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

March 17, 2003

Held in conjunction with the Second International Conference on Aspect-Oriented Software Development (AOSD 2003)

Boston, Massachusetts

College of Computer and Information Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, Massachusetts 02115

NU-CCIS-03-03

Yvonne Coady, Eric Eide, and David H. Lorenz (Eds.)

# The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)

March 17, 2003

A one-day workshop held in conjunction with the
*Second International Conference on Aspect-Oriented Software Development (AOSD 2003)*
March 17-21, 2003, Boston, Massachusetts

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.

Building on the ACP4IS meeting at AOSD 2002, 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.

**Organizing Committee**

Yvonne Coady (University of British Columbia)
Eric Eide (University of Utah)
David H. Lorenz (Northeastern University)

**Acknowledgments**

Many thanks to Maja D'Hondt and Jeff Gray, the Workshop Co-Chairs at AOSD 2003; to Richard van de Stadt, the author of CyberChair; and to Matthew Flatt, Carsten Pfeiffer, and Jan Wloka for external reviewing.

**Program Committee**

Elisa Baniassad (Trinity College)
Don Batory (University of Texas at Austin)
Yvonne Coady (University of British Columbia)
Pascal Costanza (University of Bonn)
Krzysztof Czarnecki (University of Waterloo)
Eric Eide (University of Utah)
Dawson Engler (Stanford University)
Andy Gokhale (Vanderbilt University)
Stephan Herrmann (Berlin Technical University)
Wilson Hsieh (University of Utah)
David H. Lorenz (Northeastern University)
Renaud Pawlak (University of Lille)
Mario Südholt (École des Mines de Nantes)
Jan Vitek (Purdue University)
Jonathan Walpole (OGI)

# Table of Contents

# Lock inference for systems software

John Regehr Alastair Reid
School of Computing, University of Utah
{regehr, reid}@flux.utah.edu

## ABSTRACT

We have developed task scheduler logic (TSL) to automate reasoning about scheduling and concurrency in systems software. TSL can detect race conditions and other errors as well as supporting *lock inference*: the derivation of an appropriate lock implementation for each critical section in a system. Lock inference solves a number of problems in creating flexible, reliable, and efficient systems software. TSL is based on a notion of asymmetrical preemption relations and it exploits the hierarchical inheritance of scheduling properties that is common in systems software.

## 1. INTRODUCTION

Embedded systems, operating systems, and Internet servers are fundamentally concurrent because they must respond to external events in real time. For people developing these systems, critical sections can be considered to be a functional aspect of software: they are used to maintain high-level program invariants. The implementation of critical sections, on the other hand, is a non-functional aspect — it affects response time and throughput.

In this paper we take the position that locks in systems software, which are usually named by referring to an instance of a particular lock implementation, should be specified at a higher level of abstraction. At system build time a whole-program analysis should be used to infer an appropriate lock implementation for each critical section. This has a number of benefits that can lead to the creation of robust, reusable systems software:

- Developers need not learn the complex rules that govern locking in systems software. For example, threads synchronize with interrupts by disabling interrupts, interrupts must not block, and non-preemptive event handlers are implicitly synchronized.
- Code maintenance and modification is made easier and less prone to bugs. For example, if an event handler is broken out into a preemptive thread to ensure that its response-time requirements can be met, resources that it shares with other handlers, which previously did not need protection by a lock, must now be protected. These resources can be automatically detected be TSL.
- In many cases a generic component, which implements correct locking, is instantiated in such a way that its locks serve no useful purpose, e.g., because it is accessed only by a single thread or because a component higher in the call graph provides sufficient serialization. In this case the locks can be dropped as an optimization.
- When the analysis finds a critical section where no available lock implementation works, a race condition has been detected and this should be brought to a developer's attention.
- Locks can be selected in such a way that their global side effects are desirable. For example, in a system where throughput is important, it might be the case that all locks should be implemented by disabling interrupts since this is very efficient. In another system where real-time deadlines must be met, it may be unacceptable to disable interrupts for more than a few microseconds because this delays unrelated, time-critical processing.
- Software components can be developed that are agnostic with respect to the execution environments in which they are instantiated. This is desirable because the environments in which a component executes depend on the call graph for a particular system. In effect, the execution environments in a system cross-cut its traditional modular decomposition.

These benefits are provided by task scheduler logic (TSL), a novel formalism for integrated reasoning about scheduling and concurrency in systems software. The key idea behind TSL is that the hierarchical inheritance of scheduling properties, when combined with modular specifications of schedulers and locks, can be used to formalize the rules that govern locking in systems software. These rules — previously informal and unchecked — are caused by complex relationships between multiple *execution environments*: software contexts such as kernel-supported threads, user-level threads, interrupts handlers, and event loops. Furthermore, as a side effect of deriving and checking rules about synchronization, it becomes possible to perform *lock inference*: the derivation of an appropriate lock implementation for each critical section.

Lock inference can be viewed as the top of a stack of useful capabilities for manipulating and analyzing concurrency aspects of systems software. At the bottom of the stack is externally visible and parameterizable locking — lock analysis and inference are difficult if locks are hard-wired into code. Many component systems, such as our Knit [10], have this capability. In the middle of the stack is the capacity to detect concurrency errors, which depends on a model of resources and concurrency as well as a mechanism for tracking the call graph. Many systems have done this, but TSL is the first we are aware of that can find concurrency problems in systems software where there are diverse execution environments. Finally, at the top of the stack is the ability to automatically infer an appropriate implementation for each critical section in a system. This is the primary contribution of TSL.

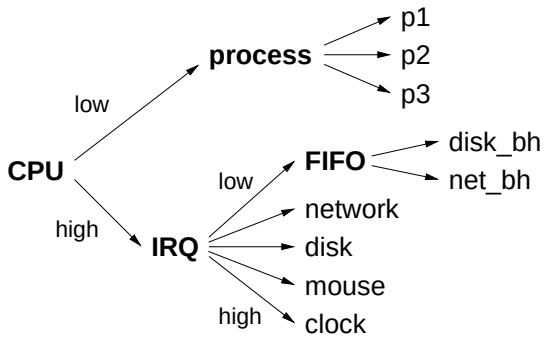



**Figure 1: A generic UNIX scheduling hierarchy**

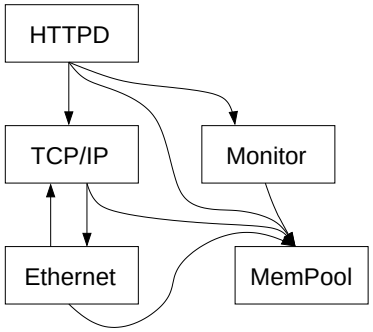



**Figure 2: A simple component-based monitoring system**

# 2. BACKGROUND

This section describes the hierarchical scheduling concepts that underlie TSL, the difficulties of creating component-based systems that motivate our work, and the lightweight program analysis that is a prerequisite for using TSL.

## 2.1 Hierarchical Scheduling

At a coarse granularity, the flow of control in a software system is determined by its schedulers. In this paper a task is a schedulable flow of control and a scheduler is any piece of software (or hardware) that controls the order of execution of tasks. Properties are imparted to a task by each scheduler that it runs under. For example, an interrupt handler cannot block and it is preemptively scheduled at higher priority than any user-mode code. If an event-processing loop is run in interrupt context, then event handlers scheduled by the loop inherit event properties, such as non-preemptive execution relative to other event handlers, in addition to all interrupt properties. The schedulers in a system create a variety of execution environments, each of which has its own rules for structuring code, sequencing operations, and interacting with other environments.

Figure 1 depicts the scheduling hierarchy for a typical UNIX-like operating system. The top-level scheduler, CPU, is implemented in hardware; it runs interrupts whenever possible and user-mode code otherwise. The IRQ scheduler preemptively schedules hardware interrupt handlers based on their priorities, as well as a software interrupt handler at the lowest priority. The software interrupt handler runs a FIFO scheduler that runs deferred bottom-half handlers disk_bh and net_bh with run-to-completion semantics. The process scheduler is the standard preemptive UNIX timesharing scheduler; it runs processes p1..p3.

We have found the hierarchical scheduling notation shown in Figure 1 to be quite useful and general for describing the execution environments provided by systems. For example:

- Linux, Windows 2000 [12], and most real-time operating systems are minor variations on the same theme.
- RTLinux [13] adds an additional level of scheduling above the CPU scheduler by virtualizing the interrupt handling structure of Linux.
- TinyOS [6] has no thread or process scheduler: its scheduling hierarchy includes only interrupts and an event loop.
- Internet servers, Java virtual machines, and other application-level systems software extend the scheduling hierarchy by implementing event loops, thread pools, and user-level threads.

TSL provides a uniform notation for modeling these and other collections of execution environments.

## 2.2 Component Based Systems Software

Figure 2 depicts the software for an embedded application that is designed to (1) monitor a system such as a pumping station on a remote section of an oil pipeline and (2) make information about the system available to HTTP clients. Upon request the $HTTPD$ component retrieves data from the $Monitor$ component and sends it out on the network using the $TCP/IP$ and $Ethernet$ components. All components make use of a memory allocator.

Although TSL applies generally to systems software, and makes no assumptions about the underlying component model, it is especially useful for analyzing component-based systems software. First, component software tends to expose interfaces for locking, making it easier to analyze and parameterize synchronization behavior. Second, component-based software is often hard to understand due to its many indirect connections between software modules. This complexity interacts poorly with the multiple execution environments that are created by a hierarchy of schedulers such as the one in Figure 1. For example, assume that the $MemPool$ component in Figure 2 reports an out-of-memory condition using a logging interface that is connected to a storage component (not shown). The storage component uses a thread mutex to protect its internal data structures. Since $MemPool$ can be called by the $Ethernet$ component while executing in interrupt context, the interrupt handler can indirectly attempt to acquire the mutex, leading to a system crash because interrupts are not permitted to acquire mutexes. This bug will not be obvious to a developer who simply wants to reuse these components and who does not have a detailed understanding of their internals. Furthermore, this bug will be very difficult to expose through testing since the allocator rarely runs out of storage. In our experience, creating correct systems using components like the ones in this example requires near-expert knowledge about component internals. Clearly there is room for improvement.

## 2.3 Analyzing Systems

TSL requires static identification of tasks, schedulers, resources, critical sections, and the call graph for a program or system. There

are well-known techniques for obtaining this information; in practice we expect that a combination of annotations and language-based program analysis will be used. For example, in our prototype implementation (see Section 6) we learn about resources using annotations and obtain an approximation of the call graph by analyzing the component linking graph.

# 3. TASK SCHEDULER LOGIC

This section provides an overview of TSL.

## 3.1 Tasks and Schedulers

Tasks are sequential flows of control through a system; they are the fundamental unit of reasoning in TSL. Each task has a well-defined entry point and many tasks also finish by returning control to the scheduler that invoked them. Other tasks encapsulate an infinite loop and these never finish — control only returns to their scheduler through preemption. Throughout this paper the variables $t$, $t_1$, etc. range over tasks.

Schedulers are modeled in a modular way by specifying the preemption relations that the scheduler induces between tasks that it schedules. Preemption relations are represented asymmetrically: we write $t_1 \not\downarrow t_2$ when task $t_2$ can preempt task $t_1$. That is, if $t_2$ can start to run after $t_1$ begins to execute but before $t_1$ finishes.

The simplest scheduler, a non-preemptive event scheduler, does not permit any child to preempt any other child. For any two children $t_1$ and $t_2$ of such a scheduler, $\neg(t_1 \not\downarrow t_2) \wedge \neg(t_2 \not\downarrow t_1)$.

On the other hand, a preemptive scheduler, such as a UNIX time-sharing scheduler, potentially permits each child task to preempt each other child task. That is, for any two children of such a scheduler, $t_1 \not\downarrow t_2 \wedge t_2 \not\downarrow t_1$.

A third type of scheduler commonly found in systems software is a strict priority scheduler such as the interrupt controller in a typical PC. It schedules a number of tasks $t_1..t_n$ and it is the case that $t_j \not\downarrow t_i$ when $i < j$.

## 3.2 Resources, Races, and Locks

At each program point a task is accessing some (possibly empty) set of resources. The variables $r$, $r_1$, etc. range over resources, and we write $t \rightarrow r$ if a task $t$ uses a resource $r$. Resources represent data structures or hardware devices that must be accessed atomically.

A race condition may occur if task $t_1$ can be preempted by $t_2$ at a point where both tasks are accessing a common resource. Problematic preemption relations can be eliminated using locks; at each program point a task holds a (possibly empty) set of locks. We write $t_1 \not\downarrow_L t_2$ if parts of a task $t_2$ that hold a set of locks $L$ can start to run while a task $t_1$ holds $L$. For example, consider two threads that can usually preempt each other. If holding a thread lock $lk$ blocks a task $t_2$ from entering critical sections in $t_1$ protected by $lk$, then $(t_1 \not\downarrow_\varnothing t_2) \wedge \neg(t_1 \not\downarrow_{lk} t_2)$.

Every lock is provided by some scheduler; the kinds of locks provided by a scheduler are part of its specification. We write $t \multimap l$ if a scheduler $t$ provides a lock $l$, and require that each lock be provided by exactly one scheduler. There are two common kinds of locks. First, locks that resemble disabling interrupts: they prevent any task run by a particular scheduler from preempting a task that holds the lock. Second, locks that resemble thread mutexes: they only prevent preemption by tasks the hold the same instance of the type of lock.

Locks satisfy three important properties. First, if $t_1$ can be preempted while holding a set of locks, then $t_1$ can be preempted while holding fewer locks:

$$t_1 \not\downarrow_{L_1} t_2 \;\wedge\; L_1 \supseteq L_2 \;\Rightarrow\; t_1 \not\downarrow_{L_2} t_2$$

Second, if $t_1$ can be preempted by $t_2$ while holding either a set of locks $L_1$ or a set of locks $L_2$, then $t_1$ can be preempted by $t_2$ while holding both sets of locks.

$$t_1 \not\downarrow_{L_1} t_2 \;\wedge\; t_1 \not\downarrow_{L_2} t_2 \;\Rightarrow\; t_1 \not\downarrow_{L_1 \cup L_2} t_2$$

Finally, preemption is a transitive relation: if $t_1$ can be preempted by $t_2$ and $t_2$ can be preempted by $t_3$, then $t_1$ can be preempted by $t_3$.

$$t_1 \not\downarrow_{L_1} t_2 \;\wedge\; t_2 \not\downarrow_{L_2} t_3 \;\Rightarrow\; t_1 \not\downarrow_{L_1 \cap L_2} t_3$$

The definition of a race condition is as follows:

$$\begin{aligned} \mathit{race}(t_1, t_2, r) \stackrel{\text{def}}{=}\; & t_1 \rightarrow_{L_1} r \\ \wedge\; & t_2 \rightarrow_{L_2} r \\ \wedge\; & t_1 \neq t2 \\ \wedge\; & t_1 \not\downarrow_{L_1 \cap L_2} t_2 \end{aligned}$$

That is, a race can occur if two tasks $t_1$ and $t_2$ use a common resource $r$ with some common set of locks $L_1 \cap L_2$, and if $t_2$ can preempt $t_1$ even when $t_1$ holds those locks. For example, if some task $t_1$ uses a resource $r$ with locks $\{l_1, l_2, l_3\}$ and another task $t_2$ uses $r$ with locks $\{l_2, l_3, l_4\}$ then they hold locks $\{l_2, l_3\}$ in common and a race occurs iff $t_1 \not\downarrow_{\{l_2, l_3\}} t_2$.

## 3.3 Hierarchical Scheduling

Each scheduler is itself a task from the point of view of a scheduler one level higher in the hierarchy. For example, when an OS schedules a thread, the thread is considered to be a task regardless of whether or not an event scheduler is provided by the thread. We write $t_1 \lhd t_2$ if a scheduler $t_1$ is directly above task $t_2$ in the hierarchy; $\lhd$ is the *parent* relation. Similarly, the *ancestor* relation $\lhd^+$ is the transitive closure of $\lhd$.

