the current version differs significantly from the earlier one in key areas.

3.1 Aspects and Aspect Composition.

Aspect composition in AO-OpenCom employs components to play the role of aspects—i.e. an aspect is simply an OpenCom component (hereafter we use the term *aspect-component* when referring to an OpenCom component that is playing the role of an aspect). Aspects are composed using so-called *AO-connectors*. These are specialised connectors that support the run-time insertion of aspect-components.

Internally, an instance of AO-OpenCom is structured as a set of per-node local instances, as illustrated in figure 2, which are combined into a multi-node AO-OpenCom distributed system. The *Distribution Framework* is a plug-in for the AO-OpenCom communication service that sends reconfiguration and management messages to every node in the system; the *ISend* interface provides a *send()* operation, while its *INotify* interface delivers received messages to the AO-OpenCom Configurator.

Turning now to the constituent components, the *Configurator* is responsible for accepting and handling reconfiguration requests from applications. It interacts with the Pointcut Evaluator and Advice Handler components on either the local node or other nodes to actually carry out the requested reconfiguration in terms of AO (re)compositions. The *Aspect Repository* holds a set of instantiable aspect-components. This is composed of a front-end proxy gateway component and a back-end database component. Finally, the *Pointcut Evaluator* evaluates pointcuts and returns a list of matching join points within the framework; and the *Aspect Handler* weaves advices at these join points in the framework.

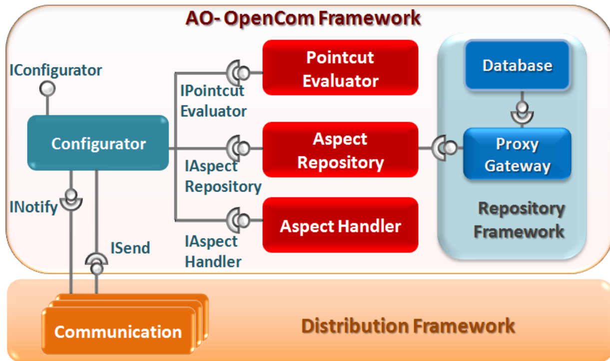

Figure 2. An AO-OpenCom per node instance

3.2 Reconfiguration in AO-OpenCom

The main API provided by an AO-OpenCom for dynamic reconfiguration takes the form of a single operation on the Configurator component:

Configurator.reconfigure(target_dcf, pc, command, aspect, scope, locus).

The *target_dcf* argument specifies which distributed system the reconfiguration should be applied to. The *pc* argument specifies a pointcut that picks out the join points at which the desired reconfiguration should occur. The *command* argument offers options—either 'add', 'remove', or 'replace' an aspect—for the action to be taken at the indentified join points. The *aspect* argument can be a direct reference to a local aspect-component, or an indirect reference to an aspect stored in an Aspect Repository, or a reference to an already-instantiated remotely-accessible singleton aspect. The *scope* argument can be either *per-instance* or *per-distributed system*. The former weaves a distinct aspect-component instance at each specified join point; the latter instantiates a single per-system instance that is connected, potentially remotely, with each specified join point. Finally, the *locus* argument describes how advices should be applied at a selected join point in terms of either *before*, *after* or *around*.

Furthermore, the Configurator is also responsible for the management of *quiescence* (i.e. it ensures that the weaving/unweaving of aspects is not carried out while affected component/aspect-components/connectors are actively processing calls). To support this, the Configurator ensures that the weaving of aspects is not carried out while the relevant connectors or other components are actively passing or processing messages or calls. To do this, it requires that all connectors and components support a basic 'quiescence' interface as follows:

status = quiesce(timeout);

status = resume();

Because of the strictly stylised composition supported by AO composition, achieving quiescence is a relatively straightforward task compared to non-AO composition (e.g. [8]). The *quiesce()* operation simply freezes the start of the chain of aspects attached to the AO Connector (i.e. the AO-Connectors that correspond to the advices of the woven aspects) to prevent new threads entering, and then waits for any currently executing threads to drain from the aspect chain.

To execute *Configurator.reconfigure()* the following distributed protocol is performed:

1. *Configurator.reconfigure()* is called on one of the AO-OpenCom nodes; we will refer to this node as the 'initiator'.
2. The initiator determines how the aspect is to be applied. In the case of a per-distributed system scope, it instantiates the aspect at a suitable node and sends a remote reference to this to the nodes where it is to be woven. Otherwise, the initiator decides if it has the specified aspect available locally (or can get it from an Aspect Repository) and wants to send it 'by value' to the nodes where it is to be woven, or if it wants to send the aspect 'by name' and implicitly instruct the other members to obtain the aspect from an Aspect Repository.
3. The initiator sends a 'reconfigure' message to all the other AO-OpenCom nodes. This contains the parameters originally passed to *Configurator.reconfigure()*.
4. Upon receiving a 'reconfigure' message, each node's Pointcut Evaluator locates the target join points within its scope.
5. Each node's Aspect Handler then actions the 'add', 'remove' or 'replace' command as appropriate. For 'add' or 'replace', this may involve obtaining the aspect from an Aspect Repository. It will also involve weaving the aspect according to the specified scope and locus.

6. Each node replies to the initiator that it has completed the reconfiguration locally.
7. When all nodes have reported completion the initiator node returns control to the caller of *reconfigure()*.

An example of the use of *Configurator.reconfigure()* is given in Section 5.2.

4. The Consistency Framework

In this section we discuss our approach to the support of *consistent* dynamic reconfiguration. This is independent of the basic AO-OpenCom reconfiguration architecture discussed in the above section which handles only the basic mechanics of dynamic aspect (un)deployment. The *Consistency Framework* (COF) illustrated in Figure 3 consists of: a *System Consistency Framework*, a *Compositional Consistency Framework* and a set of 'threat aspects' which are responsible for guarding against consistency threats such as those identified in Section 2; these threat aspects are woven into the lower-level frameworks using the usual AO-OpenCom facilities.

The fundamental strategy of the COF is to guard against consistency threats by deploying 'threat aspects' *at appropriate join points within AO-OpenCom itself*. The benefit of this strategy is that threats can be handled in an incremental, selective and extensible manner where specific threat aspects can be deployed to guard against specific consistency threats. Crucially, we are using the same approach to guard against consistency as we are for 'ordinary' application-level dynamic reconfiguration: i.e. using aspect composition.

Turning now to the detail, the Consistency Configurator is responsible for managing these threat aspects and for deploying them at appropriate join points within the AO-OpenCom-based distributed system (see below).

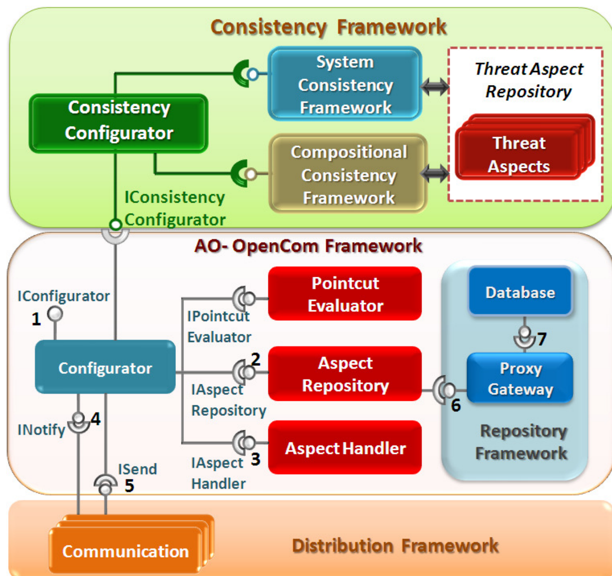

Figure 3. Applying Consistency Framework to AO-OpenCom

We now turn to a discussion of how the Consistency Configurator resolves each of the threats discussed in Section 2 by deploying appropriate threat aspects. When discussing the weaving of threat aspects, the following paragraphs refer to the numbered join points, 1-7, within the AO-OpenCom framework that are illustrated in Figure 3.

4.1 Addressing System Environment Threats

The Consistency Configurator uses the System Consistency Framework to instantiate the appropriate system environment threat aspects based on the reconfiguration needs as described in this section.

Protocol Message Disruption. To ensure that reconfiguration messages are not lost, the System Consistency Framework uses a *reliability threat aspect* and this aspect is woven at join points 4 and 5. The reliability threat aspect implements a reliability protocol atop the Distribution Framework to ensure that all messages are reliably received by each member. Because it is implemented as an aspect, this behaviour can be realised using various underlying mechanisms and can therefore be made straightforwardly applicable to a variety of implementation environments. This point is an important one and also applies to all the other threat resolution aspects to be discussed below.