TSL gains much of its power by exploiting the properties of hierarchies of schedulers. First, the ability or lack of ability to preempt is inherited down the scheduling hierarchy: if a task $t_1$ cannot preempt a task $t_2$, then $t_1$ cannot preempt any descendent of $t_2$. A consequence is that if the *nearest common scheduler* in the hierarchy to two tasks is a non-preemptive scheduler, then neither task can preempt the other. This is a useful result when showing, for example, that a lock is not necessary to protect a resource that is accessed by a particular composition of components.

When a task that is the descendent of a particular scheduler requests a lock, the scheduler may have to block the task. It does this not by directly blocking the task, but by blocking its currently running child, which must be transitively scheduling the task that requested the lock. If a task attempts to acquire a lock that is not provided by one of its ancestors in the scheduling hierarchy then there is no child task for the scheduler to block — an illegal action has occurred. Using TSL we can check for a generalized version of the "blocking in interrupt" problem by ensuring that tasks only acquire blocking locks provided by their (possibly transitive) parents in the

scheduling hierarchy. We formalize this generalization as follows:

$$\begin{aligned} \mathit{illegal}(t, l) \stackrel{\text{def}}{=} \exists\, t_1 .\quad & t_1 \multimap l \\ \wedge\quad & \neg(t_1 \lhd^{+} t) \\ \wedge\quad & t \rightarrow_L r \\ \wedge\quad & l \in L \\ \wedge\quad & \mathit{blocking}(l) \end{aligned}$$

### 3.4 Using TSL

Software developers, who compose systems using new and existing components, do not need to directly interact with TSL. Rather, they create software as usual, but in addition to protecting critical sections with locks, they have the option of using a *virtual lock* that is a cue for TSL to infer an appropriate lock implementation. Developers who create new schedulers will need to specify their properties in TSL, but we expect that these programmers will be in the minority: most will reuse an existing scheduler and its attached TSL specification.

## 4. LOCK INFERENCE

Many of the benefits of TSL are provided by its ability to infer an appropriate lock implementation for each critical section. Recall that a lock assignment is legal if the lock is not a blocking lock or if it is provided by an ancestor of the task that contains the critical section. A brute-force algorithm for synchronization inference is to enumerate all legal assignments of locks to critical sections; the enumeration can stop once an assignment is found that eliminates all race conditions. If no such assignment exists, then there is a genuine race condition and the system cannot be built. No special algorithmic support for the elimination of unnecessary synchronization is required because synchronization inference subsumes synchronization elimination. It suffices to ensure that one of the locks available to each critical section is the "null lock" that has no effect on preemption relations and is implemented as a NOP.

We currently use the brute-force algorithm to assign lock implementations to critical sections. Although it is tractable for systems that we have analyzed, we expect that we will want to develop improved algorithms. One avenue for improvement is to exploit qualities of the domain. For example, the search space can be narrowed by observing that it is probably not useful to attempt to use a different kind of lock, or a different instance of the same kind of lock, to protect different critical sections that access the same resource. In addition, for each resource the set of legal locks should be tried in an intelligent order, probably starting with a "strong" lock, like disabling interrupts, that eliminates many preemption relations. Another way to improve performance might be to cast the lock inference problem as an instance of the boolean satisfiability problem, for which very fast solvers exist [8].

Once a lock assignment that eliminates all races is found it may be desirable to optimize the choice of locks. Such optimization is outside the scope of TSL, which has no mechanism for preferring one lock assignment over another as long as both of them produce a system that is free of race conditions. In general, there is a tension between choosing an efficient lock for each critical section and picking locks that avoid unnecessarily delaying the execution of unrelated tasks.

## 5. REAL-TIME CONCERNS

Since lock choice has a pervasive effect on system performance, we plan to integrate synchronization inference with SPAK, a real-time scheduling tool that we have developed [9]. The negative effects that locks have on real-time tasks can be quantified by adding *blocking terms* — periods of time during which certain scheduling decisions cannot be made — to the schedulability analysis equations [11]. If a lock resembles disabling interrupts, it contributes blocking terms to all tasks run by the scheduler providing the lock. On the other hand, a lock that resembles a mutex contributes blocking terms only to tasks that may attempt to acquire the same lock. Blocking terms, like preemption relations, are inherited down the scheduling hierarchy.

Besides returning a binary result about overall system schedulability, SPAK can perform several useful functions that interact well with TSL. First, it can evaluate the *robustness* of a software system under timing faults: tasks that run for longer than their nominal worst-case execution times. This is useful because it can help TSL avoid creating systems that are brittle in the sense that a small perturbation in task execution will cause real-time deadlines to be missed. Second, SPAK has the capability to map a large number of design-time tasks onto a smaller number of run-time threads; this is useful for resource-constrained embedded systems because it reduces the amount of memory devoted to thread stacks and the amount of CPU time spent performing context switches. Synchronization inference and thread minimization interact favorably because strongly coupled collections of tasks, when aggregated into a single thread, will enable many locks to be eliminated. These tasks should be preferred targets for thread minimization when compared to collections of tasks that permit few locks to be eliminated.

## 6. APPLYING TSL

We have implemented a prototype TSL checker based on a forward-chaining evaluator: it takes a specification for a system and derives all possible consequences of the TSL axioms. Systems specified in TSL are finite; there are a limited number of tasks and preemption relations between tasks.

Our test environment is based on the Knit [10] component language and the OSKit [4], a library of systems software components. We extended Knit slightly to accommodate annotations about resources and locks, and we used Knit's component linking graph to generate a safe (though crude) approximation of the call graph.

Figure 3 provides a more detailed look at the embedded, web-based monitoring system from Figure 2. The scheduling hierarchy for the system is shown on the left side of the figure and application components are shown on the right side. For simplicity we have omitted many infrastructure components. The full system consists of 190,000 lines of code, 116 components, 1059 functions, 5 tasks, and 2 kinds of locks.

Before we can analyze the system with TSL we must label all the tasks (we name them $h1$, $h2$, $t$, $e$, and $m$), label all the schedulers (we name them $CPU$, $thread$, $IRQ$, and $FIFO$), and generate a TSL specification for each scheduler (not shown). We must also add resources (named $rh$, $rb$, $rt$, $re$, $rmon$, and $rmem$) and add their uses into the callgraph, add locks (named $cli$ and $lk$), attach locks to the scheduler that provides them, and label edges in the callgraph with locks acquired before the calls are made. Figure 3 shows these labels and relations. The example includes some errors in the use of locks that we shall discover using TSL. The schedulers are, of course, also components, but to keep the example to a reasonable size we do not show their resources, locks used to protect those resources, calls to the functions they export, etc.

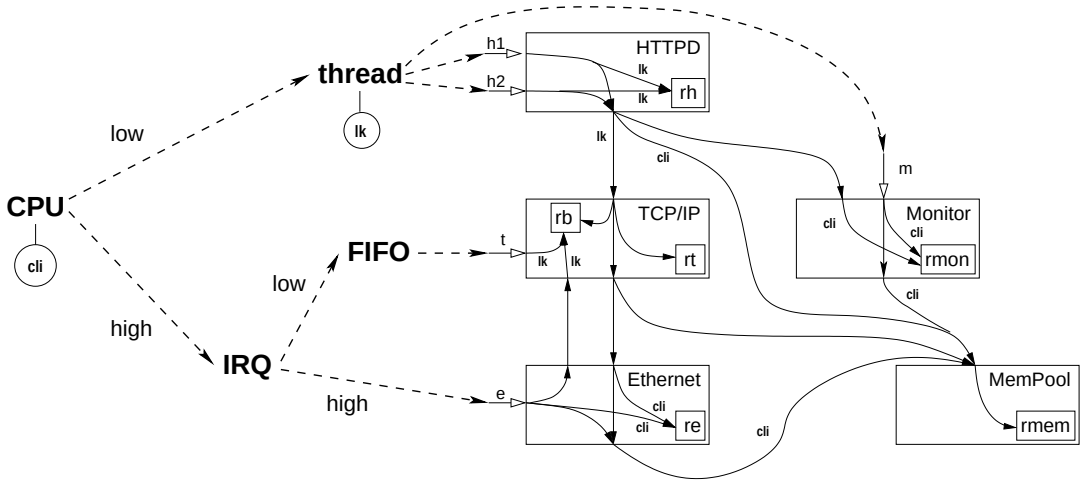



**Figure 3: A simple component-based monitoring system (right) and its scheduling hierarchy (left)**

## 6.1 Checking for illegal locking

To detect cases of illegal locking our implementation computes a list of all the resources accessed by each task with a given set of locks. For example, from the callgraph and locks shown in the figure we generate the following table:

$$\begin{array}{rll} h1, h2 & \rightarrow_{lk} & \{rh, rt, rb, rmem\} \\ h1, h2 & \rightarrow_{lk,cli} & \{re, rmem\} \\ h1, h2 & \rightarrow_{cli} & \{rmon, rmem\} \\ m & \rightarrow_{cli} & \{rmon, rmem\} \\ t & \rightarrow_{lk} & \{rb\} \\ e & \rightarrow_{lk} & \{rb\} \\ e & \rightarrow_{cli} & \{re, rmem\} \end{array}$$

Given this table and the knowledge that $lk$ is a blocking lock, it is straightforward to apply the definition of *illegal* to generate a list of all the illegal lock uses:

$$\begin{array}{l} \textit{illegal}(t, lk) \\ \textit{illegal}(e, lk) \end{array}$$

Both problems are caused by using the lock $lk$ to protect the resource $rb$ which is accessed by hardware and software interrupts. They can be resolved by changing the lock to $cli$.

Although these errors can be easily found by inspecting Figure 3, the real system has many more components and interconnections and is difficult to debug by inspection.

## 6.2 Checking for races

A race occurs when two tasks may access a resource simultaneously. TSL provides a list of potential race conditions and can be used to examine the scheduler hierarchy and call chain to diagnose the cause of problems.

For example, from the scheduler hierarchy we can deduce that the following preemption relations hold:

$$\begin{array}{rcl} h1, h2, m & \not\downarrow_{\varnothing} & h1, h2, m \\ h1, h2, m & \not\downarrow_{\varnothing} & t \\ t & \not\downarrow_{\varnothing} & e \end{array}$$

Combining this with resource use and the definition of *race*, we obtain the following race conditions.

$$\begin{array}{ll} \textit{race}(h1, h2, rmem) & \textit{race}(h2, h1, rmem) \\ \textit{race}(h1, m, rmem) & \textit{race}(h2, m, rmem) \\ \textit{race}(h1, e, rmem) & \textit{race}(h2, e, rmem) \end{array}$$

These can be fixed by acquiring the $cli$ lock when calling from $TCP/IP$ to $MemPool$.

## 6.3 Synchronization elimination and inference

The system in Figure 3 does not contain any redundant locks. However, consider what would happen if, due to memory constraints, the developer could only instantiate a single thread for the $HTTPD$ component. In this case the locks protecting $rh$ could be safely eliminated as could the thread lock providing atomic access to the top half of the $TCP/IP$ component.

All locks in our example refer to specific implementations. However, if the $cli$ locks in the $Monitor$ component in Figure 3 were declared as virtual locks then TSL would inform us that acceptable lock implementations are $cli$ and $lk$.

# 7. APPLICABILITY AND LIMITATIONS

TSL applies to static systems where tasks, schedulers, critical sections, and the call graph are known in advance. Although this is a good match for most embedded software we would like to extend TSL to handle systems with dynamic components. One way to do this would be to use static analysis or dynamic checking to bound the behavior of the dynamic part of the system. For example, if we guarantee that a particular resource cannot be accessed by a dynamic part of the system, then it is permissible to remove locks protecting this resource provided that this is otherwise a valid optimization. In general, tighter bounds on the behavior of the dynamic part of a system permit more effective analysis and optimization of the static part.

Although TSL cannot yet be used to check systems for risk of deadlock, we are exploring ways to permit this. If locks were represented as an ordered multiset, rather than as an unordered set, then TSL could be used to enforce an ordering on lock acquisitions, leading to a system that is guaranteed to be free of deadlock.

Furthermore, this would permit TSL to check for recursive lock acquisition — this is legal for some lock implementations but not for others.

## 8. RELATED WORK

Model checkers such as SPIN [7] and Bandera [1] represent a promising approach to bringing the benefits of concurrency theory to developers. Model checkers are more powerful than TSL in that they can reason about deadlock and liveness. However, TSL adds value over model checkers by specifically supporting the hierarchical inheritance of scheduling properties that occurs in systems software — this permits effective reasoning across multiple execution environments. Also, model checkers provide no support for lock inference.

The trend towards inclusion of concurrency in mainstream language definitions such as Java and towards strong static checking for errors is leading programming language research in the direction of providing annotations [2, 5] or extending type systems to model locking protocols [3]. These efforts are complementary to our work on reasoning about concurrency across execution environments; we believe that TSL and extended type systems would be a very powerful combination.

Early versions of our Knit toolchain [10] had a primitive mechanism for tracking top/bottom-half execution environments. It did not model locks and locking, but could check for the "blocking in interrupt" error that is particularly easy to make in component based systems. In the earlier version of Knit we could move components from one environment to another and check the resulting systems, but we could not add new execution environments or even model all of the environments in systems that we built.

## 9. CONCLUSION

TSL is a new logic that supports integrated reasoning about scheduling and concurrency; it supports lock inference as well as the detection of concurrency errors and elimination of redundant locking. Binding critical sections to lock implementations too early is the source of many problems in developing flexible, reliable, and efficient systems software. We believe that TSL, or something like it, is necessary to create next-generation software systems where components can be flexibly and correctly instantiated in a variety of execution environments.

## Acknowledgments

The authors would like to thank Eric Eide, Jay Lepreau, and the reviewers for providing valuable feedback on drafts of this paper.

This work was supported, in part, by the National Science Foundation under award CCR-0209185 and by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreements F30602-99-1-0503 and F33615-00-C-1696.

## 10. REFERENCES

[1] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Intl. Conf. on Software Engineering*, Limerick, Ireland, June 2000.

[2] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, CA, December 1998.

[3] Cormac Flanagan and Martin Abadi. Types for safe locking. In S.D. Swierstra, editor, *ESOP'99 Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag, March 1999.

[4] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, Saint-Malô, France, October 1997.

[5] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: Annotations and policy. In *Proc. of the 24th Intl. Conf. on Software Engineering*, pages 453–463, Orlando, FL, May 2002.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th ASPLOS*, pages 93–104, Cambridge, MA, November 2000.

[7] Gerard J. Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[8] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 39th Design Automation Conference*, Las Vegas, NV, June 2001.

[9] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. of the 23rd IEEE Real-Time Systems Symp.*, Austin, TX, December 2002.

[10] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 347–360, San Diego, CA, October 2000.

[11] Lui Sha, Ragunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[12] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[13] Victor Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, March 1999.

# Evolving an OS Kernel using Temporal Logic and Aspect-Oriented Programming*

Rickard A. Åberg*, Julia L. Lawall**, Mario Südholt*, and Gilles Muller*

*OBASCO group
École des Mines de Nantes/INRIA
44307 Nantes Cedex 3, France
{raberg,sudholt,gmuller}@emn.fr

**DIKU
University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

## Abstract

Modern operating systems such as Linux are large, complex pieces of software that are difficult to evolve, even when the modifications appear to be systematic. In this paper, we consider the problem of evolving Linux to support the Bossa framework for scheduler development. In this framework, a new scheduler connects to the kernel using events that are generated at specific *scheduling points*, which are scattered throughout the kernel. To automate the evolution of Linux to support Bossa, we use an aspect that describes how to instrument the kernel with event generation. This aspect uses rules that rely on temporal logic to identify the control-flow contexts in which the aspect should apply. In this paper, we present examples of rules that highlight the features of our approach.

## 1 Introduction

Bossa is a framework aimed at simplifying the design of kernel-level schedulers so that an application programmer can develop specific scheduling policies without expert-level operating system (OS) knowledge [1]. A Bossa scheduling policy is written in a Domain Specific Language that permits high-level safety properties to be statically verified [12]. The policy is compiled into a C file that is either linked statically with the kernel or installed dynamically as a kernel module. To achieve scheduler modularity, the policy is connected to the kernel using events (e.g., process creation, termination and un/blocking) that are generated at specific locations, *scheduling points*, in nearly all kernel services and drivers. These events are transmitted to the scheduling policy by the Bossa runtime system.