In more detail, our currently-implemented reliability threat aspect is composed of an aspect with two advices and a 'message store'. The first advice is woven 'before' join point 5, and has the task of piggybacking reliability information to the message before it is sent via the *ISend* interface. The second advice is woven as a 'before' advice at join point 4 (i.e. before the message is delivered to the Configurator via *INotify*); this monitors incoming messages (and caches them in the message store), detects any losses within the transmission sequence, and requests retransmission of lost messages.

To weave the reliability threat aspect in a consistent manner (this again applies also to all the other threat resolution aspects to be discussed below) the *quiesce()* operation is first called on the connectors at join points 4 and 5 by the Consistency Configurator. Upon successfully achieving quiescence, the reliability threat aspect is woven at the front of the advice chain list (for brevity, we discuss this weaving process only for join point 5; see Figure 4); hence, it is invoked before method calls go to the Distribution Framework. Once the reliability threat aspect have been successfully woven at both join points, the *resume()* operation is called by the Consistency Configurator.

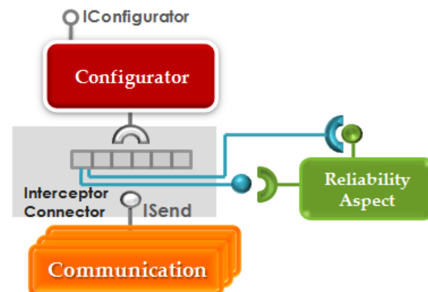

Figure 4. Weaving the reliability threat aspect at join point 5

Local Node Disruption Threat. To guard against this threat, the System Consistency Framework instantiates a *consensus threat aspect* and this aspect is woven at join points 4 and 5 to ensure that local node failures or disruptions do not compromise the consistency of the system. This aspect is 'flexible' in that it can implement any one of a range transaction protocols [7] depending on the specific requirements and deployment environment. To illustrate the operation of the advices we briefly describe our two-phase commit implementation. In this implementation, a 'before' advice woven at join point 5 takes messages before they are sent and converts them into the required sequence of messages for two-phase commit. Correspondingly, the 'around' advice at join

point 4 receives these transaction protocol messages and sends phase acknowledgements; it also communicates with the AO-OpenCom Configurator to enact or undo the local reconfiguration as appropriate.

Infrastructure Service Failures Threat. To guarantee the liveness of the infrastructure services (e.g. the Aspect Repository), the System Consistency Framework uses a *replication aspect*. This aspect is woven at join point 6 as an 'around' advice. Based on application requirements, a number of replication algorithms could be used to ensure maximum aspect availability and consistency during updates—e.g. the Coda [15] or Bayou [6] algorithms. More advanced algorithms which consider specific application and context requirements could also be used: e.g. Beloued [2].

Further, the System Consistency Framework uses a *load balancer aspect* to manage the load across the infrastructure services and this aspect is woven at join point 6 as a 'before' advice. Our current load balancer algorithm implements both the push and pull migration approaches [11]. The detailed functionality of the load balancing algorithm is beyond the scope of the paper; but, in brief, with push migration, periodic checks are made on the load of particular replicated repository loads, and as imbalances are found the load is evenly distributed from overloaded to less busy repositories. And the pull technique arranges that an idle replicated repository can transparently take tasks from a busy repository.

To prevent version conflicts in the Aspect Repository, the System Consistency Framework uses a *concurrency management aspect*. This aspect is woven as a 'before' advice at join point 7. The concurrency mechanism uses an optimistic read/write locking mechanism with priority for readers. Calls to update an aspect instance/version in the repository access the lock as a writer such that a writer can access the lock when there are no readers, while calls to retrieve aspect instances access the lock as a reader.

Simultaneous Reconfiguration Threat. To ensure that simultaneous reconfiguration requests do not interfere with one another, the System Consistency Framework uses a distributed *read/write concurrency aspect* and is woven at join point 1. This is an 'around' advice, the 'before' part being activated before the *Configurator.reconfigure()* is called. The advice then attempts to access the framework's lock set by the concurrency aspect, and blocks the call until this is obtained, at which point the reconfiguration can proceed. At this point, any reconfiguration attempts by other nodes are blocked until the present reconfiguration is complete, at which point the Configurator returns the *reconfigure()* call, and the 'after' part of the 'around' advice releases the lock.

Unauthorised Reconfiguration Threat. To prevent unauthorised nodes initiating reconfiguration, the System Consistency framework uses a series of security aspects, which are subsequently woven at join points 4 and 5. These comprise aspects that each addresses a different flavour of security threat: e.g. access control, integrity or confidentiality. The weaving order of these aspects is crucial: of the three mentioned the order would be authentication, confidentiality and then integrity.

Currently, an *authentication aspect* is woven as a 'before' advice at join point 5 such that it is called before the Distribution Framework and performs access control before allowing continuation. Then a *confidentiality aspect* encrypts the arguments of method calls as they are passed through the Distribution Framework. This is achieved by weaving an encryption advice as a 'before' advice at join point 5 and a decryption advice at join point 4, also as a 'before' advice. Finally the System Consistency Framework implements an *integrity aspect* in terms of an SSL layer between reconfigured nodes.

4.2 Addressing Compositional Threat

The Consistency Configurator uses the Compositional Consistency Framework to instantiate the appropriate compositional threat aspects based on the reconfiguration needs as described below.

Unsynchronised Weaving of dependent aspect Threat. The Compositional Consistency Framework uses a *transaction management concurrency protocol* or *coordination protocol* to preserve compositional dependencies. Each of the protocols is encapsulated as an aspect and is woven as a 'before' advice at join points 4 and 5. This process is equivalent to that used for threat 2. Here, the *Saga* transaction model [7] allows dependent aspects to be divided into a sequence of *sub-transactional aspects*, each of which manages an associated compensating sub-transaction that can be triggered to undo the effects of the committed sub-transaction aspect in case one fails.

With respect to the coordination protocol, protocol the Compositional Consistency Framework uses the NeCoMan [9] protocol which is encapsulated as an aspect and woven to provide synchronisation between the reconfigured entities.

Unsynchronised binding of distributed remote aspects. To prevent race conditions in which remote connectors attempt to communicate with remote aspects that have previously been removed, a 'before' advice is woven at join point 3. This detects when a 'remove' command is passed to the Aspect Handler, and in response weaves a *proxy caretaker aspect* this is woven in front of proxies for the removed application aspect. Then, when a remote client (connector) attempts to invoke this removed aspect, the proxy caretaker aspect is invoked instead which redirects and informs the remote connector that the referenced aspect has been removed. To avoid the connector from invoking the aspect in the future, it removes the remote aspect reference from its aspect chain when it receives the 'remove reference' message.

Mutual exclusion of Aspect(s) Threat. To ensure that conflicting aspects are not composed, the Compositional Consistency Framework uses a *semantic reasoning and resolution aspect* (e.g. [17]) and is applied at join points 1 and 4. This aspect holds application-specific rules about which mutual exclusive behaviours are allowed and not allowed when reconfiguration (both addition and removal of aspects) is performed. Using reflection, it identifies aspect(s) woven at the join point and determines if adding or removing the aspect will cause any inconsistencies. For detected conflicts an exception is raised and the reconfiguration is aborted.

4.3 Ordering of Threat Aspects

Although the threats discussed above are essentially orthogonal to one another, the order in which the corresponding aspects are composed is still important. For example, when the consensus aspect is woven at join points 4 and 5, the reconfiguration can proceed in either of the following ways: (i) if no threat aspects are deployed then the consensus aspect is then woven as a 'before' advice; or (ii) in the case where the threat 1 aspect has already been woven, the consensus aspect is woven as a 'before' advice with position 2. The decision is determined from priority ordering information attached as attributes to the individual aspects. Weaving the reliability aspect first ensures that a reliable consensus protocol is selected.

The order in which aspects woven at the same join point are invoked affects the reconfiguration semantics. This is particularly true for join points 4 and 5 at which numerous aspects are woven. Aspects being executed in the wrong order could lead to situations in which a message needing to be processed by a particular aspect has already been consumed by another.

To guard against such eventualities, the COF mandates a particular order for the weaving of the threat aspects. These are illustrated in Figures 6(a) and 6(b) which respectively illustrate the required ordering at join points 4 and 5.

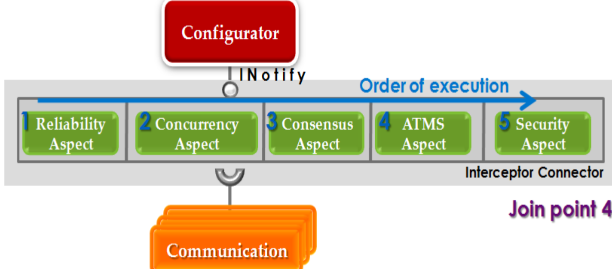

Figure 6(a). List of threat aspects woven at join point 4

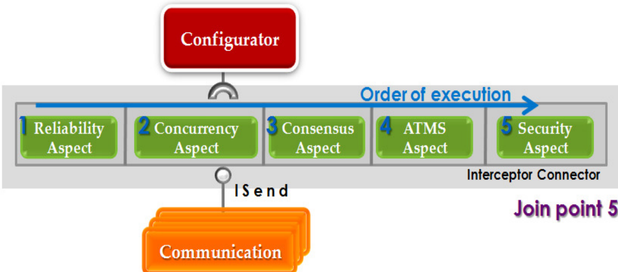

Figure 6(b). List of threat aspects woven at join point 5

5. Evaluation

We focus on two dimensions of evaluation: (i) the extent to which AO-OpenCom/COF achieves our stated goals of configurability and transparency; and (ii) the overhead of AO-OpenCom/COF in 'typical' usage scenarios.

5.1 Configurability

In Section 4.1 we have already demonstrated the *configurability* of AO-OpenCom/COF in addressing a wide range of consistency threats. Our general approach to dealing with such threats—i.e. by selectively applying threat aspects to join points in AO-OpenCom itself—is inherently highly configurable and can be changed or extended simply by applying different threat aspects. However, two potential vulnerabilities of our approach might become evident if new threat aspects are added to the set we have already identified: (i) it could become harder to keep track of the threat aspect ordering constraints discussed in Section 4.2; and (ii) there could be an increased possibility of undesirable interactions between the behaviour of the different threat aspects. The extent to which these vulnerabilities become problematic will become clearer with experience. However, we believe that the set of threat aspects we have identified is already quite comprehensive, and that many cases can be covered with the current set alone. Underlying this belief is our experience that most threats seem to reduce to a tractable number of common underlying patterns.

5.2 Transparency

Turning now to the issue of *transparency*, AO-OpenCom/COF naturally supports a *selectively transparent* approach. At one extreme, an appropriate set of threat aspects can be pre-configured at application start-up time so that the application programmer who wishes to initiate a run-time reconfiguration needs only to make the appropriate call to *Configurator.reconfigure()*. This achieves complete transparency of consistency-related mechanisms. At the other extreme, the programmer can be explicit about which threat aspects should be put in place for each reconfiguration. In this case, COF will apply the requested threat aspects on-the-fly (if they are not already present) before proceeding to perform the requested reconfiguration. Note that this extreme is still *partially* transparent as the programmer is protected by the Consistency Configurator from the low level details of actually weaving the threat aspects.

To illustrate the partially transparent case consider a reconfiguration scenario relating to the case study in Section 2. Assume that the application programmer wants to add an MPEG4 video codec aspect to all nodes in domains 1 and 2 which already have video-codec components with an *IMPEG* interface. Further assume that domains 1 and 2 offer reliable TCP-based communications. The programmer would specify the reconfiguration request by writing code along the lines of Figure 7 (the code is simplified for presentational purposes).

Note that the required threat aspects are specified as part of the aspect specification. In this case no compositional threats are applicable, and the *protocol message disruptions* threat (T1) is not applicable either because of the availability of TCP. This leaves only the remainder of the 'system environment' threats: i.e. threats T2-T5. The *Configurator.reconfigure()* call takes the given pointcut and aspect specifications and also specifies that the specified aspect should be added, that the scope of the reconfiguration should be the entire DCF and that the weaving locus should be *before*.

```
Pointcut pc = new Pointcut( "domain1* && domain2*", "video-
        codec*", "IMPEG", "video-player*");
Aspect aspectVideo = new Aspect(MPEG4VideoCodec, "T2 T3
        T4 T5");
Configurator.reconfigure(multimedia_app, pc, add, aspectVideo,
        perDCF, before);
```

Figure 7. Reconfiguration specification

5.3 COF Overhead

The following experiment was performed on two Core Duo 2, 1.8 GHZ PCs' with 2GB RAM running Windows, and using the Java-based version of AO-OpenCom. Each measurement was repeated ten times and mean values taken to discount anomalous results. The purpose of the experiment was to evaluate the performance overhead of dynamic reconfiguration operations using AO-OpenCom and COF, We approached this by instrumenting an implementation of the application scenario described in Section 5.2, while using different threat aspect configurations from the consistency framework.

The results are shown in Figure 8 which shows the measured overhead of the following 4 cases: (i) reconfiguration without COF; (ii) reconfiguration using COF with the system consistency framework threat aspects only; (iii) COF with the compositional consistency framework threat aspects only; and (iv) COF with both the system and compositional consistency framework threat aspects.

We can see a linear increase in overhead when applying COF for compositional threat aspect while a non-linear increase of overhead for System Consistency Threat aspect used as the number of reconfigured nodes is increased. This is explained by:

- the fact that the initiator node is a bottleneck (this could in principle be alleviated by configuring AO-OpenCom with slave Configurators to increase parallelism);
- weaving of dependent aspects are treated as sub-transactions over a mixed set of nodes. The set of affected nodes having dependent causes affected nodes to dependent on each other, causing the overhead to be higher.

Overall, based on our experiments, we can conclude that the run-time overhead of COF is acceptable; with each threat aspect capable of being independently woven each threat aspect can be individually deployed based on the required reconfiguration context, thus significantly reducing the overhead compared to all threat aspects being deployed.

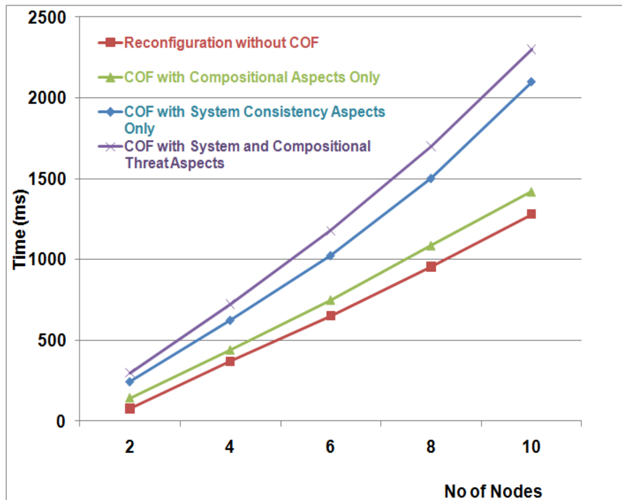

Figure 8. Overhead of reconfiguration using COF in AO-OpenCom

6. Related Work

Few AO middleware platforms have addressed the challenges of performing consistent dynamic reconfiguration. DyMac [10], and CAM/DAOP [14] are prominent examples of distributed AOP platforms that have no support for dynamic reconfiguration. Other prominent platforms such as Spring AOP [1] and FAC [13] do support reconfiguration, but do not support distribution; these systems have not needed to consider strong consistency mechanisms as reconfiguration is considerably simpler when confined to a single node.

JAC [12] is an early example of a distributed platform that supports dynamic reconfiguration. However, this support involves only the reconfiguration of advices at individual join points and provides no support for distributed consistency management.

AWED [3] supports dynamic weaving of aspects using the DJAsCo [20] distributed AOP architecture. It supports the weaving of stateful distributed aspects, and through the use of a consistency protocol ensures that whenever an aspect is woven at a specific host, mirrors are also woven at other involved hosts. However, AWED do not consider any other consistency threats as discussed in the our proposed solution.

ReflexD [18] also supports dynamic weaving/unweaving of mirrored aspects, and uses a framework to provide system-wide consistency. However, as in AWED ReflexD aspects exist only as mirrored aspects although unlike AWED, ReflexD ensures that whenever an aspect is changed the corresponding remote copies are synchronised. But again, the consistency mechanisms provided do not generalise to the extent of our proposal.

Finally, DyReS [19] is an AO middleware framework developed on top of JBOSS dynamic AOP [4] and Spring AOP [1] that provides consistent dynamic reconfiguration in a more sophisticated manner than the systems reviewed above. More specifically, DyReS uses a coordination protocol that allows aspects to be dynamically added and removed in a consistent manner by achieving quiescence. The protocol is based on two synchronisation primitives: *wait* blocks the ongoing reconfiguration process until it gets a notify message from a specified node; and *notify* sends a synchronisation message to a specified node. Although this approach supports a degree of generality (i.e. it is portable over multiple underlying platforms), it again does not generalise to a wider set of consistency threats. For example, when deployed in a wireless network environment there is no way to address the possibility of lost or reordered synchronisation messages or other system viroment threats as in our approach. Furthermore, compositional threats are not addressed in DyReS. Our approach is more flexible, allowing different consensus and consistency protocols to be chosen based on the required reconfiguration, the current environment, and the wide range of threats that are posed.