Bossa has been developed with the goal of independence from the kernel, and has been implemented in versions 2.2 and 2.4 of Linux. Currently, an OS kernel is prepared for use with Bossa by manually tracking down scheduling points according to informally-specified conventions and instrumenting these scheduling points with calls to macros that generate Bossa events. Even though this re-engineering needs to be done only once for a given kernel, performing this task manually is tedious and error-prone. The source code of the Linux 2.4 kernel exceeds 100MB. The current integration of Bossa into this version of Linux includes over 300 event generations and supports 25% of the available system services and 15% of the available drivers.

The wide distribution of scheduling points across the kernel indicates that scheduling as a concern crosscuts OS kernel code and that Aspect-Oriented Programming (AOP) should be useful to automate the integration of Bossa into an existing kernel. Nevertheless, common AOP techniques [2, 9], based on instrumenting function call and return points, are not sufficient for instrumenting kernel code with Bossa event notifications. Kernel coding conventions specify a precise sequence of instructions to carry out scheduling actions, and Bossa event notifications must be inserted at the level of these instructions. Furthermore, the choice of events is often determined by the structure of an entire such sequence of instructions rather than by a single operation. This advocates for an application-specific transformation system that addresses the structure of kernel code.

In this paper, we present preliminary work on an as-

*Appears in Proceedings of the Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2003), Boston, MA, March 17, 2003. This work was sponsored in part by Microsoft under contract 2003-146.

```
1   set_current_state(TASK_INTERRUPTIBLE);
a   BOSSA_BLOCK(MEM_DMAREAD,current);
2   add_wait_queue(&md->lynx->mem_dma_intr_wait, &wait);
3   run_sub_pcl(md->lynx, md->lynx->dmem_pcl, 2, CHANNEL_LOCALBUS);
b   BOSSA_CHECK_PENDING_SIGNAL(MEM_DMAREAD,current);
4   schedule();
5   while (reg_read(md->lynx, DMA_CHAN_CTRL(CHANNEL_LOCALBUS))
6             & DMA_CHAN_CTRL_BUSY) {
7             if (signal_pending(current)) {
8                   retval = -EINTR;
9                   break;
10            }
c             BOSSA_YIELD_SYSTEM_IMMEDIATE(MEM_DMAREAD,current);
11            schedule();
12        }
13  reg_write(md->lynx, DMA_CHAN_CTRL(CHANNEL_LOCALBUS), 0);
14  remove_wait_queue(&md->lynx->mem_dma_intr_wait, &wait);
```

Figure 1: Excerpt of the PCILynx driver after Bossa instrumentation inserted

pect system that allows advice to be woven at the level of granularity required for the integration of Bossa in an OS kernel. Our approach is in the spirit of Event-based AOP [4], in which crosscuts are defined in terms of both arbitrary events that occur during program execution and the relations between such events, thus generalizing AspectJ's pointcuts. Our main contribution consists of an aspect for kernel instrumentation expressed as a set of transformation rules that use formulas of temporal logic to precisely describe the sequences of source code instructions to which the rules apply.

The rest of this paper is structured as follows: Section 2 gives background information on scheduling in Linux and corresponding Bossa events. Section 3 shows how Bossa kernel instrumentation can be defined using rewrite rules based on temporal logic. Section 4 presents related work and Section 5 concludes.

# 2 Linux Scheduling Points

The heart of scheduling in Linux is the function `schedule()`, which preempts the running process and elects a new process from among those that are currently ready. In this paper, we focus on this preemption of the running process, although our approach is applicable to other scheduling actions. The effect of preemption depends on the current state of the preempted process. Three states are commonly used. `TASK_RUNNING` indicates that the process remains ready. `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE` indicate that the process blocks until explicitly awakened. A process in the state `TASK_INTERRUPTIBLE` can also be awakened by a timer or a signal. In particular, if a signal is pending for the process at the time of the call to `schedule()`, a process in the state `TASK_INTERRUPTIBLE` does not block at all; it remains ready as for a process in the state `TASK_RUNNING`. Because state change operations and calls to `schedule()` both influence how a new process is elected, we consider both kinds of operations to be scheduling points.

## 2.1 Bossa events in Linux

To illustrate the process of integrating Bossa into existing kernel code, we use an extract of the Texas Instruments IEEE1394 PCILynx driver for Linux 2.4.18, as shown in Figure 1 (code added for Bossa is shown in italics). Lines 1-4 cause the running process to block until the resource associated with the wait queue `md->lynx->mem_dma_intr_wait` becomes available. The loop between lines 5 and 12 causes the running process to repeatedly pause until it receives a signal or the condition of the `while` loop is no longer satisfied. This code contains three scheduling points: the setting of the state of the running process to `TASK_INTERRUPTIBLE` in line 1, and the calls to `schedule()` in lines 4 and 11.

In Linux, the setting of the state of a process to `TASK_INTERRUPTIBLE` amounts to a declaration that the process should block at the next call to `schedule()`, unless there is a pending signal. To inform the Bossa policy that the process should block, we insert a use of the `BOSSA_BLOCK` macro at this point (Figure 1, line a).

The treatment of a call to `schedule()` depends on the current state of the running process. At the call to `schedule()` in line 4, the state is known to be `TASK_INTERRUPTIBLE`. In this case, we insert a use of the `BOSSA_CHECK_PENDING_SIGNAL` macro (Figure 1, line b). This macro checks whether there is a signal

pending for the running process; if one is detected the policy is informed that the process should remain ready at the next call to `schedule()`. At the call to `schedule()` in line 11, the state of the running process is `TASK_RUNNING`; this is the state of a process on return from a call to `schedule()` (*i.e.*, in line 4 or line 11) and there is no intervening process state change. In this case, we insert a use of the `BOSSA_YIELD_SYSTEM_IMMEDIATE` macro (Figure 1, line c), which informs the policy to prepare to preempt the running process such that the process remains ready.

The Bossa events do not affect the algorithm expressed by the PCILynx driver code. Instead, they inform the Bossa scheduling policy of the state of the running process, so that the policy can use this information when it next elects a new process.

### 2.2 Automating Linux instrumentation

We now consider some issues that arise in automating the instrumentation process. Our analysis of the Linux kernel code suggests that an intraprocedural analysis is sufficient to detect patterns of scheduling points. Furthermore, in each case where a change to a process state follows such patterns, the affected process is the running process. Thus, we do not need to detect aliases between process references. The program patterns we consider always appear in one of only a few fixed forms. For example, the setting of the process state is expressed by either the direct assignment of a constant state value to the `state` field of the process, or by use of a macro, such as `set_current_state`, that has the same effect.[1] Thus, a dataflow analysis is not needed. Overall, these properties imply that automatic instrumentation need not be prohibitively expensive.

Every occurrence of `TASK_INTERRUPTIBLE` should to be instrumented with `BOSSA_BLOCK`. Thus, a transformation rule that only examines individual instructions is sufficient in this case. The instrumentation of a call to `schedule()`, however, depends on the state of the running process, which in turn depends on the set of updates to this state that can reach the call to `schedule()`. A static analysis to determine the set of such updates must potentially take many instructions into account, and these instructions can appear both before and after the given call to `schedule()`. For example, treatment of the call to `schedule()` in line 11 of Figure 1 requires considering both considering the code preceding the `while` loop and the entire loop body. We thus argue for a flow-sensitive method to express crosscuts.

[1] In this paper, we assume that `set_current_state` is always used.

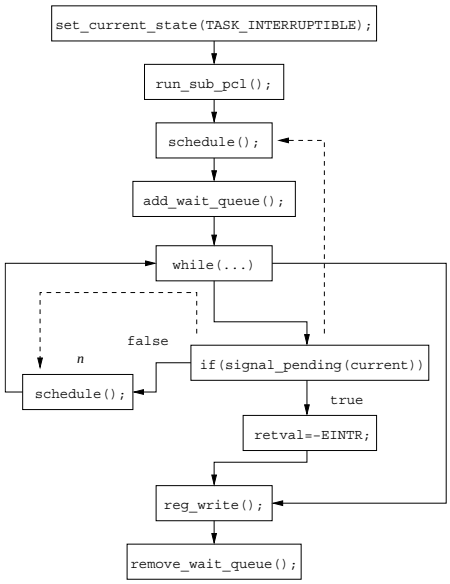



Figure 2: CFG for excerpt of PCILynx driver

## 3 Describing Crosscuts using Temporal Logic

Temporal logic is a logic that is commonly used to express properties of sequences of events. This logic is often used to define properties verifiable using model checking [8], and has been found to be useful for describing paths in control-flow graphs in order to guide compiler optimizations [10]. Following the latter work, we implement an aspect for Bossa integration as a collection of rewrite rules that use temporal logic to describe conditions under which Bossa event notifications should be inserted in kernel code.

### 3.1 Rewrite rules

We propose to define rewrite rules based on control-flow graphs (CFGs), *i.e.*, graphs in which nodes represent statements and decision points of the program and edges connect nodes that can be executed in sequence. Figure 2 shows the CFG for the code excerpt of Figure 1.

We use rewrite rules of the form:

$$LHS \Rightarrow RHS \quad \text{If } condition$$

where *LHS* is a pattern to match against CFG nodes, *RHS* describes the code that should replace the code represented by the given node when this match succeeds, and *condition* describes the conditions under which this transformation should take place.

An example of such a rule is:

$n$ : (`set_current_state(TASK_INTERRUPTIBLE);`) $\Rightarrow$
{$n$; `BOSSA_BLOCK(fn_name,current);`}

The left-hand side of this rule matches a node representing the statement `set_current_state(TASK_INTERRUPTIBLE)` and labels this node $n$. The right-hand side of the rule indicates that the matched statement should be replaced by a sequence consisting of the original statement and generation of a `BOSSA_BLOCK` event. As explained in Section 2.2, this rule applies whenever the state of the running process is set to `TASK_INTERRUPTIBLE`, and thus no condition is needed.

## 3.2 Predicates

To simplify the presentation of the rewrite rules, we define some predicates that describe relevant constructs in the source program.

The predicate stmt($s$) holds of any node representing a statement of the form $s$. The predicate If-t($e$) (or If-f($e$)) holds of any node representing the test portion of a conditional statement, where the test is the expression $e$ and the current control-flow path includes the true (or false) branch of the test. Typical examples are stmt(`schedule()`), which holds at any node representing a call to `schedule()`, and If-f(`signal_pending(current)`), which describes the failure of a test for a pending signal for the current process. The predicate Entry() holds of the node representing the entry point of the current function.

To refer to instructions that set the process state to a specific value, we use the predicate set_state($t$) where $t$ is the name of a Linux process state. For example, set_state(`TASK_INTERRUPTIBLE`) matches `set_current_state(TASK_INTERRUPTIBLE)`. Other predicates match more general sets of state changing operations. The predicate change_to_blocking() holds of a node representing a statement that sets the state of the running process to indicate that the process should block. Examples include `set_current_state(TASK_INTERRUPTIBLE)` and `set_current_state(TASK_UNINTERRUPTIBLE)`. Similarly, change_to_running() holds of a node representing a statement that sets the state of the running process to indicate that the process should remain ready. Examples include `set_current_state(TASK_RUNNING)` and `schedule()`. Finally, we define change_of_state() to be change_to_blocking() $\vee$ change_to_running().

## 3.3 Describing CFG nodes and paths

The conditions in our rewrite rules describe properties of the nodes along a collection of paths in a CFG. For this purpose, we use judgments of the form: $n \vdash \phi$ where $n$ is a node of the CFG and $\phi$ is a formula of temporal logic (specifically, a variant of CTL [10]). Formulas in this logic are as follows:

$$\begin{aligned}\phi \quad ::= \quad & p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ & \mid A(\phi_1 \,\mathsf{U}\, \phi_2) \mid E(\phi_1 \,\mathsf{U}\, \phi_2) \\ & \mid A\triangle(\phi_1 \,\mathsf{U}\, \phi_2) \mid E\triangle(\phi_1 \,\mathsf{U}\, \phi_2) \\ & \mid AX(\phi) \mid EX(\phi) \mid AX\triangle(\phi) \mid EX\triangle(\phi)\end{aligned}$$

The formula $p$ is any proposition. The operators $\neg$, $\wedge$, and $\vee$ are defined as in propositional logic. The remaining formulas describe universally and existentially quantified collections of paths. We illustrate the semantics of these formulas by examples.

A judgment of the form $n \vdash A(\phi_1 \,\mathsf{U}\, \phi_2)$ is satisfied if for each path beginning at $n$, every node along the path satisfies $\phi_1$ *until* a node $n'$ satisfying $\phi_2$ is reached, or the path loops infinitely and every node in the path satisfies $\phi_1$.[2] The node $n'$ need not satisfy $\phi_1$. For Bossa, we are primarily interested in analyzing nodes whose execution precedes a call to `schedule()`, and thus we consider paths that end, rather than begin, at the given node $n$. Such a backwards search is expressed by the operator $\triangle$; thus, we typically use judgments of the form $n \vdash A\triangle(\phi_1 \,\mathsf{U}\, \phi_2)$ rather than $n \vdash A(\phi_1 \,\mathsf{U}\, \phi_2)$.

Our analysis of the instrumentation of Linux for Bossa (Section 2.1) suggests that we would like to define a specific treatment of `schedule()` that should apply when the state of the running process is known to be, *e.g.*, `TASK_RUNNING`. A necessary condition is that every control-flow path ending in the node $n$ representing the given call to `schedule()` changes the state of the running process to `TASK_RUNNING` or reaches the entry point of the function. We express this condition as follows:

$$n \vdash A\triangle(\mathit{True} \,\mathsf{U}\, (\text{change\_to\_running}() \vee \text{Entry}()))$$

The proposition *True* holds at any node. The proposition change_to_running() $\vee$ Entry() holds at any node setting the state of the running process to `TASK_RUNNING` as well as at the entry point of the function. The complete formula thus checks that an assignment of the state to `TASK_RUNNING` occurred at some previous node $n'$, but puts no conditions on the nodes between $n$ and $n'$.

The formulas $E(\phi_1 \,\mathsf{U}\, \phi_2)$ and $E\triangle(\phi_1 \,\mathsf{U}\, \phi_2)$ are analogous to $A(\phi_1 \,\mathsf{U}\, \phi_2)$ and $A\triangle(\phi_1 \,\mathsf{U}\, \phi_2)$ but only

[2] Technically, we use the "weak" form of $A(\phi_1 \,\mathsf{U}\, \phi_2)$.

require the existence of a path whose nodes satisfy the subformulas. Here, however, looping is not allowed; the path must contain a node that satisfies $\phi_2$. As an example, to express that there *exists* a backward path from node $n$, that eventually reaches a node satisfying change_to_running() $\vee$ Entry(), we use the judgment:

$$n \vdash E\triangle(\mathit{True}\ \mathsf{U}\ (\mathsf{change\_to\_running}() \vee \mathsf{Entry}()))$$

In some rules, the analysis should start at all of the nodes preceding the given node $n$. This is expressed by the formula $AX\triangle(\phi)$, which specifies that all direct predecessors of the current node must satisfy $\phi$. The following judgment states that all backwards paths starting from each of the nodes preceding $n$ eventually reach a node satisfying change_to_running() $\vee$ Entry():

$$n \vdash AX\triangle(A\triangle(\mathit{True}\ \mathsf{U}\ (\mathsf{change\_to\_running}() \vee \mathsf{Entry}())))$$

The dashed arrows in Figure 2 represent the paths whose nodes are tested in checking this judgment with respect to the node representing the call to `schedule()` within the `while` loop. $EX\triangle(\phi)$, $AX(\phi)$, and $EX(\phi)$ are defined analogously.

### 3.4 Instrumenting calls to `schedule()`

As a first example of a rule using these temporal operators, we consider the instrumentation of a call to `schedule()` when the state of the running process is known to be `TASK_RUNNING`. In this case, a use of `BOSSA_YIELD_SYSTEM_IMMEDIATE` should be inserted before the call to `schedule()`. The transformation itself is expressed as follows:

$n$ : (`schedule();`) $\Rightarrow$
`{BOSSA_YIELD_SYSTEM_IMMEDIATE(fn_name,current);`$n$`}`

To complete the rule, we must express the conditions under which this transformation applies:

- Starting from the predecessors of $n$ every backwards path should lead either to a node that sets the process state to `TASK_RUNNING`, (*i.e.*, where change_to_running() is true) or to the first instruction of the current function.
- On these paths there should not be any intermediate change of the process state (*i.e.*, change_of_state() should be false at each node before the end of such a path).