7. Conclusion and Future Work

In this paper we have identified a number of important threats to maintaining the consistency of distributed reconfiguration operations in AO middleware environments. We believe these threats to be representative of the type of threats that should be considered by all dynamic AOP platforms. More specifically, we have presented the AO-OpenCom platform which supports the composition and reconfiguration of distributed aspects, and an associated distributed consistency framework called COF that ensures that all of the identified threats are handled in a transparent manner. COF has the following important benefits. First, it is *simple and elegant* in that it uses aspect composition to deploy these consistency mechanisms. Second, it is *flexible and configurable* in that appropriate threat aspects can be dynamically woven and unwoven according to the types of threat and environmental conditions currently pertaining. Third, it is *inherently extensible* in that new threat aspects can be developed and woven into the system at appropriate join points as and when new threats are identified. Fourth, it achieves the maintenance of consistency with a *reasonable overhead* compared to unsafe reconfiguration.

There are several research directions that we would like to investigate in the future. First, we are currently working on performance optimisations to reduce reconfiguration overheads through the use of multiple (slave) Configurators in cases where a reconfiguration needs to be carried out on a large number of nodes. This should reduce the overheads identified in Section 5 to something closer to constant time. Second, we will investigate the potential for embedding our approach in a self-managing, autonomic environment. Finally, we plan to integrate our framework with appropriate modelling tools which can support the developer in designing, evaluating and validating complex aspect reconfigurations before they are deployed into a distributed system.

References

[1] Spring website. http://www.springframework.org/.

[2] Beloued, A., Gilliot, J.M., Segarra, M.T., Andre, F. “Dynamic data replication and consistency in mobile environments”, In Proceeding of the 2nd doctoral symposium on Middleware, ACM, NY, 2005.

[3] Benavides, L., Sudholt, M., Vanderperren, W., et al., “Explicitly distributed AOP using AWED”, In Proceeding 5th International Proceeding Conference on Aspect Oriented Software Development, Bonn, Germany, March 2006.

[4] Burke, B., “JBoss AOP Tutorial”, 3rd Conference on Aspect Oriented Software Development, Lancaster UK, 2004.

[5] Coulson, G. Blair, G., Grace, P, Taiani, F., Joolia, A., Lee, L., Ueyama, J., Sivaharan, T., “A Generic Component Model for Building Systems Software”, ACM Transactions on Computer Systems, TOCS, 2008.

[6] Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Welch, B., “The bayou architecture: Support for data sharing among mobile users.” In Proceedings IEEE Workshop on Mobile Computing, pages 2-7, 1994.

[7] Garcia, H., Salem, K., “Sagas”, ACM Conference on Management of Data, 1987.

[8] Grace, P., Coulson, G., Blair, G., Porter, B., “A Distributed Architecture Meta Model for Self-Managed Middleware”, In Proceeding 5th Workshop on Adaptive & Reflective Middleware, 2006.

[9] Janssens, N., Joosen, W., Verbaeten, P., “NeCoMan: middleware for safe distributed-service adaptation in programmable networks”, In IEEE Distributed Systems Online, 2005.

[10] Lagaisse, B., Joosen W., “True and Transparent Distributed Composition of Aspect-Components”, In Proceeding Middleware Conference, LNCS 4290, Melbourne, 2006.

[11] Minson, R., Theodoropoulos, G., “Adaptive Support of Range Queries via Push-Pull Algorithms”, 21st Workshop on Principles of Advanced and Distributed Simulation, 2007.

[12] Pawlak, R., Senturier, L., Duchien, L., Florin G., “JAC: A Flexible Solution for AOP in Java”. In Proceeding 3rd International Conference on Metalevel Architectures and Seperation of Crosscutting Concerns, 2001.

[13] Pessemier, N., Seinturier, L., Duchien L., Coupaye, T., “A component-based and aspect-oriented model for software evolution”, International Journal of Computer Applications in Technology, Volume 31, Number 1-2, 2008.

[14] Pinto, M., Fuentes, L., Troya, J.M., “A Component And Aspect based Dynamic Platform”. The Computer Journal, 2005.

[15] Satyanarayanan, M., “Coda: A highly available system for a distributed workstation environment.” IEEE Trans. Computing, 39(4) pg. 447-459, 1990.

[16] Surajbali, B., Coulson, C., Greenwood, P., and Grace, P. “Augmenting reflective middleware with an aspect orientation support layer. In Proceeding 6th Workshop Adaptive and Reflective Middleware, 2007.

[17] Surajbali, B., Grace, P. and Coulson, G. 2009. A Semantic Composition Model to Preserve (Re)Configuration Consistency in Aspect Oriented Middleware. In Proc. 8th Workshop on Adaptive and Reflective Middleware. 2009.

[18] Tanter, E., Toledo, R., “A Versatile Kernel for Distributed AOP”. In Proceeding International Conference on Distributed Applications and Interoperable Systems, June 2006.

[19] Truyen, E., Janssens N., Sanen, F., Joosen, W., “Support for distributed adaptations in aspect-oriented middleware”. In Proceeding of the 7th International Conference on Aspect Oriented Software Development, April 2008.

[20] Vanderperren, W., Suvee, D, Wydaeghe, B., Jonckers, V., “PacoSuite and JAsCo: A visual component composition environment with advanced aspect separation features”, Conference on Fundamental Approaches to Software Engineering Poland, 2003.

Malleability, Obliviousness and Aspects for Broadcast Service Attachment

William Harrison

Department of Computer Science
Trinity College
Dublin 2, Ireland*
(+353) 1-896 8556
Bill.Harrison@cs.tcd.ie

Abstract

An important characteristic of Service-Oriented Architectures is that clients do not depend on the service implementation's internal assignment of methods to objects. It is perhaps the most important technical characteristic that differentiates them from more common object-oriented solutions. This characteristic makes clients and services malleable, allowing them to be rearranged at run-time as circumstances change. That improvement in malleability is impaired by requiring clients to direct service requests to particular services. Ideally, the clients are totally oblivious to the service structure, as they are to aspect structure in aspect-oriented software. Removing knowledge of a method implementation's location, whether in object or service, requires re-defining the boundary line between programming language and middleware, making clearer specification of dependence on protocols, and bringing the transaction-like concept of failure scopes into language semantics as well. This paper explores consequences and advantages of a transition from object-request brokering to service-request brokering, including the potential to improve our ability to write more parallel software.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, concurrent programming structures.* D.2.11 [**Software Engineering**]: Software Architectures – *data abstraction, languages.*

General Terms Design, Languages.

Keywords Service-Oriented, Aspect-Oriented, Programming Language, Middleware, Concurrency.

1. Introduction

With current approaches to software design and implementation, software artifacts, like classes or methods, embody many decisions made at they time they are designed and implemented. In more fluid environments, including distributed, autonomic, grid, and service-oriented, that are emerging today, we need more of these choices to be deferred until run-time. For example, today the client of a method specifies where to find its implementation, whether in an object or in a service. In common usage, malleability is the ability of an article to be molded or shaped to fit for changing circumstances, and we have applied the term to software artifacts [5]. Enhancing malleability requires us to re-think the boundary between programming languages and middleware and introduce a point at which intelligent choice can be injected into the otherwise rigid semantic specification. But introducing a locus for such an intelligence allows us to better address the need for greater parallelism that we confront in the multi-core future.

To address the need for greater malleability, we advocate the use of a programming model, called the *broadcast service model*, with several novel characteristics:

- There is a modularizing construct, called *service*, that contains a coherent collection of classes and has a run-time instantiation. Services may be responsible for handling method calls made by a client or may be attached obliviously, as aspects. They may be bound within a process or located remotely, and they hold the state for various aspects of objects.
- There is an interface-like construct, called *face*, that characterizes a set of methods that can be called safely, but does not indicate which object or service implements them.
- Clients may use a single reference to an object even when its state is distributed across several services. The services are responsible for resolving references, so that an object's methods can access its state.
- Method invocations do not indicate a particular target object or service. Instead, invocation is broadcast, with the intelligence guiding the delivery of a method call to one or more implementations being provided by a middleware-defined dispatcher, caller a *service request broker*.
- All execution takes place within a transaction, which serves to circumscribe behavior on failure.

To explore this model, we are developing a programming language, Continuum [14], which embodies this structure and introduces constructs that enhance malleability. In the next section of this paper, we use a small example to illustrate the underlying issues of malleability and obliviousness, point-to-point service provision, aspect attachment, and broadcast service provision. In the third section we then outline some of the technical challenges that must be met to realize those advantages. The fourth section then describes advantages to be achieved by combining the different kinds of obliviousness provided by aspect-oriented and by service-oriented technologies. These advantages include not only increased malleability, but also a basis for describing statically enforced future processing commitments. These commitments can be used to merge process-flow and call/return paradigms and naturally express latent parallelism, to better exploit multi-core processors.

2. Underlying Issues

2.1 Malleability

Malleability is much like reusability, except that instead of characterizing how an artifact can be reused during the development of new artifacts, malleability characterizes how flexibly it can be used at deployment-time or run-time. For example, using Java's ability to describe the type of a parameter with an interface rather than with a class makes methods both more reusable and more malleable. But unlike Java, ADA and Modula-3 can identify a parameter's purpose by name. The order in which these names

* This work was made possible by a grant from the Science Foundation Ireland

are used by a method's caller can differ from the order in which the implementer listed them. This improves the malleability of both. But it does not improved their reusability because the order used by the selected implementation is evident at development time in any case. The information needed to reorder parameters can be provided without undue burden on the developer by referring methods and parameters defined in interfaces to a glossary that, like JavaDoc, provides extra-lingual information about meanings.

Generally speaking, malleability cannot be achieved by adding one or another language feature to address it, although features that increase the specification content of software over its algorithmic content add to software's malleability. It is instead easier to enumerate characteristics of software that inhibit malleability, and propose their removal or the substitution of equivalent characteristics in different form. We mentioned sensitivity to parameter order as one example, above, but there are many other ways that two implementations of the same function can differ. A trivial example is tolerance for name variations to be used for methods or types, which can be resolved through the use of the same glossary mentioned above. A more subtle inhibitor to malleability is the assignment of the implementation to a particular "target" object, which is otherwise just one among the several parameters. We have called the ability to ignore such differences "structural abstraction" [8] and defined a Java-compatible programming language called Continuum that permits the objects responsible for method implementations to be imagined differently by clients and services [9],[14].

2.2 Obliviousness

Obliviousness is an important way to achieve malleability. That is one of the reasons it is so important in separating concerns. Using structural abstraction, the service-oriented model for software provides a way to make clients oblivious to the issue of where a method is implemented within a service. But it does so by introducing a new structural dependence. By modeling services as objects, it replaces dependence on the assignment of methods to objects with dependence on the assignment of methods to services. As with traditional target-directed object calls, service requests and responses are *point-to-point*, forcing the client to rigidly reflect the realization of function by services.

On the other hand *obliviousness* [1] is one of the hallmarks of aspect-oriented technologies, which hide service attachments from clients. The concept of obliviousness recognizes that the flow of logic within software is not sensitive to independently described aspects that may each carry part of its state. Aspects can be used for attachment of systemic function like management of transactions, or for composition of component functionality like editing, display and validation of the elements of a development environment. No matter which, the fact that the combined aspects' code is oblivious to the manner of their combination is a major contributor to the software's malleability. However, today's exploitation of aspect-oriented concepts is only applied where clients do not control or direct the aspect code. It is not appropriate for modeling service attachment to clients because, after all, the conventional model of method call forces the client to know the method's implementer.

In [9], we observed that structural abstraction can be achieved by changing the nature of method call from point-to-point to broadcast. This is a deep change conventional object-oriented semantics in which method calls are always directed to a target object in a point-to-point fashion Rather than statically binding methods to classes, the face merely indicates that the methods' implementations have been demonstrated to be available. The same flexibility can be used to organize services in a manner making clients are oblivious to their structure.

2.3 Using Broadcast Method Call for Malleability

We will use a family of related examples shown in Figure 1 to illustrate several points related to use of structural abstraction and broadcast for method call. The problem being addressed is motivated by the commonly seen phenomenon of mobile phones' ability to optionally display a clock on the phone's window. Presuming a method "display" that a client can use to put the clock in the display – one can ask how to write the method call that invokes it. The lower part of Figure 1 shows such a client. To focus on structural abstraction, we ignore the grey-shaded material. The upper part of Figure 1 shows several possible server implementations. In (a), "display" is implemented in the window object, while in (b) it is implemented in the clock. In the interest of malleability, we wish the client to use either service supplier, where in this case we might presume that the "service" is local and in the clients classpath. The use of both interface and face declarations in the client is intended to highlight the fact that a legacy client may have expected the implementation to be in "Window". Treating interfaces as a "sugar" allows use of the legacy style.

The syntax in these examples is familiar in form, but has subtly different meanings from what may be expected. Full description of the type model supporting structural must be left elsewhere [3], as it would consume the space allotment for this paper. In brief, there is a classification hierarchy for objects in which a classifier can have more than one super-classifier. Classes, as distinct from classifiers, define the state and method implementations for objects, and are attached as leaves to the classification "tree". Classifiers

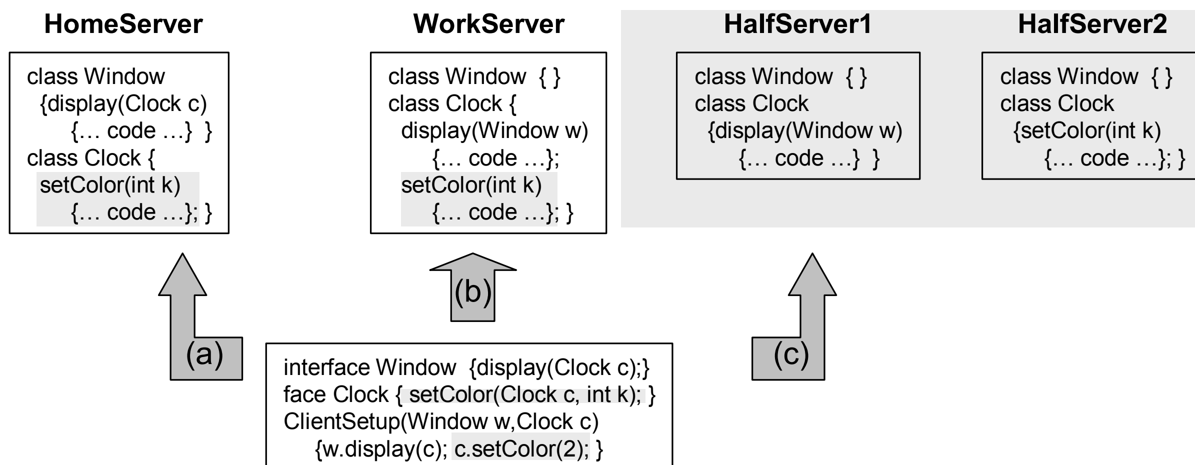

Figure 1 – Examples of Different Server Combinations for a Client

are not statically tied to sets of methods, which are called "faces". Instead, the declaration of a reference variable indicates a classifier, a face, and whether the reference value may be null. If the reference is not null, the methods in the face are assured to be accessible by the dispatcher. The set of assured methods can grow over time, and any one declaration need mention only a subset of the assured methods. The methods' implementations need not lie in the object referenced by the reference. They may be in any of the required parameters, or even statically available. The client need not know the service provider's class structure at all, which characteristic we call *structural* abstraction. We treat declaration of an interface as a syntactic shorthand for a face in which all of the methods have an additional parameter with the appropriate defaulted classifier. The net effect of this type model is that the client in Figure 1 is type-compatible with any of the servers.

Broadcasting method calls increases the malleability of software artifacts. It can do this because it replaces the concept of an interface that indicates methods available from an object with the concept of the *face* that indicates methods available somewhere in the environment. In the usual model, methods in an interface are available only if the associated reference is not null. As with conventional point-point interfaces, for broadcast the methods in a face are also only assured to be available if the associated reference is not null.

Interfaces are quite useful as vehicles for labeling the known contents of objects as they come from their implementer. But from the point-of-view of a function's consumer, what is of interest are the behaviors themselves, not which behavior is implemented by which object. So the "face" concept that replaces it identifies a set of methods on a variety of objects that must be available in the computing environment – in the "cloud", so to speak. A method call is not directed to an object, but is broadcast through the cloud to an implementation. The implementation has been proven, through the type system's interaction with the dispatcher, to be available somewhere. The difference is illustrated in the transition from Figure 1a to Figure 1b. When connected to the HomeServer (a), "display" has an implementation in the Window, as the client seems to expect. But the client can equally use the WorkServer (b). The client's use of a "target" in the call, as in the example's call to "w.display(c)" does not imply that the message is delivered to the target. In the example, since both window and clock are required (i.e. may not be null), it may lie in either. The actual target of the call is not defined by the language but by an extended dispatcher, the service-request broker, which finds the target's service as well as the class implementing the method, as described in the next section.