We express these conditions as follows:

$$n \vdash AX\triangle(A\triangle(\neg\mathsf{change\_of\_state}()\ \mathsf{U}\ (\mathsf{change\_to\_running}() \vee \mathsf{Entry}())))$$

We next consider instrumentation of a call to `schedule()` when the state of the running process is known to be `TASK_INTERRUPTIBLE`. If it is possible that a signal is pending for the running process, `BOSSA_CHECK_PENDING_SIGNAL` should be inserted before the call to `schedule()`. The transformation itself is expressed as follows:

$n$ : (`schedule();`) $\Rightarrow$
`{BOSSA_CHECK_PENDING_SIGNAL(fn_name,current);`$n$`}`

To verify that the state of the running process is `TASK_INTERRUPTIBLE`, we use the following condition:

$$n \vdash AX\triangle(A\triangle(\neg\mathsf{change\_of\_state}()\ \mathsf{U}\ \mathsf{set\_state}(\texttt{TASK\_INTERRUPTIBLE})))$$

To verify that a pending signal is possible, we also check that there is some control-flow path to the call to `schedule()` on which a test for a pending signal has not already been performed:

$$n \vdash EX\triangle(E\triangle(\neg\mathsf{If\text{-}f}(\texttt{signal\_pending(current)})\ \mathsf{U}\ (\mathsf{stmt}(\texttt{schedule();}) \vee \mathsf{Entry}())))$$

It is straightforward to see that this rule applies to the call to `schedule()` before the `while` loop in Figure 2. More interestingly, we observe that the rule does not apply to the call to `schedule()` in the body of the `while` loop. Specifically, the first condition fails. Every backwards path from this call to `schedule()` either loops or eventually leads to the setting of the state of the running process to `TASK_INTERRUPTIBLE`, but each non-looping path contains the first call to `schedule()`, which performs a change of state. This example thus illustrates the usefulness of temporal logic in this setting.

## 4 Related Work

AspectC is an aspect system targeted towards C code and has been used to implement various OS concerns [2, 3]. In this work, the `cflow` construct of AspectJ has been found useful to describe the set of functions that should appear on the call stack if an aspect is to apply. Walker and Murphy have further proposed to consider ordered sequences rather than simply sets of pending calls [14]. While the order of operations is essential to our rules, our rules depend on sequencing of individual instructions rather than nested function calls. Furthermore, `cflow` describes dynamic control flow, whereas a specific Bossa event notification can in almost all cases be chosen statically, leading to more efficient code than a dynamic solution.

There have been several uses of logic in specifying non-local properties in program transformation rules. Lacey and de Moor proposed to use temporal logic to describe conditions on rewrite rules [10]. We follow their approach here. Subsequent work by Lacey *et al.* showed how to prove the correctness of standard compiler optimizations based on this approach [11]. Drape *et al.* have developed a variant of logic programming that permits to conveniently express rules of the form we have used here [5]. Nevertheless, their target is .NET rather than kernel C code.

Metal is a language for writing static checkers, that are then executed using the xgcc static analysis engine [7]. Metal checkers have been used to find many bugs in Linux and OpenBSD [6]. Although the goal of Metal and xgcc is to check properties, xgcc provides some functions for modifying the abstract syntax tree that could possibly be used for program transformation. The main difference between Metal and our approach is in the underlying logic that is used. Metal is essentially a language for describing state machines. While arbitrary C code can be invoked, thus extending the expressiveness of the language, this code must be manually verified to satisfy the independence and determinacy conditions imposed by xgcc. Indeed, our rules rely on existential and universal quantification over paths, and cannot be expressed in Metal without resorting to the use of C code. By expressing our rules completely within a single logic, we are assured that the rules are well-defined. Furthermore, we can profit from a large body of research on understanding and implementing temporal logic, and our rules serve as an unambiguous form of documentation.

# 5 Conclusion

In this paper, we have presented an AOP-based transformation system for re-engineering an existing OS kernel to support the Bossa framework. The aspect defines rules that use the temporal logic CTL to define conditions for instrumenting the original kernel.

At the current state of our work, we have defined a set of 15 rules that are sufficient to carry out the instrumentation of the Linux 2.4 kernel that we previously performed by hand. Preliminary inspection of the remaining Linux code does not reveal any issues that these rules do not address. We are currently implementing our approach using the CIL infrastructure, developed by Necula *et al.* [13]. This implementation will both transform code that satisfies the rules and warn the user about unanticipated patterns of scheduling points.

In the longer term, we plan to port Bossa to other OSes, such as BSD and Windows, and to apply the Bossa approach to other system services. We anticipate that aspects should be useful to integrate the framework with the OS in these settings as well.

The Bossa prototype is available at `http://www.emn.fr/x-info/bossa`.

# References


[1] L. Barreto, G. Muller, J. Lawall, and K. Kono. A framework for simplifying the development of kernel schedulers: design and performance evaluation. Technical Report 02/8/INFO, École des Mines de Nantes, Apr. 2002.

[2] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, pages 88–98, Vienna, Austria, Sept. 2001.

[3] Y. Coady, G. Kiczales, J. S. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 79–86, St. Emilion, France, Sept. 2002.

[4] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd Intl. Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, Sept. 2001.

[5] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic. In *Intl. Conf. on Principles and Practice of Declarative Programming*, pages 133–144, Pittsburgh, PA, Oct. 2002.

[6] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[7] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, pages 69–82, Berlin, Germany, 2002.

[8] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kerstin, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, Budapest, Hungary, June 2001.

[10] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Intl. Conf. on Compiler Construction (CC)*, volume 2027 of *Lecture Notes in Computer Science*, pages 52–68, Genova, Italy, 2001.

[11] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 283–294, Portland, OR, Jan. 2002.

[12] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–62, Saint-Emilion, France, Sept. 2002.

[13] S. McPeak, G. Necula, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Intl. Conf. on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Grenoble, France, Mar. 2002.

[14] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Proceedings for Advanced Separation of Concerns Workshop*, pages 134–139, Toronto, Canada, May 2001.

# Speed vs. Memory Usage
# -
# An Approach to Deal with Contrary Aspects

Wolfgang Schult and Andreas Polze
Hasso-Plattner-Institute
14440 Potsdam, Germany
{wolfgang.schult|andreas.polze}@hpi.uni-potsdam.de

## ABSTRACT

Besides design and implementation of components, software engineering for component-based systems has to deal with component integration issues whose impact is not restricted to separate components but rather affects the system as a whole. The bigger the software system is, the more difficult it will be to deal with. Aspect-Oriented programming (AOP) addresses these cross-cutting, multi-component concerns. AOP describes system properties and component interactions in terms of so-called aspects. Often, aspects express non-functional component properties, such as resource usage (CPU, memory, network bandwidth), component and object (co-) locations, fault-tolerance, timing behavior, or security settings. Typically, these properties do not manifest in the components' functional interfaces.

Aspects often constrain the design space for a given software system. System designers have to trade off multiple, possibly contradicting aspects affecting a set of components (e.g.; the fault-tolerance aspect may require replication of component data, whereas the security aspect may prohibit it). Component software may be deployed in varying contexts, maybe requiring emphasis on only a few of the aspects considered during design and implementation. Static aspect weavers often require compromises with respect to the generality of services provided by a component system.

In this paper, we focus on dynamic management of aspect information during program runtime. We introduce an approach called "dynamic aspect weaving" to interconnect aspect code and functional code. Using our approach, it is possible to decide at runtime whether objects living inside a component should be instantiated with support for a particular aspect or not. We present a distributed Mandelbrot computation as an example and discuss dynamic aspect weaving as a technique to manage speed versus memory usage trade-offs. We have implemented our approach in the context of the C# language and the Microsoft .NET and the ROTOR environment.

## 1. INTRODUCTION

There exists a variety of application areas for Aspect-Oriented Programming (AOP). Generally, it is very acceptable to have a preprocessor-like aspect-weaver to interconnect functional code and aspect code. However, sometimes it is desirable to postpone the decision about whether aspect information is to be interwoven with a particular component until program runtime. For instance, one may have a huge resource consuming image processing algorithm located in a component, and depending on system load and available computing nodes a trade-off between data distribution, memory allocation scheme, and utilization of computing power has to be made at runtime. It might be desirable to distribute calculations for better performance if computing nodes are available. Minimizing local memory usage might be at high priority if the same program is run in a different setting. Both are crosscutting concerns. An aspect may be defined to manage distribution of method invocations across machine boundaries, whereas a different (somewhat contrasting) aspect may deal with local and remote memory utilization during a distributed computation.

Typically, one has to decide at compile time whether an aspect should be interwoven with a set of components or not. Classical AOP techniques provide neither a solution to 'switch off' (ignore) aspect code at runtime nor to dynamically interweave another aspect with the component software.

In this paper, we present a solution to this problem and demonstrate how to interweave previously defined aspects with functional component code. This 'Dynamic Aspect Weaving' is promising because of its flexibility: neither at design nor at compilation time does a definite decision has to be made about whether a particular aspect should be applied to a set of components or not. Aspects specialized for a particular situation can be defined and can be interwoven depending on actual runtime requirements. Furthermore one can parameterize the aspects during program runtime. We discuss how this can be accomplished without usage of a special 'aspect weaver' tool.

The remainder of the paper is organized as follows: Section 2 presents related work. Section 3 describes our approach to dynamic aspect weaving. In Section 4 we demonstrate a case study whose experimental evaluation is presented in Section 5. In Section 6 we summarize our conclusions.

## 2. RELATED WORK

The concept of aspect-oriented programming (AOP) offers an interesting alternative for specification of non-functional component properties (such as fault-tolerance properties or timing behavior). There are a variety of language extensions to deal with AOP. One of which, AspectJ [13], a Java extension, can be cited as the most prominent example. The central concept of most AOP-frameworks is a join point model described in [12][5].

JAC is a Java framework that provides support for dynamic aspect-oriented applications [20]. With JAC it is also possible that an aspect can be woven and unwoven at runtime. An aspect oriented program in JAC is entirely written in regular Java and consists of several different parts, such as base program, and other different aspect programs. The weaver deploys the aspect objects so that the aspect program crosscuts the base program.

Mehmet Aksit has developed the composition filters object model, which provides control over messages received and sent by an object [3][1]. In this work, the component language follows traditional object-oriented programming techniques, the composition filters mechanism represents an aspect language that can be used to control a number of aspects including synchronization and communication. Most of the weaving happens during runtime.

The authors have implemented a static aspect weaver, which uses the unmanaged metadata interfaces from .NET to interweave aspect code [21].

A restricted technique for dynamic aspect weaving for .NET has been described in [15]. However, this solution uses the current internal debug interfaces of the .NET framework implementation to interweave aspect code during runtime and is therefore less general and portable than our approach.

## 3. DYNAMIC ASPECT WEAVING

Dynamic aspect weaving means that a component (a *target class*) and an *aspect class* will become interwoven during runtime. There is no need for the aspect class to have a priori knowledge about the target class and vice versa. To understand how the weaving process works, some notions have to be defined.

### 3.1 What is an Aspect Class?

An aspect description for a set of components focuses on crosscutting concerns. In our case, an aspect is a simple C# class derived from the base class **Aspect**. It will be called *aspect class*. Aspect classes may implement methods, properties, and member variables. In any case, an aspect class describes a way to modify the behavior of another class (the so-called *target class*). Therefore, there is no point to instantiate an aspect class on its own. Rather it has to be instantiated jointly with a target class. This process is called *dynamic aspect weaving*. Its technical details will be described later in this section.

### 3.2 Connection Points

As mentioned above, an aspect class works only in conjunction with an instance of another class. At a *connection point* both will become interwoven. Methods of the aspect class can be identified as connection points, which is indicated by the C# **call** attribute above the method definition in the aspect class. The call attribute is defined as follows:

```
[call(Invoke.InvokeOrder{, Alias=AliasName})]
```

During dynamic aspect weaving, all of the connection points inside an aspect class will become interwoven with a target class's method if at least one of the following requirements is met:

1. The method name and the signature are equivalent.
2. If there is an *AliasName* defined, and the method name from the target class is the same as the alias, and the signatures of both are equivalent.
3. If there is an *AliasName* and the alias contains a wildcard at the end, or the signature of the Aspect class method contains wildcards, and the target method matches.

The following example demonstrates requirement 1:

```
[call(Invoke.Instead)]
void mymethod(int i) { /* ... */ }
```

In this case any target method **mymethod** with one **int** as parameter and **void** as result will interweave with this method in the aspect class.

To demonstrate requirement 2 let us assume that one defines **Alias="myspecialmethod"** for a method. This results in interweaving all target methods named **myspecialmethod** with an **int** parameter and a **void** with the annotated method in the aspect class.

Requirement 3 basically says that if one modifies the alias to **Alias="my*"** every target method beginning with "my" and the same parameters will become interwoven. Furthermore one can use *signature wildcards*. A wildcard for the result type is **object**, and for the parameters **params object[]**, this is like a method with variable arguments. An alias has to be defined in order to flag the argument list **params object[]** as wildcard. The following connection point:

```
[call(Invoke.Instead, Alias="*")]
object catchall(params object[] args)
```

will become interwoven with every method in the target class and *args* will contain each parameter one passes through the method. For instance, if the target class has a method **void f(int i, double d)**, then *args[0]* will contain $i$ and *args[1]* will contain $d$ after the method is called.

Now, since we have described the rules for interweaving connection points with target methods, we will focus on the actual algorithm implementing dynamic aspect weaving. This is described by the *InvokeOrder* parameter of the call attribute. There are three possibilities:

- **Invoke.Before**: The aspect method of the connection point will be invoked *before* the target method will be called.
- **Invoke.After**: As to be expected, the aspect method will be invoked *after* the target method has been called.
- **Invoke.Instead**: The target method will not be called automatically - but can be called from inside the aspect method.

The first two cases are useful if one wants to trace method calls only. The last case is to be used in order to gain full control over the target method's behavior.

## 3.3 Aspect Context

When defining an *Invoke.Instead* connection point, one needs a mechanism to call the appropriate target class method. The problem is that neither the type of the target class (the aspect class can become interwoven with any type) nor, in some cases, the signature of the called method (this is when one uses signature wildcards) are known. The solution is to define a **Context** property in the *Aspect* base class. This property allows access to an object of type **AspectContext** which contains the required information. There are two methods defined for **AspectContexts**:

```
public object Invoke(params object[] args)

public object InvokeOn(object target, params object[]
    args)
```

The first simply invokes the target class's method on an object with the given parameters. The second method allows invocation of the target method on a different, arbitrarily chosen instance (*target*) of the target class. This is useful if there are special instances of the target class stored in the aspect code, and one wants to invoke these.

## 3.4 Implementation Issues

In the previous section, we have introduced our notions of an aspect class, of connection points, and of object contexts. Here, we are going to discuss our implementation of this concept. Our approach relies on a number of language features, namely:

- Support of attribute definition.
- Support of reflection to analyze the target class's and the aspect class's signatures (methods and their parameter types).
- Runtime code generation, to emit the interwoven class.

We have implemented our solution based on Microsoft .NET. The Microsoft .NET runtime environment allows to generate, load, and run code on the fly. This code can be presented to the environment in an intermediate language (IL). There exist a variety of programming languages which support .NET and map on the same intermediate language. Since our approach it is possible to interweave an aspect written in one language (say C++) with a component written in a different .NET language (say Pascal).

We have implemented our technique for dynamic aspect weaving in a .NET library. This library provides several classes and attributes defined within the namespace **Aspects**:

- **Aspect** is the base class for all defined aspects.
- **Weaver** is a class which implements the weaving functionality.
- **Call** is an attribute to define connection points.
- **AspectContext** allows invocation of instance methods via *Aspect.Instance*.

## 3.5 The Dynamic Aspect Weaver

As described above, the **Aspects** namespace contains a class called **Weaver**. It provides a function named **CreateInstance** to interweave a given target class. This function

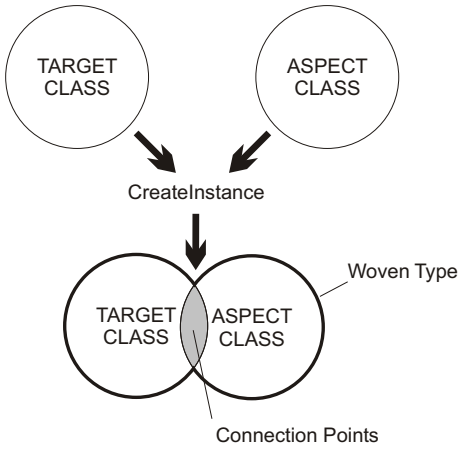


Figure 1: The Weaving Process

does the same as the **new** statement - it creates a new object of a given class (the target class). But furthermore this function interweaves the target class with an aspect-object. This can be done in two ways; dynamic or static. The dynamic version is as follows:

```
A a=Weaver.CreateInstance(typeof(A), null, new MyAspect
    ()) as A;
```

In this example, an instance of the class *A* will be generated and dynamically interwoven with an aspect object of *MyAspect*. In the static case one can simply use .NET-attributes to express that a class should be interwoven with a certain aspect:

```
[MyAspect]
class A
{ /* ... */ }
/* ... */
A a=Weaver.CreateInstance(typeof(A), ...) as A;
```

Giving the aspect instance explicitly as a parameter to *CreateInstance* is more flexible than naming it via attribute - as the aspect and its parameters can be identified at runtime. The code implementing dynamic aspect weaving first looks for a custom attribute derived from **Aspect**. If there is no aspect given, the *CreateInstance* call is equivalent to **new A(*args*)**. What happens during the creation is illustrated in Figure 1. The weaver looks for connection points and tries to join them with the target class's methods as described above. With this information, it builds a new type, and creates a new instance of this type. After that the weaver calls a special method named *ctor* in the aspect, to inform them that it was interwoven with a newly created object. This method can be overridden and has the following signature:

```
virtual void ctor(Weaver weaver, object target, params
    object[] args)
```

Inside the method, the parameters have the following meaning:

- *weaver* is the aspect weaver itself.
- *target* is the new interwoven instance.
- *args* are the constructor parameters.

Finally, the newly constructed and interwoven object instance will be returned to the caller.

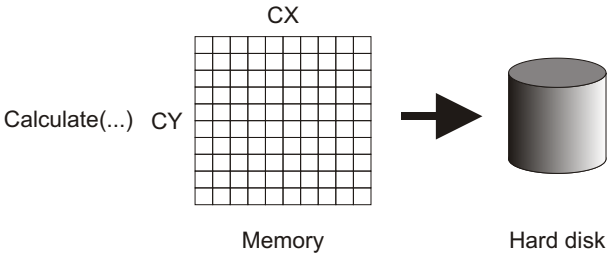



Figure 2: Mandelbrot Function Call

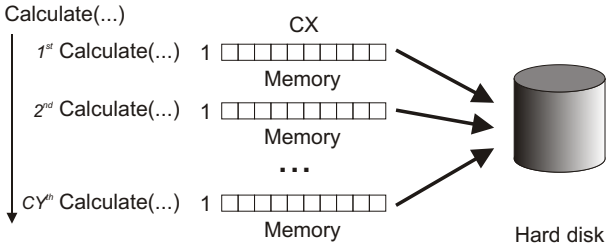



Figure 3: Function Call with the SaveMemory Aspect

# 4. CASE STUDY - OPTIMIZING RESOURCE USAGE

Listing A in the appendix shows a C# class which calculates a Mandelbrot set [18]. The input for the algorithm is a filename, a bounding box, and a resolution.

Figure 2 illustrates the behavior of our Mandelbrot computation: The algorithm first calculates the whole Mandelbrot set in memory and then stores it to the hard disk. For small resolutions this is fine. But what happens if the resolution is increased? The amount of memory consumed will increase polynomial (one needs $cx*cy$ memory storage). A possible solution is to rewrite the algorithm. But under certain circumstances, there is no possibility to do that (i.e. the algorithm exists as binary only), so ... another solution is needed.

## 4.1 The Save Memory Aspect

The idea is that the function calls are split so that single lines will be processed in memory and subsequently written to separate files on the hard disk. Finally, all these files are joined together to complete the Mandelbrot computation. Figure B shows this approach. The envisioned effect can be accomplished using an aspect class (which would not be visible to clients of our Mandelbrot computation). Listing B shows a possible implementation of this aspect.

As visible in the aspect class, the function *calculate* is defined as a connection point. As described in Section 3, if the target class contains a function *Calculate* with the same signature, then both will become interwoven. The **for**-loop simply invokes, via the aspect context, the Mandelbrot computation line by line. For $n$ lines, it will generate $n$ files on the hard disk. Finally, these $n$ files will become concatenated to form a new file containing the data originally requested.

## 4.2 The Distribution Aspect

The second goal was to utilize all available processors in a system. Again, we are defining an aspect to tackle this problem. Figure 4 demonstrates what has to be done: First, one instantiates a replica of the original Mandelbrot object on each processor available. Second, on every function call,

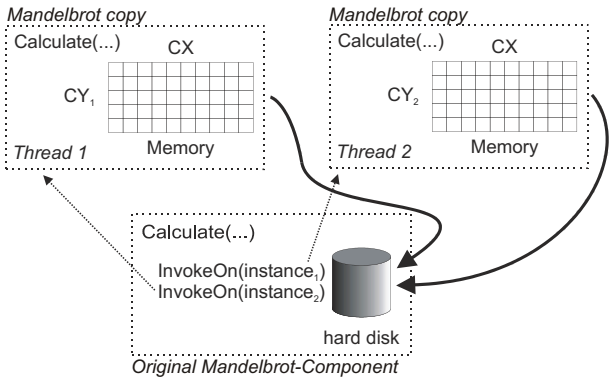



Figure 4: Function Call with the Distribution Aspect

one splits the calculation up and delegates each part to a separate thread. One can use the .NET threadpool for this. The original Mandelbrot object gets the results back from each replica and joins them together. Listing C shows an excerpt of the actual C# implementation.

The aspect class contains three important functions. The first is **ctor**, which will be called by the Weaver when the instance is created. It is used to create additional instances of the same type which may process function calls in parallel. The second is **Calculate**. This method contains the **call** attribute, which defines it as connection point as well. Here the function calls are executed in separate threads operating on disjunct copies of the Mandelbrot object.

## 4.3 The Client Side

On the client side, only the instantiation of the Mandelbrot class changes. Depending on the actual runtime environment, one or the other aspect will become interwoven with the Mandelbrot class (Listing 1).

```
Mandelbrot mb;
// we need less memory usage
if(opt_memory.Checked)
  mb=Aspects.Weaver.CreateInstance(typeof(Mandelbrot),null,new
       SaveMemory()) as Mandelbrot;
// we need more performance
else if(opt_speed.Checked)
  mb=Aspects.Weaver.CreateInstance(typeof(Mandelbrot),null,new
       Distribute("d:/temp")) as Mandelbrot;
// we need nothing of both
else mb=new Mandelbrot();
```

Listing 1: The Client Side

The function call initiating the actual Mandelbrot computation does not change.

# 5. PERFORMANCE MEASUREMENTS

After implementing a dynamic weaver and designing two aspects, we evaluate the performance impact of our dynamic aspect weaver. For our experiments, we have used a 1GHz Dual-Pentium III System with 256MB RAM. We show here the impact of both the Distribution and the Save Memory Aspect on our system resources. Figure 5 shows the average duration of the mandelbrot calculation in dependence on the number of calculated columns in the mandelbrot matrix ($CX$). We have sketched out two representive row counts ($CY$) for the three cases. For a row count of 4096 one can see that the algorithm with the Distribution Aspect assigned is approximately twice as fast as it is without an aspect. With

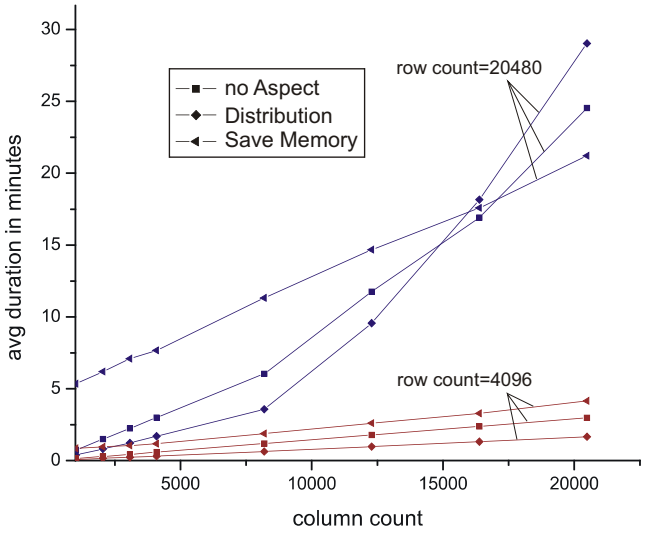



Figure 5: Comparison of average duration (20 measurements per point) between both aspects and without any aspect in the mandelbrot component

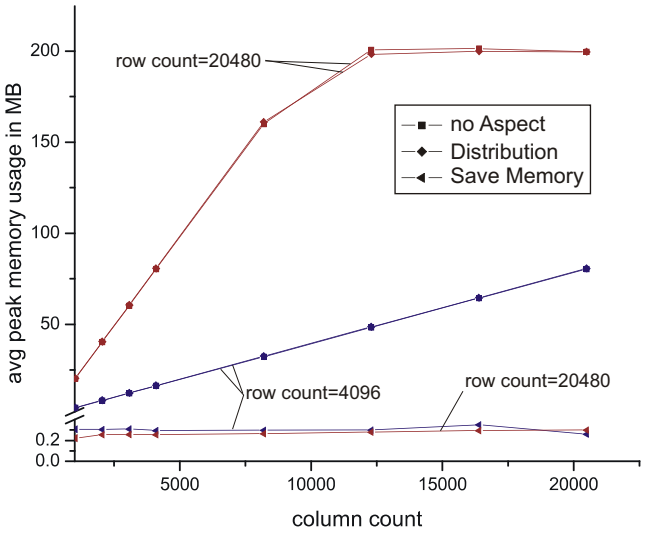



Figure 6: Comparison of average peak memory usage (20 measurements per point) between both aspects and without any aspect in the mandelbrot component

a Save Memory aspect we have a performance gap of approximately fifty percent compared to the calculation without an aspect. But with a row count of 20480 the situation changes markedly.

Beginning at a column count of approximately 17200 the algorithm assigned with the Save Memory aspect gets the best performance. The explanation for that is shown in Figure 6. One sees that in the measurements with a column count 12288 the maximum of available physical memory has been exhausted. On the other hand, the algorithm assigned with the Save Memory aspect uses a consistently low amount of memory. This prevents it from swapping out memory and so decreasing its performance.

## 6. CONCLUSIONS

Aspect-oriented programming (AOP) is a relatively new approach for separation of concerns in software development. AOP makes it possible to modularize crosscutting aspects of a system.

We have presented our approach to dynamic management of aspect information at program runtime. We have introduced a technique called "dynamic aspect weaving" which allows for late binding (weaving) of aspect code and functional code. Using our approach, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment. Relying on the .NET support for a variety of programming languages, our approach is not restricted to C#, but works for all of the .NET languages and other .NETenvironments like ROTOR, among others.

Our current implementation has some constraints for the programmer of a component. Currently, only virtual methods can be interwoven dynamically. The reason for this lies in our implementation of late binding of the function calls. Currently the Weaver "overrides" the function so that the virtual method table maintained inside the .NET virtual machine points to the woven function (the version enriched with aspect information). Other members of a class, such as fields, properties, static, and class functions currently cannot be accessed this way. However, recursively applying the AOP techniques described here and in [21], it is a simple task to generate proxy classes which substitute non-virtual member functions and fields with their virtual counterparts.

## 7. REFERENCES

[1] M. Aksit and L. Bergmans. Composing multible concerns using composition filters. *Communications of the ACM*, 44, Issue 10:51–57, Oktober 2001.

[2] M. Aksit and B. Tekinerdogan. Aspect-oriented programming using composition-filters. In *ECOOP'98 Workshop Reader*. Springer Verlag, 1998.

[3] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. AOP'98 workshop position paper, 1998.

[4] T. Archer. *Inside C#*. Microsoft Press, 1 edition, 2001.

[5] AspectJ Homepage. http://www.aspectj.org/, 2002.

[6] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *1st International Conference on Aspect-Oriented Software Development*

*(AOSD)*, pages 86–95, Enschede, The Netherlands, April 22-26 2002. ACM press.

[7] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aop. In *Communications of the ACM*, volume 44, pages 33–38, Oktober 2001.

[8] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. In *Communications of the ACM*, volume 44, pages 30–32, Oktober 2001.

[9] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *Second Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 21-22 2002.

[10] S. Hanenberg and R. Unland. Concerning aop and inheritance. In *Aspektorientierung - Workshop der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung*, Paderborn, Germany, May 3-4 2001.

[11] S. Hanenberg and R. Unland. A proposal for classifying tangeled code. In *Second Workshop on Aspect-Oriented Software Development*, Bonn, Germany, February 21-22 2002.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44, Issue 10:59–65, October 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, Finnland, June 1997. Springer Verlag LNCS 1241.

[14] J. O. K.Lieberherr, D. Orleans. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44, Issue 10:39–41, Oktober 2001.

[15] J. Lam. My runtime aspect weaver. http://www.iunknown.com, 2002.

[16] C. V. Lopes and G. Kiczales. *Recent Developments in AspectJ.* Xerox Palo Alto Research Center.

[17] D. Mahrenholz, O. Spinczyk, and W. Schrder-Preikschat. Program instrumentation for debugging and monitoring with aspect c++. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 249–256, Crystal City, VA, USA, April 29 - May 1 2002.

[18] B. Mandelbrot. *The Fractal Geometry of Nature.* Freeman, San Francisco, 1982.

[19] Microsoft Cooperation, http://msdn.microsoft.com/net/ecma/. *ECMA C# and Common Language Infrastructure Standards*, 2001.

[20] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Reflection 2001*, September 2001.

[21] W. Schult and A. Polze. Aspect-oriented programming with C# and .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 241–248, Crystal City, VA, USA, April 29 - May 1 2002.

[22] W. Schult and A. Polze. Dynamic aspect-weaving with .NET. In *Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen*, TU Berlin, Germany, November 7-8 2002.