2.4 Using Broadcast Method Call for Obliviousness

Broadcasting method calls combines structural abstraction's obliviousness to object structure with aspect-oriented software's obliviousness to service structure. Doing so increases the malleability of software artifacts further above either structural abstraction or service-oriented approaches. The service request broker tracks the availability of services and routes called methods to the appropriate object in the appropriate service, freeing clients from the knowledge of the object or service that implements them. The grey-shaded material in Figure 1 focuses our attention on obliviousness to service structure. The services can be local to the client's classpath, or distributed elsewhere. The figure illustrates a composition of two services (c) that supplies the same needs as did (a) or (b) as single services. It shows a situation in which the displayed clock has settable state information to control its color, made available as a "setColor" method.

To completely separate client from the implementation's structure, we do not allow a client to name classes (implementations) at all. This carries the use of interfaces for characterizing object types to an extreme. Unless employing a factory pattern, the most common practice today is for a developer to instantiate a class whose characteristics are known to meet the functional needs, rather than to leave that selection to be made at run-time. This, again, implies knowledge of the class structure of the anticipated provider of services. To completely decouple the client from the service structure, we instead simply specify the classification (how the class must relate to its subclasses) and the face (which methods that are needed). In some cases, the provided set of methods may need to result from the composition of several available services, carried out behind-the-scenes from the client that creates the object.

In better separating client from object structure, the broadcast model's *face* also better separates clients from service structure. The fact that service boundaries are transparent allows the service model to be used at fine granularity. While we want to allow services to be distributed and mobile, for services to be composed locally, within a single process, and it is important that a two-level dispatch be avoided. The client does not target the method to a specific service provider, but allows the service-request broker to find and direct the call appropriately. If we include the grey-shaded material in examining Figure 1, we note that when supported by the composite server (HalfServer1 and HalfServer2), the client's apparent call to a the "display" method in Windows is actually implemented in HalfServer1's Clock, while the call to setColor, made with no specified target is implemented in HalfServer2's Clock.

3. Broadcast Service Model Challenges

We can foresee several challenges in trying to move from the current target-directed models for objects or services to the kind of broadcast service model that would provide the advantages described, including: compatibility, state maintenance, service visibility management, general service management, and commitment satisfaction tracking.

3.1 Compatibility

Any shift in programming paradigm will fail if it can not accommodate previously-written software. Even the successful shift from procedural to object-oriented was enabled by fact that C++ by definition included all of C, and the ongoing transition to service-oriented architectures is facilitated by treating services as objects within the object-oriented paradigm. It is therefore no accident that the broadcast service model for method call can include the conventional object-oriented model as a syntactic and semantic subset, though it is challenging to do so when eliminating the concept of target. As sketched in Section2.3, it preserves the concepts of a type hierarchy of classes (classifiers), of the association between classifiers and sets of methods, and of the fact that non-null references are required to assure the safety of method calls. But it permits parameters to be reordered and does not require the method implementation to reside in the target of the method call made by the client. Existing class implementations all function properly when interpreted as broadcast method calls rather than point-to-point calls.

3.2 State Maintenance

Many object-oriented systems maintain consistency using the simple premise that an object holds all its state and sees all of the method calls that make changes to the state. While simple, the premise is also frequently invalid in its over-simplistic view of the nature of state. For example, objects may indirectly access and return state maintained in other objects, in which case they will not see when the value changes by a call on the other object. In fact, as discussed in [4], the idea that an object has an objectively-definable state is itself limiting. Using an object's identity, other

objects may maintain additional data, whether in hash-tables or in aspects to which an application is oblivious.

Most object models rely on the idea that specifications about method dependence need not be included in a class specification because the object is guaranteed to have "seen" all methods called on it since its creation, in the order they are called. However, concurrent call and dynamic attachment of oblivious aspects violate this principle. On one hand, a dynamically attached aspect does not "see" methods called prior to its attachment. But on another hand, an object's state is not an opaque totality, but is the product of state contributions made by the independent aspects.

Because an object's state can be distributed among several services, a service that becomes newly available may not have an accurate picture of the object's state with respect to a client's prior calls within a transaction. To prevent inappropriate action, descriptions of the faces provided by services must include declarations that identify dependencies between method calls. Such dependencies are generically called *choreography*, and are recognized as important for service composition[16]. In Continuum[14], these constraints are expressed by indicating that a particular method is available only if all prior calls to specified other methods have been seen by the same service.

3.3 Service Management

Protocol constraints make it possible to determine that some services should not be visible to certain clients, but there can be other reasons, like cost, service-levels, or business arrangements that play as well. Current service-oriented systems generally manage visibility in a rather static fashion in which clients initiate a service-finding operation and then access the found service through a proxy. But this approach only works because the clients are dependent on the structure of the services, and would inhibit the kind of flexibility implied by the grey-shaded material in Figure 1. To free clients from this concern, the matching of clients to services is performed by the service broker, which can perform necessary bookkeeping with respect to the transaction, identified on each call.

But this does not serve the needs of dynamic, mobile environments well. If a service being used moves out of range, an alternative one visible for the client needs to be used instead. The service request broker is responsible for receiving communication from services joining the bus, and for managing their exit. In addition, the service broker must recognize that some services have a mutual awareness – they may be substitutable, as would be the local entry ports for commercial enterprises, or they may be incompatible or have other contractual relationships

In addition, the service request broker may perform *ad-hoc* composition of services needed to satisfy a client. If a client expects a face providing services for managing both air and hotel bookings and the available services provide one or the other, the broker can compose the services into a larger structure automatically, rather than requiring that the aggregated service be implemented particularly to perform both functions or coordinate both services.

While today's service-brokers could be imagined to provide an appropriate place for managing these functions, the fact that clients must specifically recognize distinct services as objects as discussed in Section 2.2 makes extending their capabilities cause changes to the clients. However, use of a broadcast service architecture allows the capabilities to be provided to clients transparently, without disrupting their operation.

3.4 Static Tracking of Commitment Satisfaction

There is a traditional gulf between object-oriented programming languages and work-flow architectures. In object-oriented languages, the client determines the target of a call, and waits for its completion. This is a powerful inhibitor to greater use of parallelism. In work-flow architectures on the other hand, the sender does not wait for its completion but target of the message is specified by the flow-designer. This provides many opportunities for parallelism, but the use of two architecturally disparate elements seems too cumbersome for use in algorithm description. This may be the reason the combination has not been adopted as a conventional programming language. The broadcast service model's use of a service request broker provides a novel way to integrate the concerns of programming language and work-flow architecture.

To exploit this capability, two additions are made to conventional programming language constructs: 1) the method call and message send constructs are unified, and 2) the concept of a statically declared "commitment to call" is introduced. A method may be declared to guarantee the future call of another method, as illustrated with the "`sends`" keyword in the `face` definition:

```
face X {void f1(A a, B b) sends f3(A a));}
```

This declaration defines a face, X, that declares support for a method f1 of two parameters. Method f1 commits to the eventual calling or sending of another method, f3, using the value provided by f1's first parameter.

Unlike conventional call's semantics, the static commitment to eventually call f3 need not be satisfied before the method carrying the declaration returns, but must be satisfied by the end of the outermost transaction in which the commitment is required. Thus, the commitment can be satisfied by the method itself during its execution or by the execution of methods it calls, or by a method to which a message is sent, perhaps much later than the client's completion.

A method's declaration may include a list of such static commitments. The method declaration can only be satisfied by an implementation that itself declares the satisfaction of the commitment. To enforce the behavior, the commitment must be satisfied on all paths from the entry of the method, either directly, or by *call* or by *send*.

Presuming this, another method, f2, which is also committed to send f3 can be implemented as:

```
void f2(A u) sends(f3(A u)) {
       // other computation
       send f1( u, new B() );     }
```

This implementation is valid because f2 sends f1 which is committed to send f3, thus satisfying f2's commitment. But if calls to f2 are made in a loop, only the "other computation" is serialized in the loop. The execution of the resulting f1's can all occur in parallel with the loop's execution.

As described in [6], dynamic failure to satisfy a commitment, whether by thrown exception or by reduction in resources can be handled locally, or it causes the transaction to abort.

Where appropriate, the use of static commitments also enables a *call* that would occur inside a loop to be transformed into a *send*. The service request broker can enable these activities to occur in parallel. Because the committed action is not guaranteed to take place immediately, the original caller can employ this mechanism only if further computation in that caller does not need to use the results. However, it is possible to define commitments in a way that enables subsequent gathering and processing of the results.

This alternative view of computation is made possible because unlike a conventional target-directed call, the use of a broadcast model allows the request broker to act in a store-and-forward capacity for parallel messaging in addition to the immediate-invocation-and-return capacity for conventional dispatchers.

4. Broadcast Service Model Advantages

The broadcast model enhances malleability by changing the programming language model to employ broadcast rather than point-to-point semantics for its call and to make clients oblivious to the structure of services. In doing so, it eliminates the need for a syntactically special target object on call. This is perhaps fortuitous, because instead of passing an implicit target, the language can instead reflect the concurrent structure of the software by passing a transactional context for bounding the action to be taken on failure.. The failure recovery points must be indicated directly within the code that engenders possible failures to permit us to write software with more latent concurrency than present.

4.1 Enhanced Malleability

The increased software malleability made possible by changing the programming model from the point-to-point model used by both classical object-oriented programming and popular service-oriented architectures to the broadcast model provides several malleability advantages:

Greater tolerance for different implementation structures. The un-shaded material in Figure 1 illustrates how a client needs no change to tolerate a different service provider that moves the display method either to different classes (since the implementing class need not be mentioned in the call) or to different services (since the service also need not be mentioned in the call).

Accommodation of dynamic service composition. The grey-shaded material in Figure 1 further illustrates how the client needs no change to tolerate a change to a different service provider structure altogether, since the service is not mentioned in the call. Since neither target objects and services are not mentioned in the call statements, combinations of services used to satisfy a client's needs can be fluidly composed by the service request broker.

Scalable component composition. Component structures like those employed in service-oriented software architectures suffer from severe performance problems when used at finer granularity in an attempt to obtain improved the software structure it offers [11].The use of transparent services, with a broadcast model of method call like that illustrated in Figure 1 enables the implementation to move the task of message and data format transformation out of the client and into the service request broker. This enables the associated overheads to be avoided when component structures are tightly bound within a process.

Avoidance of proxy management. In the usual division of concern between programming language and middleware, the programming language specifies the complete semantics of method call, including the rules for determining how to find the implementation corresponding to any particular method call. In architectures for distributed, autonomic, grid, and service-oriented systems, the linguistic specification is ultimately quite incorrect. The intervention of middleware takes the "call" out of the realm of language specification and makes non-linguistically specified choices. In fact, modern Object Request Brokers allow the client and service to be realized in different programming languages, making the specification of the dispatch process as a linguistic characteristic impossible. But the task of interfacing this flexibility with the language specification is forced upon the client in the form of "proxies" – local objects that intercept the linguistic specification and inject alternative mechanisms. Much greater flexibility can be derived if the client and the service provider left such intervention to the underlying implementation of the dispatch process – the request broker provided by middleware. Then the overheads associated with preparing for potential mismatches [11] could be omitted if the targets are near and have similar or identical signatures.

Reflecting middleware's flexibility in language's typing. Today's programming language specifications over-specify the interpretation of method-call, to the detriment of the software community in general. In systems that rely heavily on redirection via proxies, it would be more accurate for the programming language to specify only the semantics of the behavior occurring between entry and call, leaving the definition of a call's resolution to middleware. Language specification today is caught in a bind – to keep dispatch specification simple, the type systems generally require too much knowledge of the implementation structure to which a call is directed. Flexibility can be gained if they instead focused on accepting an indication from middleware about the safety of calling a method and propagating that information throughout the client. If the language specification simply carries forward a decision about the existence of implementation rather than trying to specify the matching rules, more flexible typing systems can be accommodated than those that require knowledge of the details of class implementations prior to run-time. The dispatch middleware then has flexibility in inserting conversions and rearrangements of the parameters, and even of employing different name-matching rules. What is required is a formal statement of the middleware's constraints, perhaps similar to the rule we propose: "the set of methods available to a client in a transaction is static or grows monotonically or the transaction fails."

Run-time selection of object classes. This same locus of intelligence applies when objects must be created. In traditional software, the implementing class of a new object is selected at the time a client is developed, generally after the developer inspects specifications for alternative implementations. With a service request broker, the client indicates what kind of object is needed and what methods must be made available for this kind of object. The "kind" is indicated by its classifier, with locally-defined meaning that allows individual services to describe subtyping relationships among different kinds of objects. Kinds of objects that support the same methods may still fall into different classifications because they attach different meanings to them. The focus is on characterizing the kind of object and the methods needed instead of on the implementing class. This allows the actual implementation class to be selected contextually, at run-time, by the service broker using new or local alternatives that may not have been known or available to the client's developer. It is also possible to augment the set of methods needed after objects have been created. This augmentation is known as *service-finding* and extends the idea of "down casting" in more familiar languages. When successful, the type system treats the knowledge that the methods are safe to call as if they were known to be available from creation.

Accommodation of service-substitution protocols. In mobile computing, and even in dynamically evolving system structures, it can be necessary to determine when one provider of services can be dynamically substituted for another. A contact for banking or travel information services may, for example, change in crossing regional boundaries. Or, some service providers may be more reliably reachable within one local region than another. The introduction of an intelligent service bus allows these issues to be addressed in a more organized and more easily maintained fashion than do proxies. Service providers interact with the service request broker when they are attached, and may provide information that helps determine their dynamic interchangeability.

4.2 Parallelism and Multi-core Support

4.2.1 Expressing Transactional Needs

If programming language design is to confront the issue of increased exploitation of parallel architectures, whether in distributed services or in multi-core machines, the ability to clearly delineate the transactional boundaries of failure of a concurrent element within must be provided. This is in addition to making provision for tracking the interferences of concurrent access to shared

data. While threads and transactions model the concurrency itself they have a natural intersection for the handling of errors. But programming languages generally have neither constructs to establish the boundaries of transactions within the execution nor a definition of their relationship. As with dispatch, the line between transaction model and transaction denotation needs to allow the detailed meaning and the implementation of transactions to be left to the middleware, but still permit the assignment of work to transactions to be expressed syntactically by the developer in a clear and direct manner. All execution takes place within a transaction. For convenience sake, the transaction in which an interpretation is taking place is best passed implicitly from caller to called method, as the thread is in familiar languages. But the language needs to allow for it to be explicitly specified on occasion. Explicit provision may be in the form of the creation of a new transaction or of the resumption of an existing one. While not proposing that the broadcast service model specify or restrict the transaction model unduly, it would be in line with many common specifications of transaction semantics for the transaction to be passed as an implicit parameter from client to service as method calls are made. Some provision must be made for changing the transactional context at the point of call. One possibility is to allow explicit change in the transaction context by using no-longer necessary syntactic position of the target object at the point of method call.

4.2.2 Static Enforcement of Task Commitments

One of the obstacles to greater exploitation of parallel structures is the fact that programming languages maintain commitment and failure response in a dynamic manner, using run-time interpretation to respond by waiting for the service to return. In addition to inhibiting parallelism, dynamic tracking is resource-intensive. Avoiding the need to hold resources has led to the exploitation of so-called "stateless services" in service-oriented architectures. But stateless services have no expression of flow dependencies between them. Since each service is finished before the next service acts, there is no way to express dependency on success or failure of later services, or to indicate back-out and recovery mechanisms. One common alternative is to combine them with separate process-flow specifications. While process-flow specification may be suitable for the niche in which service-oriented architectures operate, trying to use it to address the need for the widespread exploitation of parallelism called for by multi-core proponents is unlikely to succeed because it splits the specification into two language paradigms. At a coarse grain this may be acceptable, but at fine granularity it imposes too much intellectual and bookkeeping burden on the programmer. The service broker allows us to integrate process-flow more tightly into the usual programming language structures, in a way that allows a single program developer to exploit it easily.

5. Related Work

5.1 Broadcast Models

We are advocating the use of a broadcast model for method call to substantially improve the malleability of software. Broadcast models for processing have a long history of their own. One family of broadcast models center on a shared data-store. In that context, emphasis has been put on the use of *coordination languages*, like Linda [2]. Linda and subsequent tuple-space coordination systems provide primitives for controlling access by concurrent processes to a shaped data space of tuples. The point of intersection with message processing is that the database reading operations can wait for the appearance of a tuple matching an abstract template[13]. The effect of being able to wait for the appearance of a tuple matching an abstract template is much the same as the effect of a concurrent multiple dispatch, but the emphasis in Linda-based systems is in applications like data-mining, supported by a persistent tuple-store. We seek a replacement for the method call mechanism to remove layers of bookkeeping from clients and encourage malleability of software. The use of a separate coordination language or framework on top of the native language in a client scarcely makes it clearer or more malleable. In view of the performance overhead associated with Linda's point-of-view[13], other broadcast systems focus more on the delivery of ephemeral messages rather than a replacement for method call.