# APPENDIX

## A. THE MANDELBROT CLASS

```
public class Mandelbrot
{
  // calculates a given point of the Mandelbrot matrix
  private byte CalculatePoint(double x, double y) { /* ...*/ }

  // only this method is accessible from outside
  // it calculates the matrix and
  // stores the result to the hard disk
  public virtual void Calculate(string filename, double x1, double y1, double x2, double y2, int xRes, int yRes)
  {
    double dAddx=(x2-x1)/((double)xRes);
    double dAddy=(y2-y1)/((double)yRes);
    Byte[] matrix=new Byte[yRes*xRes];
    for(int y=0;y<yRes;y++)
    {
      x2=x1;
      for(int x=0;x<xRes;x++)
      {
        matrix[xRes*y+x]=CalculatePoint(x1,y1);
        x1+=dAddx;
      }
      y1+=dAddy;
      x1=x2;
    }
    FileStream fs=new FileStream(filename, FileMode.Create, FileAccess.Write);
    fs.Write(matrix,0,matrix.Length);
    fs.Close();
  }
}
```

## B. THE SAVE MEMORY ASPECT

```
public class SaveMemory:Aspect
{
  [Call(Invoke.Instead)] // connection point
  public void Calculate(string filename, double x1, double y1, double x2, double y2, int xRes, int yRes)
  {
    // split up in lines
    double dStep=(y2-y1)/((double)yRes);
    for(int i=0;i<yRes;i++)
    {
      // call original function
      Context.Invoke(filename+i.ToString(),x1,y1,x2,y1,xRes,1);
      y1+=dStep;
    }
    // join the files together
    Byte[] data=new Byte[xRes];
    FileStream fsdst=new FileStream(filename, FileMode.Create, FileAccess.Write);
    for(int i=0;i<yRes;i++)
    {
      FileStream fssrc=new FileStream(filename+i.ToString(), FileMode.Open, FileAccess.Read);
      fssrc.Read(data,0,data.Length);
      fssrc.Close();
      fsdst.Write(data,0,data.Length);
    }
    fsdst.Close();
  }
}
```

## C. THE DISTRIBUTION ASPECT (EXCERPT)

```
public class Distribute:Aspect
{
  private object[] instances;
  private int workcount;

/* ... */

  // here we generate the copies from the mandelbrot component
  public override void ctor(Weaver weaver, object o, object[] args)
  {
    // get processor count from current system
    System.Int32 affinity=System.Diagnostics.Process.GetCurrentProcess().ProcessorAffinity.ToInt32();
    int iInstances=0;
    while(affinity!=0)
    {
      if((affinity & 1)!=0) iInstances++;
      affinity=affinity>>1;
    }
    // and generate copies
    instances=new Object[iInstances];
    while(iInstances--!=0)
    {
      instances[iInstances]=weaver.CreateInstance(o,args);
    }
  }
  // the connection point
  [Call(Invoke.Instead)]
  public void Calculate(string filename, double x1, double y1, double x2, double y2, int xRes, int yRes)
  {
    // split up calculation in threads
    workcount=instances.Length;
    int nyRes=yRes/workcount;
    double yStep=(y2-y1)/((double)yRes);
    double yRange=yStep*nyRes;
    AutoResetEvent ev=new AutoResetEvent(false);
    double ny1=y1;
    int iNum;
    for(iNum=0;iNum<instances.Length-1;iNum++)
    {
      double ny2=ny1+yRange;
      System.Threading.ThreadPool.QueueUserWorkItem(
        new WaitCallback(Distribute.InvokeWorker),
        new WorkItem(       // this is a container for
          this,             // aspect
          ev,               // event
          instances[iNum],   // mandelbrot instance
          GetFilename(iNum), // temporary filename
          x1, ny1, x2, ny2,  // boundaries
          xRes, nyRes));     // resolution
      ny1=ny2+yStep;
```

```
    }
    System.Threading.ThreadPool.QueueUserWorkItem(
      new WaitCallback(Distribute.InvokeWorker),
      new WorkItem(this, ev, instances[iNum],GetFilename(iNum), x1, ny1, x2, y2, xRes, yRes-(nyRes*(instances.Length-1))) );
    // wait until ready
    while(workcount!=0) ev.WaitOne();
    // join files
    FileStream fsdst=new FileStream(filename, FileMode.Create, FileAccess.Write);
    for(iNum=0;iNum<instances.Length;iNum++)
      Copy(GetFilename(iNum),fsdst); // copy file to filestream
    fsdst.Close();
  }
  // callback for threadpool
  public static void InvokeWorker(object para)
  {
    // unpack parameters from workitem and start calculation
    WorkItem item=(WorkItem)para;
    item.aspect.Context.InvokeOn(item.target, item.filename, item.x1, item.y1, item.x2, item.y2, item.xRes, item.yRes);
    // signal ready
    Interlocked.Decrement(ref item.aspect.workcount);
    item.readyevent.Set();
  }
}
```

# Managing Complexity In Middleware

Adrian Colyer
IBM UK Limited
Hursley Park, Winchester
England. SO21 2JN
+44 (0)1962 816329
adrian_colyer@uk.ibm.com

Gordon Blair
Computing Department
Lancaster University, Bailrigg
Lancaster, England.LA1 4YR
+44 (0)1524 593809
gordon@comp.lancs.ac.uk

Awais Rashid
Computing Department
Lancaster University, Bailrigg
Lancaster, England. LA1 4YR
+44 (0)1524 592344
marash@comp.lancs.ac.uk

## ABSTRACT
Middleware is becoming increasingly complex, and this complexity is at odds with one of middleware's key goals – to make it easier to build distributed systems. A new emphasis on simplicity, componentization and application-middleware independence is required to redress the situation. Aspect-oriented software development techniques hold great promise in helping to meet these challenges, though the large scale of many middleware development projects raises additional requirements that must be met.

## 1. INTRODUCTION
*"There is literally no sensible, economic way to develop distributed applications without middleware services."*
– Richard Schreiber, 1995 [1]

Enterprise applications depend on distributed systems, and therefore on middleware. Within the enterprise, distributed systems are used to provide high levels of availability and scalability, to physically separate components for security reasons, to cope with the geographic spread of multi-national corporations, and to exploit the price-performance characteristics of PC and Unix based workstation clusters. Distributed systems also arise naturally through mergers and acquisitions, and business-to-business applications that span organizational boundaries.

Building distributed systems directly on top of networked operating systems is expensive, error-prone and difficult [2], therefore corporate developers rely on middleware, whose primary purpose is to make it easier to build, deploy and operate distributed applications. Middleware makes building distributed systems easier by resolving heterogeneity, providing transparency of various kinds, and by providing qualities of service.

In section 2, we argue that middleware itself is becoming increasingly complex as we strive to build ever more sophisticated distributed systems. If left unchecked, this trend will leave us facing the same set of problems that middleware was intended to solve in the first place – building distributed systems for enterprise applications will be too complex.

Section 3 sets a direction for the development of future middleware platforms based on simplicity, independence of applications from middleware, and componentization. In section 4 we discuss the application of aspect-oriented software development (AOSD) techniques to meet these goals, and in section 5 we discuss the implications of the large scale of many middleware development projects on AOSD. Section 6 concludes and provides a brief summary of related work.

## 2. MIDDLEWARE COMPLEXITY
*"There is already too much diversity of middleware for many customers and application developers to cope with ... the complexity of current middleware is untenable over the long term"* – Philip Bernstein, 1996 [3].

Middleware resolves heterogeneity and provides transparency in order to make it simpler to build distributed systems. Yet middleware itself is becoming increasingly complex – there are many heterogeneous middleware environments that need to be integrated, and the use of middleware is not transparent to the application developer.

This complexity is driven by rich feature sets and feature interactions both within and across middleware products, the need to support ever more diverse environments, and the introduction of (needed) more sophisticated capabilities that threaten to reduce transparency further.

### 2.1 Feature Complexity
Many modern middleware products are rich in features, each feature independently justifiable for sound business reasons. Current state-of-the-practice is to roll these features into a large, monolithic product. This can result in significant complexity and confusion for the application developer working on the middleware platform. Often there are several ways of achieving the same goal with no obvious rationale for choosing between them, and the sheer number of features to learn and investigate can be overwhelming. Consider as an example the Java™2 Platform Enterprise Edition (J2EE™) [4]: J2EE provides a wealth of APIs and services, the cornerstone of which are Enterprise JavaBeans™(EJB™), JavaServer Pages™(JSP™), and Servlets. Mastering the art of building EJB based systems alone can be quite a challenge [5].

### 2.2 Portfolio Complexity
As well as complexity within a single middleware product, there is additional significant complexity when multiple middleware products are used. The inevitable overlap in capabilities caused by their large feature sets creates additional duplicate means of achieving an end, and hence further developer confusion.

Nearly all large enterprises contain a broad mix of middleware products and home-grown capabilities acquired through the processes of time, business and mission growth, and even mergers and acquisitions. The resulting computing facilities are hard to integrate, and even harder to operate and administer.

## 2.3 New Requirements

The need to support increasingly heterogeneous environments also drives middleware complexity. Pervasive (or ubiquitous) computing brings challenges due to widely varying device formats and capabilities, unreliable network connections, disconnected operation considerations, and mobility of devices (nomadic devices and ad-hoc networking) [6]. Pushing into new markets, such as taking an enterprise product into the small-medium business (SMB) segment, places new emphasis on existing requirements, and creates entirely new requirements too. The integration of business applications across business boundaries (B2B applications) unites very diverse operating environments across independent domains of control. New transaction models and interaction protocols are needed to deal with this additional complexity.

Meanwhile, middleware research continues to push the boundaries of current capabilities, for example in support of very large scale systems [7, 8] and reflective middleware [9]. A common trend in this research is to give the programmer more influence over the behavior of the middleware. The consequence is that more of the middleware becomes visible to the programmer, and some aspects of distribution and heterogeneity become less transparent [2].

# 3. MANAGING COMPLEXITY

*"For the next several years, corporate buyers will…look for technologies that address business problems directly; provide near-term return on investment, and improve customer acquisition and retention, cost-cutting, revenue or profits."* – FORTUNE, March 18th 2002 [10].

The rise in middleware complexity comes at a time when technology discussions are moving from the IT department to the boardroom. Here the debate centers on solutions to business problems, not technology platforms. Business solutions need to be built and deployed as speedily as possible in order to remain competitive; this calls for a responsive middleware platform in which mastery and deployment of only those components absolutely necessary for the task in hand is required.

To take middleware forwards, we need to focus on simplicity instead of complexity, on componentization and configuration instead of monolithic construction, and on loosening the ties between an application and the middleware platform it executes on. All three of these goals are considered from the perspective of the user of the middleware.

## 3.1 Simplicity

Simplicity is required in the tools used to build applications for a middleware platform, in the programming model exposed by the middleware, in the administration and configuration of the middleware, and in the operation of the middleware.

For middleware programming models and tools, the goal is to make the use of middleware as transparent as possible, so that enterprise application developers spend the majority of their time working in the business application domain focusing on the business problem at hand. Several authors have shown that achieving full transparency of middleware is not possible [2, 11] as some aspects of distribution such as network latency and end-to-end correctness cannot be fully hidden. Approaches that have been, or are being, tried to get as close as possible to this goal include 4GLs, modeling, code generation and declarative specification.

Simplicity in administration and operation requires systems that are self-configuring, self-optimizing, self-protecting and self-healing. IBM® calls such systems "autonomic" [12]. Internally, such a system may well be more sophisticated and more complex than current generation middleware, but the system externals should be much simpler.

## 3.2 Componentization

Users need to be able to subset the full capabilities of a middleware platform in order to select a feature and footprint combination suitable to the task in hand. The large size of middleware products points to a desperate need for greater componentization of middleware to support this goal. Instead of monolithic middleware products, we need a sophisticated middleware *production line*[1] that can assemble components on demand to provide a given set of capabilities within a given operating environment. Advanced platforms may also permit runtime component selection and configuration.

Software engineering tools to analyze, separate, manage and compose the rich set of features and feature interactions typically found in middleware are immature or non-existent. The problem is hampered by middleware's performance sensitivity, which makes developers wary of large frameworks with layers of indirection.

## 3.3 Application-Middleware Independence

Middleware platforms continue to change and evolve, and large enterprises tend to acquire plenty of them [13]. Loosening the dependence of a given application on a particular middleware platform or version of a middleware product is good for both application developers and middleware vendors. Application developers can preserve their investment across multiple middleware platform iterations, and middleware vendors can lower the version-to-version migration or competitive win-back costs.

Application-middleware independence necessitates that much of the detail and complexity of middleware is hidden from the application.

# 4. THE ROLE OF AOSD

Section 3 described *what* needs to be done, but said nothing about *how* the requirements could be met. In this section we describe how a combination of aspect and component based techniques may be employed to that end. We assume that the reader is already familiar with aspect-oriented concepts.

## 4.1 Simplicity

Declarative specification is one of the most promising approaches for achieving simplicity in programming models. In this section we argue that aspect-oriented software

[1] The sophistication in the *production line* is its ability to create variants of the middleware. The resulting set of products form a *product line*, which may or may not be sophisticated. In the extreme case, the production line may produce product variants tuned for individual customers.

development is a natural fit with a declarative style, aiding simplicity by furthering its application

Declarative specification separates the declaration of the (middleware) services required from the implementation of the business logic that requires them. Declarative specifications are typically honoured by the middleware through some or all of application development-time code generation, deployment-time code generation, and runtime configuration and interpretation. The following Xdoclet [14] fragment is an example of declarative specification through attribute-oriented programming. It declares that a transaction is required to execute the method being commented on:

```
/**
 * … other comments omitted for brevity
 * @ejb.transaction
 *   type="Required"
 */
```

Attributed programming in .NET® [15], and *explicit programming* as exemplified by the ELIDE system [16] work in a similar fashion. In a post [17] to the AspectJ [18] users mailing list, Gregor Kiczales describes the techniques as a form of "early-AOP:"

*"…(the) approach requires tagging methods and classes where aspects might apply with attributes. I believe the approach [they outline] can perhaps be called early AOP, but it is missing one of the most critical properties of all other AOP systems, and this significantly limits its power. I call it early AOP because when some people first hear about AOP, this is one of the first mechanisms they propose to achieve it."*

In a full aspect-oriented approach, instead of explicitly tagging each element that is to acquire a certain property, we can separate the concerns and encapsulate (for example) the transaction policy of the system into a single unit. The elements that are to acquire transaction semantics are not individually tagged. Thus we can view, maintain, and add or remove the transaction policy of our system as a single unit. Clearly this treatment can be applied to any attribute already separated from the user application through declarative specification (it is not the intent of this paper to discuss the wisdom or otherwise of declaratively specifying transactions [19]).

Declarative specification, when coupled with an ability to interpret declarations and apply appropriate aspects to an element at either class-load time or run-time, can remove the need for code generation completely. JBOSS [20] uses this approach to apply advice to EJBs in the form of interceptors [21]. A more fully fledged form of aspect-oriented programming is promised for the JBOSS 4.0 release, which will permit interceptors to be added to methods, constructors and fields of not just EJBs, but any Java object.

In situations requiring more sophisticated capabilities than can be provided by interceptors alone (such as introducing new methods, fields or parent classes to an application domain object), aspect-orientation allows the generation of code that fully separates the concerns of the middleware from the pure application concerns. The sample code for a stateless session bean from Sun's online EJB tutorial [22] is 90 lines long, and contains only 6 lines of business application logic. It is typical of the kind of template implementation that may be generated by an EJB tool. Using aspect-oriented software development techniques such as those offered by AspectJ or Hyper/J [23], the application class can simply become:

```
public class DemoBean {
  public String demoSelect( ) throws
RemoteException {
    return( "hello world" );
  }
}
```

The application class is not cluttered with EJB-specifics. An aspect-aware EJB tool could then generate an accompanying aspect (shown here as an AspectJ example) that might look something like this:

```
aspect DemoBeanEJB {
  declare parents:
     DemoBean implements SessionBean;

  static final boolean
     DemoBean.verbose = true;

  private transient SessionContext
     DemoBean.ctx;
  …
  public void DemoBean.ejbActivate( ) {
    if (verbose) {
      System.out.println(
         "ejbActivate called" );
    }
  }
  // etc.
}
```

This clear separation between middleware specific concerns and the business logic simplifies the task of application development and greatly improves application-middleware independence. It is also a tremendous advantage for modeling tools supporting round-tripping since the user-written code and generated middleware code are cleanly separated – allowing for safe regeneration of the middleware code without any concern for loss of user updates.

The Java Aspect Components project (JAC) [24] seeks to take these ideas to their logical conclusion, replacing EJBs altogether with simple Java objects and aspect components that can be dynamically plugged into the system at runtime.

## 4.2 Componentization

Many research and commercial projects are investigating the componentization of middleware – JBOSS for example has a "super-server" architecture with componentization and configuration handled through a JMX (Java Management Extension) spine. TAO [25] is a CORBA implementation focused on high-performance and real-time scenarios. It is built on ACE [26], which can automate system configuration and re-configuration by dynamically linking services into applications at runtime. The Eclipse IDE [27] demonstrates excellent componentization through its model of features,

plugins and extension points[2]. In this section we focus explicitly on the application of aspect-oriented techniques to facilitate and further these efforts.

Aspect-oriented software development techniques provide us with new ways to modularize and encapsulate concerns that were previously entangled (or scattered) across multiple other concerns. Until a concern is encapsulated, it is very difficult to add, remove or replace that concern in a middleware system. Therefore by improving our ability to modularize, aspect-oriented techniques improve our ability to factor a middleware system into components. In a study conducted at IBM's Hursley Laboratory for example, we have shown that tracing, logging, first-failure data capture and performance monitoring instrumentation in a commercial middleware system were all amenable to modularization via aspect-oriented programming techniques [28]. Previously these concerns were scattered throughout the system. Similarly, [29] shows the use of aspect-oriented techniques within a database system for concern encapsulation.

Frank Hunleth has studied the use of AspectJ for feature and footprint management in middleware systems, using aspects to introduce features incrementally and as independently as possible [30, 31]. Lasagne [32] uses aspect-oriented software development to construct customizable middleware and distributed services, focusing on context-sensitive customizations.