Non-storage-based systems, often called *message brokers*, play an important role in commercial systems, supported by products like IBM WebSphere MQ[17]. The primary advantage of such publish/subscribe systems is that the clients and servers need not be modified when message routing specifications change. Message frameworks like Java's JMS[15] have also been specified, but generally require the client to make static advance decisions about routing by specifying a class of object, like Topic, that manages where messages are sent. With these systems, the client typically uses a cumbersome framework that often involves data format conversions. This interferes with the transparency and malleability which is our goal. Academic interest in message brokers is thin, except when viewed as multi-methods.

5.2 Multi-methods

Methods that may be dispatched on the basis of the types of more than one argument are generally called multi-methods and are an area of significant and long interest. Although it is an object-oriented language, the invocation construct provided by CLOS [7] provides for structural abstraction to shield the client from the structure of choice-making in the implementation. But CLOS does not provide static typing, and its use was limited to the LISP community for many years. An excellent recap of work on multi-methods is given in [10]. It is important to note that most of this work is directed at achieving multiple-dispatch in languages that permit declarative typing, and not at hiding the dispatch criteria and implementation structure from clients, and they generally sacrifice either conditional safety or structural abstraction to do so.

Programming language research generally exploits multi-methods to specialize the behavior of methods for various argument types. This has led to thorough investigation of issues of ambiguity. In the absence of a modularizing structure larger than classes (like OSGI's "bundles"), restrictions that reduce the malleability of software have been used to introduce *modular* type-checking[10]. Pursuing malleability, we have exploited the concept of a language-defined service to bound the scope of possible ambiguity in addition to providing a separate container for service state.

5.3 Aspects

In addition to providing a malleable component construct, services can serve as aspects as well. Symmetric approaches to aspect-oriented software separate the materialization of the aspects, which contain state and method definitions that extend the semantics of objects from the expression of the pointcut specifications that indicate when they should be employed. FuseJ[12] provides a unified aspect/component model with these capabilities. With symmetric aspects, the pointcut or aspect interaction language is separated from the programming language and, in our case, expressed when services are introduced to the service request broker.

6. Summary

To improve the malleability of software, we have employed a model of method call in which a method's caller can safely declare and call methods without knowledge of which object(s) or service(s) implement the method. We exploit component model in which *services* form coherent collections of classes and manage

their supporting state. Method call is treated as a broadcast in which a call can be dispatched to multiple services through the operation of a service-request broker. Services can be composed locally or they can be mobile or distributed. The structure of the services used by a client is fluid and transparent, and the overhead of conversion or marshalling takes place in the broker and is avoided when components are locally supplied. The state associated with an object can be distributed across several services, much as it can be distributed across several aspects, enabling the services to be used as symmetric aspects. In addition to increasing the malleability of software, the service-broker construct permits the integration of the concept of transaction into method call semantics. Expressing transaction boundaries in the programming language permits an extension of the concept of responsibilities for future action. It is possible to use the static type system of a programming language to enforce the future execution of a method after the return from a method which commits to that future execution. This facility can be used to provide greater parallelism when such methods are called in loops.

Acknowledgements

I would like to thank the reviewers and especially Eric Eide for suggestions that have substantially improved this presentation.

References

[1] Filman, R.E. and D.P. Friedman: Aspect-Oriented Programming is Quantification and Obliviousness. *In: Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Minneapolis, MN, October 2000

[2] Gelerntner, D., Carriero, N, Coordination Languages and their Sgnificance, *Communications of the ACM*, *35,2*, (February, 1992), pp. 97-107

[3] Harrison, W., Lievens, D., Walsh, T.., Achieving Recombinance to Improve Modularity. Software Structures Group Report 102, October, 2006, available from https://www.cs.tcd.ie/research_groups/ssg

[4] Harrison, W. and Ossher, H., Subject-Oriented Programming - A Critique of Pure Objects, *In Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

[5] Harrison, W., Ossher, H., and Tarr, P., Software Engineering Tools and Environments: A Roadmap, in *Future of Software Engineering*, Anthony Finkelstein (Ed.), ACM Press, June 2000

[6] Harrison, W.. De-constructing and Re-constructing Aspect-Orientation, *In Proceedings of the Seventh Annual Workshop on Foundations of Aspect Languages*, Brussels, Belgium, 1 April, 2008, edited by Gary T. Leavens , ACM Digital Library, 2008, pp. 43-50

[7] Keene S., *Object-Oriented Programming in Common Lisp,* Addison-Wesley, 1989

[8] Lievens, D., Harrison, W.. Symmetric encapsulated multi-methods to abstract over application structure, *In Proceedings of the 24th Annual ACM Symposium on Applied Computing, Symposium on Applied Computing*, Honolulu, HI, March 8-12, 2009, ACM, 2009, pp. 1873 - 1880

[9] Lievens, D., Walsh, T., Dahlem,, D. Harrison, W.. Promoting Evolution Through Abstraction Over Implementation Structure, *In Proceedings Companion of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 16-19, 2009.

[10] Millstein, T., and Chambers, C. Modular Statically Typed Multimethods. in *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP 99)*, Lisbon, Portugal, June 14-18, 1999

[11] Mitchell, N., Sevitsky, G., and Srinivasan, H., Modeling Runtime Behavior in Framework-Based Applications, in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 06)*, Nantes, France

[12] Suvee, D., De Fraine,B., and Vanderperren, W., A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development*, in Component-Based Software Engineering*, LNCS 4063, Springer, Berlin / Heidelberg, 2006

[13] Wells, G., Coordination Languages: Back to the Future with Linda, Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05), pp. 87-98, 2005.

[14] Continuum Language Specification, available from https://www.cs.tcd.ie/research_groups/ssg

[15] Sun Java Message Service (JMS), http://java.sun.com/products/jms/, retrieved 24 Jan 2010

[16] Web Services Choreography Description Language, http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/, retrieved 24 Jan 2010

[17] WebSphere Message Broker Technical Overview, http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/topic/com.ibm.etools.mft.doc/ab20551_.htm, retrieved 24 Jan 2010.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
32	978-3-86956-037-3	**STG Decomposition: Internal Communication for SI Implementability**	Dominic Wist, Mark Schaefer, Walter Vogler, Ralf Wollowski
31	978-3-86956-036-6	**Proceedings of the 4th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
30	978-3-86956-009-0	**Action Patterns in Business Process Models**	Sergey Smirnov, Matthias Weidlich, Jan Mendling, Mathias Weske
29	978-3-940793-91-1	**Correct Dynamic Service-Oriented Architectures: Modeling and Compositional Verification with Dynamic Collaborations**	Basil Becker, Holger Giese, Stefan Neumann
28	978-3-940793-84-3	**Efficient Model Synchronization of Large-Scale Models**	Holger Giese, Stephan Hildebrandt
27	978-3-940793-81-2	**Proceedings of the 3rd Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
26	978-3-940793-65-2	**The Triconnected Abstraction of Process Models**	Artem Polyvyanyy, Sergey Smirnov, Mathias Weske
25	978-3-940793-46-1	**Space and Time Scalability of Duplicate Detection in Graph Data**	Melanie Herschel, Felix Naumann
24	978-3-940793-45-4	**Erster Deutscher IPv6 Gipfel**	Christoph Meinel, Harald Sack, Justus Bross
23	978-3-940793-42-3	**Proceedings of the 2nd. Ph.D. retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
22	978-3-940793-29-4	**Reducing the Complexity of Large EPCs**	Artem Polyvyanyy, Sergy Smirnov, Mathias Weske
21	978-3-940793-17-1	**"Proceedings of the 2nd International Workshop on e-learning and Virtual and Remote Laboratories"**	Bernhard Rabe, Andreas Rasche
20	978-3-940793-02-7	**STG Decomposition: Avoiding Irreducible CSC Conflicts by Internal Communication**	Dominic Wist, Ralf Wollowski
19	978-3-939469-95-7	**A quantitative evaluation of the enhanced Topic-based Vector Space Model**	Artem Polyvyanyy, Dominik Kuropka
18	978-3-939469-58-2	**Proceedings of the Fall 2006 Workshop of the HPI Research School on Service-Oriented Systems Engineering**	Benjamin Hagedorn, Michael Schöbel, Matthias Uflacker, Flavius Copaciu, Nikola Milanovic
17	3-939469-52-1 / 978-3-939469-52-0	**Visualizing Movement Dynamics in Virtual Urban Environments**	Marc Nienhaus, Bruce Gooch, Jürgen Döllner

ISBN 978-3-86956-043-4
ISSN 1613-5652