A promising direction seems to be the use of several aspect-oriented techniques in combination to factor the middleware in multiple dimensions: a Composition Filter [33] like approach for simple interception based strategies, an aspect-oriented language such as AspectJ for cross-cutting concerns, sophisticated composition models such as those offered by Hyper/J for large-scale feature and system integration, and adaptive programming techniques such as those offered by DemeterJ [33] for structure-shy object relationship traversals. Such a hybrid approach was first proposed in [34].

## 4.3 Application-Middleware Independence

Current (non-AO) approaches to application-middleware independence such as the OMG's Model Driven Architecture (MDA) [13] rely mainly on abstraction and thus either reveal complexity and tight coupling at the more concrete levels (generated code or platform-specific models), or are limited in their application by an inability to express often needed details (declarative specifications and 4GLs). By separating middleware concerns from application domain concerns at all levels of abstraction, using techniques such as those promoted by aspect-oriented software development, we retain the ability to fully express an application's middleware requirements at the needed level of detail without adversely affecting coupling. Section 4.1 illustrated the use of aspect-oriented software development techniques to separate application and middleware concerns at the implementation level. The combination of abstraction and separation in the binding of applications to middleware is illustrated in Figure 1.

Quadrant A shows an abstract model of the business application with no middleware details. Quadrant B adds an abstract model of middleware requirements. Quadrant C shows a typical concrete middleware based application with business and middleware concerns entwined. Quadrant D shows a concrete middleware based application with business and middleware concerns separated.

Many applications begin and end life in quadrant C. Code generation and simple application domain modeling follows the path A → C. MDA attempts to introduce the path A → B → C, although the separation in quadrant B is not as clean as depicted. Combining abstraction and separation gives us the new endpoint D, and development path A → B → D. It can be clearly seen how the concrete, separated system in quadrant D can handle evolution or replacement of the middleware portion much more gracefully than the system in quadrant C.

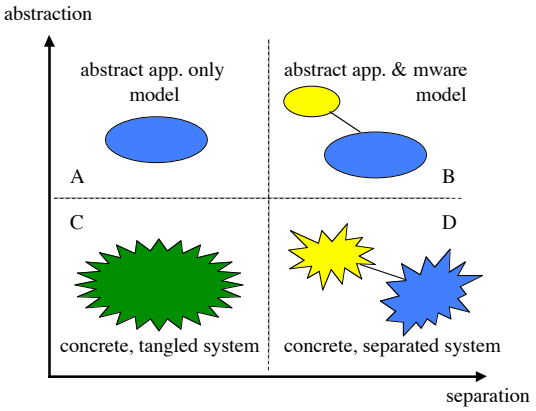



**Figure 1: Abstraction and Separation in Middleware**

The adaptive programming approach supported by Demeter provides another useful tool in the quest for application-middleware independence. It supports "structure-shy" traversal strategies that separate behaviour from structure, and hence allows inter-position of middleware components (such as wrappers and facades) in a manner transparent to the application.

# 5. THE CHALLENGE OF SCALE

As discussed in section 2.1, middleware products tend to be large – for example, IBM's WebSphere® Application Server comprises many thousands of Java classes and is developed and maintained by hundreds of staff. Many concerns in the middleware system are looked after by entire teams, and apply broadly to the independently developed work of multiple other teams. It is therefore essential to be able to specify the broad policy pertaining to a concern and where it should be applied, and then independently permit special cases (exceptions or additions to the general policy) to be specified by the owners of the affected concerns.

We envisage the production line of section 3.2 working by configuring *and composing* components to produce the required variants. A single level of configuration or composition is not tenable for software of this complexity (both because the configuration file itself would be overwhelming, and because updating the file under version control would bottleneck parallel development streams).

[2] Application development tools are an important part of a middleware platform.

Instead a "fractal" approach is required, whereby any given component may be composed of multiple sub-components, which in turn are composed of multiple sub-components and so on. The decomposition terminates with primitive (atomic) software units as determined by the metamodel of the programming language and runtime in use. Component composition is hidden from users of that component. Note that this philosophy requires that we distinguish carefully between aspect-oriented techniques that create or compose new units of existing (meta)types, and those that introduce new meta-types.

Fitting neatly with the fractal view of software composition is the observation that middleware products are not simply "compiled," but rather "built" on a production line involving many stages from initial compilation through to deployment and automated system testing. One component depends on other components, and the build infrastructure ensures that dependencies are built (compiled, assembled, composed) before the dependent component. This points to the need for the strong integration of AO-techniques for middleware with build environments, the primary of which is Apache Ant[35].

## 6. RELATED WORK

In addition to those tools and techniques already mentioned, there are many other active research projects, of which a few are highlighted here. In general, the emphasis of these projects is on enhancing the capabilities of an infrastructure platform, rather than the (more internally oriented) use of aspect-orientation to simplify the construction and presentation of existing capabilities.

DAOP [36] is a dynamic aspect-oriented platform providing a composition mechanism for integrating aspects and components dynamically at runtime. The DADO [37] (distributed aspects for distributed objects) project helps program crosscutting features in heterogeneous environments. Choi [38] shows how aspect-orientation can be used to build an open extensible container with EJB facilities. Duclos [39] extends the concepts in EJB and the CORBA Component Model to fully separate container services from business logic. In contrast, Kim [40] discusses the relevance of AOP within an existing EJB container.

## 7. SUMMARY

Enterprise computing requires distributed systems, even though distributed systems introduce considerable complexity into application development and system management. Middleware facilitates the building of distributed systems, resolving many of the lower-level problems associated with distribution and heterogeneity.

Now middleware itself is suffering a crisis of complexity and heterogeneity. This paper presents an analysis of the causes of middleware complexity, and sets a direction to return to the original focus of middleware – making distributed systems easier to build. To achieve this end the middleware community needs to focus on:

- Simplicity of application development, administration, and operation.
- Separating middleware into pluggable components that can be put together in middleware production lines to more precisely meet the needs of a given application running in a given environment.
- Loosening the ties between an application and the middleware platform(s), products, and product versions that it executes on.

We have shown that aspect-oriented software development is well suited to helping middleware address these challenges. AOSD is a natural fit with a declarative specification style, and aids in the drive for simplicity by furthering its application. AOSD also provides new mechanisms to compose software artifacts, allowing us to separate and encapsulate concerns that previously could not be easily separated. It can therefore facilitate the separation of middleware components and their subsequent re-composition to meet the needs of a given application or environment. Finally, AOSD can also help separate middleware details from application domain concerns, improving application-middleware independence.

Future directions for this work include the evolution of aspect-oriented techniques to meet the challenges described in section 5, when applied to componentization of large-scale commercial middleware. Work is also underway to investigate the role of AOSD within the OMG's MDA, which shares a common goal in application-middleware independence. This includes using aspect-oriented techniques to facilitate generation, from declarative specifications in models, of cleanly separated code implementing middleware concerns.

## 8. ACKNOWLEDGMENTS

IBM and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries or both.

Microsoft and .Net are registered trademarks of Microsoft Corporation in the United States, other countries or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

## 9. REFERENCES

1. Schreiber, R., *Middleware Demystified*, in *Datamation*. 1995. p. 41-45.
2. Emmerich, W., *Software Engineering and Middleware; A Roadmap*, in *The Future of Software Engineering 2000*, A. Finklestein, Editor. 2000, 22nd International Conference on Software Engineering. p. 117-129.
3. Bernstein, P., *Middleware: A model for distributed systems services*, in *Communications of the ACM*. 1996. p. 86-98.
4. *Java 2 Enterprise Edition (J2EE)*, Sun Microsystems: http://www.java.sun.com/j2ee.
5. Zeichek, A., *WebSphere Goes Lite (sidebar)*, in *Software Development Times*. 2002: http://www.sdtimes.com/news/068/story1.htm.
6. Geihs, K., *Middleware Challenges Ahead.* IEEE Computer, 2001. **34**(6): p. 24-31.
7. van-Steen, M., P. Homburg, and A. Tanenbaum, *Globe: A Wide-Area Distributed System.* IEEE Concurrency, 1999. **7**(1): p. 104-109.

8. Vaughan-Nichols, S.J., *Developing the Distributed Computing OS.* IEEE Computer, 2002. **35**(9): p. 19-21.
9. Coulson, G., *What is Reflective Middleware?*, in *IEEE Distributed Systems Online*. 2002: http://dsonline.computer.org/middleware/RMarticle1.htm.
10. Kirkpatrick, D., *Beyond buzzwords*, in *FORTUNE*. March 18, 2002.
11. Saltzer, J.H., D.P. Reed, and D.D. Clark, *End-to-End Arguments in System Design.* ACM Transactions on Computer Systems, 1984. **2**(4): p. 277-88.
12. *Autonomic Computing*, IBM: http://www.research.ibm.com/autonomic.
13. *Model Driven Architecture*, OMG: http://www.omg.org/mda/.
14. *XDoclet: Attribute Oriented Programming*, The XDoclet team: http://xdoclet.sourceforge.net.
15. Shukla, D., S. Fell, and C. Sells, *Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse*, in *MSDN Magazine, March*. 2002.
16. Bryant, A., et al. *Explicit Programming*. in *1st International Conference on Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands: ACM press.
17. Kiczales, G., *AOP .net?* 2002: post to users@aspectj.org, http://aspectj.org/pipermail/users/2002/001846.html.
18. Kiczales, G., et al. *Aspect-oriented programming*. in *ECOOP '97 - Object Oriented Programming 11th Europeann Conference*. 1997. Jyvaskyla, Finland: Springer-Verlag.
19. Kienzle, J. and R. Guerraoui. *AOP: Does it Make Sense? The Case of Concurrency and Failures.* in *ECOOP 2002 - Object-Oriented Programming*. 2002. Malaga, Spain: Springer.
20. *JBOSS Home Page*, JBOSS Group: http://www.jboss.org.
21. Fleury, M., *BLUE: "Why I Love EJBs"*. 2002, JBOSS: http://www.jboss.org/blue.pdf.
22. *Online EJB Tutorial: Writing the Enterprise JavaBean class*, Sun Microsystems: http://developer.java.sun.com/developer/onlineTraining/Beans/EJBTutorial/step4.html.
23. Ossher, H. and P. Tarr, *Using Multidimensional Separation of Concerns to (re)shape Evolving Software.* Communications of the ACM, 2001. **44**(10): p. 43-49.
24. Pawlak, R., et al., *JAC: A flexible solution for aspect-oriented programming in Java.* Reflection 2001, 2001. **LNCS 2192**: p. 1-24.
25. Schmidt, D., *Applying Patterns to Develop Extensible ORB Middleware.* IEEE Communications Magazine, 1999(April).
26. Schmidt, D. *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*. in *12th Annual Sun Users Group Conference*. 1994. San Francisco, CA.
27. Amsden, J. and A. Irvine, *Your First Plug-In*. 2002: http://www.eclipse.org/articles.
28. Bodkin, R., A. Colyer, and J. Hugunin. *Applying AOP for Middleware Platform Independence*. in *Practitioner Reports, 2nd International Conference on AOSD - To Appear*. 2003. Boston, MA.
29. Rashid, A. and P. Sawyer, *Aspect-orientation and database systems: an effective customisation approach.* IEE Proceedings - Software, 2001. **148**(5): p. 156-164.
30. Hunleth, F. and R. Cytron. *Footprint and Feature Management Using Aspect Oriented Programming Techniques*. in *LCTES 02*. 2002. Berlin, Germany: ACM.
31. Hunleth, F., R. Cytron, and C. Gill. *Building Customizable Middleware using Aspect Oriented Programming*. in *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. 2001. Tampa, Florida.
32. Truyen, E., et al. *Dynamic and Selective Combination of Extensions in Component-based Applications*. in *Proceedings of the 23rd International Conference on Software Engineering*. 2001. Toronto, Canada.
33. Bergmans, L. and M. Aksit, *Composing Crosscutting Concerns Using Composition Filters.* Communications of the ACM, 2001. **44**(10): p. 51-57.
34. Rashid, A. *A Hybrid Approach to Separation of Concerns: The Story of SADES*. in *Reflection 2001*. 2001. Kyoto, Japan: LNCS.
35. *Apache Ant*, The Apache Jakarta Project: http://jakarta.apache.org/ant.
36. Pinto, M., L. Fuentes, and J.M. Troya, *DAOP-ADL: An Architecture Description Language for Dynamic Aspect-Oriented Development.*
37. Wohlstadter, E., S. Jackson, and P. Devanbu, *DADO: Enhancing Middleware to support cross-cutting features in distributed, heterogeneous systems.* To Appear.
38. Choi, J.P. *Aspect oriented programming with Enterprise JavaBeans*. in *Fourth International Enterprise Distributed Objects Computing Conference*. 2000. Makuhari, Japan: IEEE Computer Soc.
39. Duclos, F., J. Estublier, and P. Morat. *Describing and Using Non Functional Aspects in Component Based Applications*. in *1st International Conference on Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands: ACM Press.
40. Kim, H. and S. Clarke, *The relevance of AOP to an Applications Programmer in an EJB environment*, in *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*. 2002.

# The Aspect-Oriented Interceptors' Pattern for Crosscutting and Separation of Concerns using Conventional Object Oriented Programming Languages

John Zinky and Richard Shapiro

BBN Technologies
Cambridge MA, USA
jzinky@bbn.com, rshapiro@bbn.com
http://quo.bbn.com http://cougaar.org

## Abstract

With disciplined use of the aspect-oriented interceptors' pattern [10], limited but effective crosscutting techniques can be used with conventional programming languages such as Java and C++. We have developed this pattern for use in Cougaar [7], a comprehensive infrastructure for supporting distributed agents. Cougaar can adapt to changes in the runtime environment, supporting such dynamic features as performance tuning, security, dependability, and agent mobility. Adaptation in this context affects not what the system does, but how it does it. Adaptive features, developed by various programming teams, must be dynamically enabled at runtime based on policy assertions and resource constraints. Adaptive features touch every part of the system, hence they are said to crosscut the dominant decomposition (which is based on class hierarchies). The pattern presented in this paper helps control these features by separating them into explicit components and by allowing the components to be attached to the base system at multiple points. The pattern shows interesting use of crosscutting, not only for ease of implementation (reuse), but also for dynamic control and composition of features. The paper presents some example adaptive features to illustrate how aspect-oriented interceptors' pattern is used in the implementation of the Cougaar agent-based middleware. The paper concludes with a discussion of how the aspect-oriented interceptors' pattern compares with emerging Aspect Oriented Programming languages.

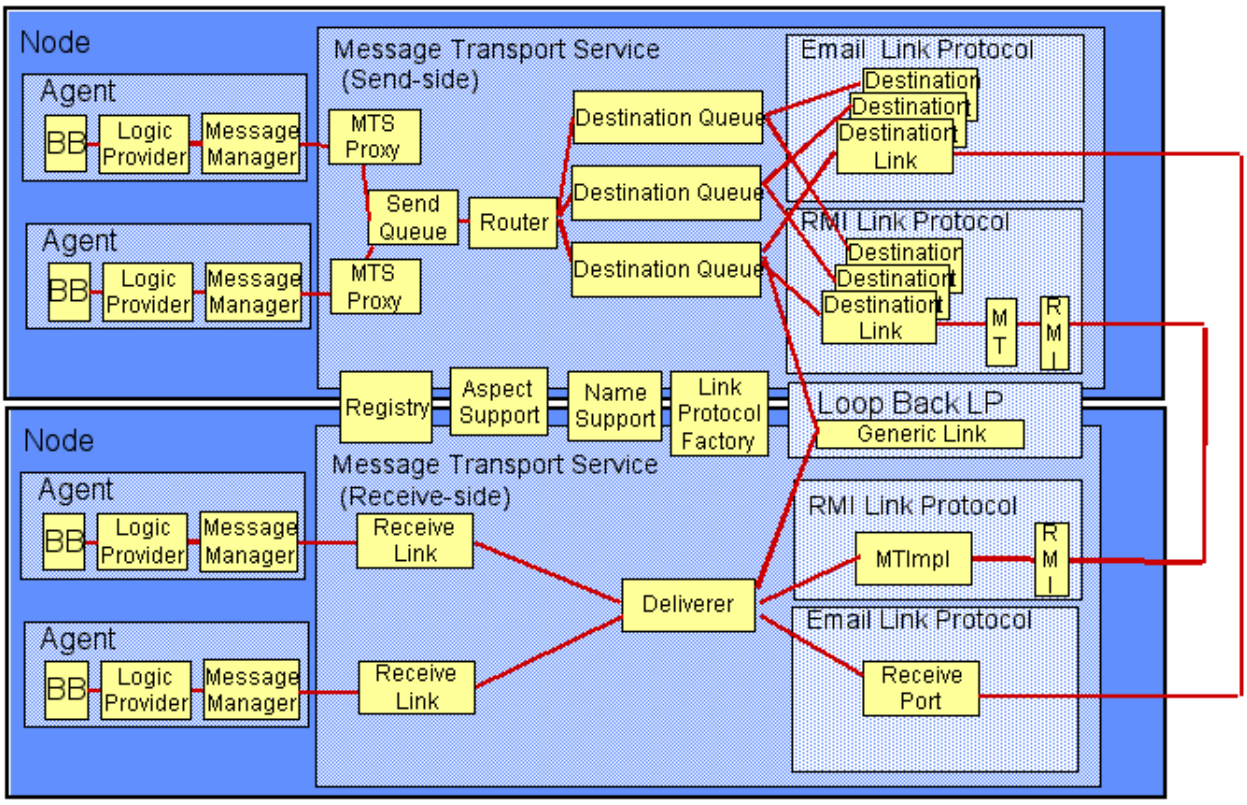
## 1. Introduction

One category of crosscutting features is concerned with system issues, such as performance, security, dependability, and time constraints. This is because the application's dominant decomposition is based on the functionality of the application (what it does), and not on the system issues (how it is done). Adaptive features gather information about policies and resource constraints from many parts of the system, local and remote. The system information is used to decide how the application should implement its functionality and must be coordinated across all the relevant parts.

Crosscutting is often equated with helping the software engineering process by allowing a finer grain of code reuse [1,2,5]. In the reuse case for crosscutting, the same block of code can be woven into many different parts of the system. This results in a kind of incremental implementation of classes from many pieces. Also, it allows the different pieces to be named and hence controlled independently at code-weave time. But code weaving does not help with runtime control of crosscutting features, in fact it may actually make that job harder. For example, in AspectJ, when an aspect is added to a class, the aspect's code fragments are added to the implementation of the class. The class name does not change, but the class now contains both the base functionality and the newly woven-in feature. If at runtime the program needs to instantiate an object without the new feature, it will not be able to do so because the base class is no longer available (because AspectJ globally replaced its implemenation). Even if the weaving process made both classes available, the solution would only work for a small number of feature weaves, until the combination of classes explodes.

Crosscutting needs to be extended beyond weave-time to allow for control of adaptive features at runtime. When objects are instantiated, the instance needs to choose which features to enable. When a client gets a reference to an object, the client should be able to choose between an object instances that has the feature and one that does not. Further, at runtime, the features themselves must coordinate the interaction among the many objects that contain them, implying that features need to be first-class objects. Adaptive features must be made explicit at runtime and they need to be named. Further, they need well-defined inputs and outputs and they need to know their dependencies on other features and on domain objects. Finally, they need to know how to connect to domain objects to get information and to assert control.

Large distributed applications, such as Cougaar [7], are written in conventional programming language, such as Java and C++. These adaptive features are developed by different software-development groups and need to be enabled dynamically at runtime. If two groups need to add their crosscutting features to the same object implementation, both need to extend the base class. Even though their features may not overlap, one feature must extend the object through inheritance before the other feature. Similarly, removing a feature requires using a different class that was not extended with that feature. This implies that the program needs to define all the combinations of features, with some features in and some features out. If a feature needs to be disabled at runtime, the right class must be chosen for all the objects involved in the crosscut at instantiation time. Also, once the object is instantiated, the feature cannot be removed without destroying the object.




## 2. Aspect -Oriented Interceptors' Pattern

The aspect-oriented interceptors' pattern [10] is about *controlling adaptive features at run-time*, rather than code reuse. The pattern enables control of adaptive features at multiple times in the application's runtime life-cycle. An adaptive feature is encoded as a class and created as an explicit object/component at runtime. Adaptive features decide at runtime what adaptive code to attach, if any. Also, they can expose interfaces for exchanging information among features and other external clients. But since the patterns can only add behavior to explicitly exposed places in the dominant decomposition of the application, the actual feature code is bound to the application and cannot be used in other contexts.

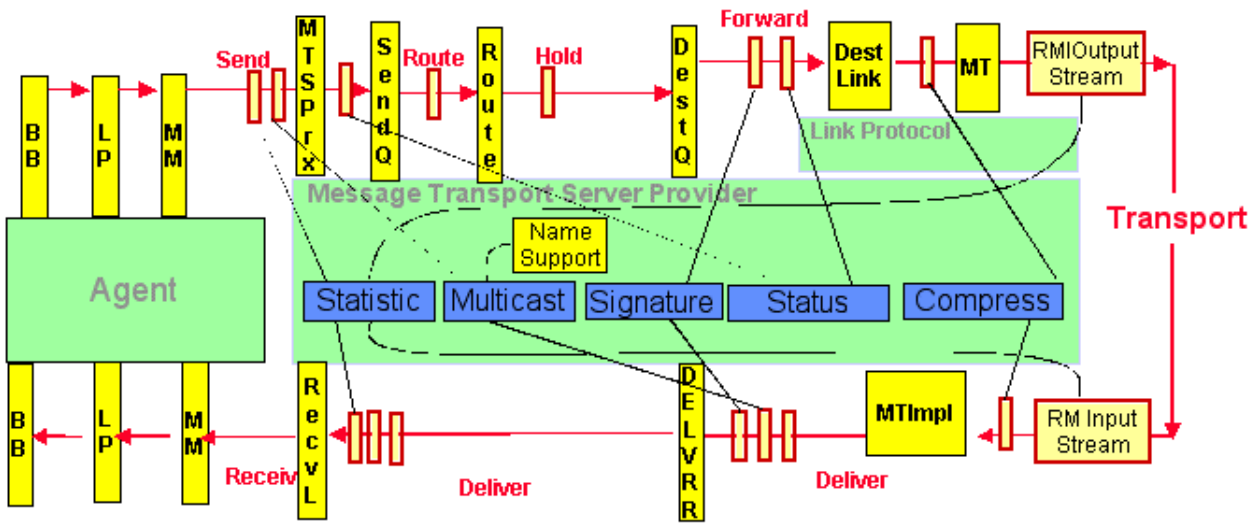
The requirements for using this pattern are very simple:

- Each of the objects over which the cross-cutting is done must have an explicitly defined interface. We call the objects that will participate in cross-cutting *Stations*
- Each Station needs a base implementation class that performs the core functionality described in its interface.
- For each interface/implementation pair, the base implementation instances should be made in a single place (effectively, a *Factory*).

An *Aspect* is then an object that can create implementation delegates for one or more Station interfaces. The Aspect class will typically include inner implementation classes for each Station interface for which the Aspect is providing a delegate. Also, Aspects may keep state.

The mechanism is very simple. Suppose the Station interface in question is *Iface* and the default implementation class is *IfaceImpl*. In the one place where *IfaceImpls* are made (i.e., the *Iface Factory*) we allow each known Aspect in turn to attach a delegate for *Iface* if it wants to. This results in a cascaded series of delegate objects, each of type *Iface*. The last object in the chain is the original *IfaceImpl*. The first object in the chain is returned as the newly created *Iface* (if no Aspect attaches a delegate, the first object will of course be the *IfaceImpl* itself). Any Aspect that wishes to attach a delegate to *Iface* would then define its own *Iface* implementation class, typically as in inner class.

One problem with this simple mechanism is that the Aspect delegates always run in the same sequence. This ordering is too restrictive. For example, in a communication subsystem, a common paradigm is that the work done in the sender needs to be accommodated in reverse order in the receiver. To handle this we added a bit of complication to the attachment of delegates. Aspects are in fact given two opportunities to attach delegates. One set of delegates will run "forward" (i.e., earlier delegates will run before later ones), the other will run in "reverse" (earlier delegates run after later ones).




**Aspect join-points** are station proxies

**Insertion** of Station Proxies happen at Station creation time

**Aspect State** is held within the Aspect object

**Export and import** additional information from Outside message transport via Cougaar Services

## 3. Example Application: Cougaar Message Transport

Cougaar [7] is a comprehensive infrastructure for supporting distributed agents. Cougaar Message Transport Service uses the aspect-oriented interceptors' pattern to add adaptive features to communication among agents. Cougaar is a large system with over 500k lines of code developed by several groups across several independent projects. Ultra*Log is one such project, designed to make Cougaar robust in the face of chaotic changes in its environment such as simultaneous system failure, security attacks, and global shifts in usage patterns. Using dynamic adaptation to manage the interaction among agents is a key feature that Ultra*Log will be adding. The Cougaar Message Transport Service will be the locus of much of this adaptation, and will be developed simultaneously by various groups.

The Message Transport architecture was designed to manage the flow of messages among agents. The internal design is open, i.e., it consists of a number of abstract interfaces with a variety of implementations. The constituents of the Cougaar Message Transport are analogous to CORBA interceptors and pluggable protocols; that is, new communication features can be added without modifying the base code. Messages flow through the abstract constituents, or "Stations", in a predefined sequence, but the behavior at each Station is determined dynamically.

The Cougaar Message Transport is divided into a dozen such Stations. In the simple case, in which all the Stations provide their default behavior and in which no errors have occurred, a message flows from one Station to the next with minimal processing. In this case the complexity of architecture may look like overkill, since most Stations act as pass-throughs, at most with buffering. In the real world, errors can occur and changing system conditions can cause the default Station behavior to be inadequate. In this more realistic case, the Stations need to do their jobs differently, but in a coordinated way. Combining the Aspect pattern with the Message Transport's open implementation provides a clean solution via crosscutting. In other words, an Aspect provides code that cuts across the dominant decomposition provided by the Station interfaces.

The Stations are described in the "top view" diagram (see Figure 1). While strict layering is not used, Stations can be grouped into the traditional communication layers. Stations handle issues for end-to-end (on the left in the figure), routing (middle), link protocols (right). Note that the "physical layer" is made from full communication stacks, such as RMI, CORBA, Email, or raw sockets. Adaptive features in the message transport tend to reimplement the classic communication services, such as addressing, flow control, retransmission, etc., but also employ extra knowledge from the host and the application domains. The reimplementation of lower-level services by higher-level services is an ongoing issue addressed by several technologies, such as micro protocols . We use crosscutting to insert the features across multiple layers.

## 4. Example Aspects

Cougaar has over 20 Aspects implemented, which handle a diverse set of adaptive features. Cougaar applications can be configured to include any of these Aspects to match the external requirements of the system. Hence Cougaar can be configured to run as an embedded controler with minimal functionality, or as a robust distributed system with security, robustness, and performance-tuning features enabled.

The following examples show some different uses of Aspects. The "side view" (Figure 2) shows how these Aspects could be combined into a specific system configuration. Note how various Aspects insert themselves into the message flow at various Stations. Aspects allow an adaptive feature to obtain access to the parts of the message flow where it needs to add behavior, and to ignore the rest.

## 4.1 Message Statistics

Instrumenting code for debugging is the classic AOP example[6]. All the trace and logging code is removed from the Stations and placed in Aspects. The observation Aspects can be hooked into any of the Stations and can correlate measurements across interfaces. Also, the summary of the observations is kept in the Aspect state. The Aspects can expose service interfaces so that their observations can be viewed by external components or other Aspects.

## 4.2 Message Multicasting

Message Multicasting detects the Multicast message type and forwards it to all the agents in a society. The Multicast messages are expanded at multiple levels. First the message is sent to all the nodes in the society and then to each agent in the node. Thus, Multicasting has to insert itself at many Stations, to convert message types, to look up the addresses of remote nodes and local agents, and to copy messages. Some of these tasks happen when Multicast messages are sent and others occur when agents register with its node or move. Thus, Multicasting crosscuts the Station decomposition.

On the one hand, Multicasting is a single, fairly simple, concept. One would expect a good software design for Multicasting to be implemented in a single class. Otherwise it's a nuisance to maintain. On the other hand, a typical message-handling system would handle sending in one class and receiving in another, for all the usual OOP reasons. Since Multicasting requires changes both on the sender side and receiver side, we cannot use traditional OOP to implement it unless we are willing to violate the first point (i.e., keeping the Multicasting code as a self-contained unit).

The Aspect pattern resolves this difficulty. By implementing Multicasting using an Aspect, the core message-handling code remains simple and stable, while all the Multicasting code lives in a single place where it is easy to maintain.

## 4.3 Message Serialization

One of the Stations exposes an interface when a message is serialized or deserialized. Different read and write filters can be added dynamically, for such things as encryption, compression, signiture, and byte-counting statistics. Some of these serialization features need to be added at both the sender side and the receiver side. For example, a compression Aspect may want to add compression to the message serialization, when the message goes to a destination that has a low bandwidth path, but not to other destination that has a high-speed connection. The Aspect must tradeoff CPU cost to compress the message against the savings in bandwidth. When compression is used, the sender-side Aspect must signal the receiver-side Aspect to add the decompression filter to the deserialization Station. Signaling is done by adding an attribute to the header of the message. The message header actually carries a list of the Aspects to be called to add filters on the receiver side. So the sender and receiver Aspect instances cooperate to dynamically add behavior to the system.

## 4.4 Heard-From status

Determining if the connection to a remote host/agent is work requires correlating information that comes from many sources. One indicator may be receiving a message from a remote agent, implies that the agent's host and communication path is working. Likewise, when an acknowledgment is received for a message sent to an agent. The heard-from Aspect inserts itself in the connections to multiple agents, determines agent's host and maintains state about when the last time the host was heard-from.

# 5. Comparison to other AOP technologies

**The Quality Objects (QuO) Project** [8] builds adaptive middleware for distributed and embedded systems. The QuO middleware offer support of QoS adative behavior at both design time and runtime. The QuO is is used to implement some of the Cougaar Aspect, helping to structure the implementation of Service Proxies and Aspect Delegates But QuO has no direct support for Cougaar Aspect objects themselves. QuO needs to be extended to handle bind-time issues and managing Aspect state based on information gathered from multiple delegates.

**Aspectual collaborations** [2] extend the concept of advice by allowing aspects to be parameterized over the class and method names that are to be advised. This extension is important for connection aspects, where connection patterns may be reused several times between Cougaar Stations. Collaborators are analogous to reusable Cougaar Aspect objects. The Collaboration roles are like the specification for where the Aspect inserts its delegates. Also, Collaborations can have their own state, just like Cougaar Aspects. The advantage of Collaboration is they are like templates that can be bound to different interfaces. Hence Collaborations could be used to reuse adaptive behavior between Cougaar Aspect that modify a specific Link Protocol, such as email or RMI.

**AspectJ** [5] implements aspects as wrappers (called "advice") that can be executed before, after, or around program points such as method calls. The Cougaar Aspect delegates are similar, but they can only wrap the interaction at the server side and not where the call is made (client-side) like AspectJ. AspectJ can also insert advice at finer granularity, than just the predefined Station interfaces.

**Subject-oriented programming** [4] and its derivative HyperJ [3] are other AOP approaches. Hyper-J allows the extraction of classes from an existing class decomposition. This allows would allow the extensions to Station classes to be developed independently and combined relatively easily. Hyper-J class are composed at class loading time, which would allow some dynamic composition. But Hyper-J does not support for adaptation at service lookup time or message forwarding time.

**Composition Filters** [1] are similar in some respects to Cougaar Aspect delegates, providing wrappers for class methods that can change class behavior. Composition filters compose with formal semantics so that they can be used to infer the composed properties from its pieces. This is needed when Cougaar Aspects begin to interact in more complicated situations. However, they

lack some of the features of Cougaar Aspects: the ability to measure and react to external, systemic conditions and coordination among filters that are inserted at several Stations.

## 6. Conclusions

The aspect-oriented interceptors' pattern developed for the Cougaar distributed agent system, allows the message transport to be extended by multiple development groups without modifying the base code. Dynamic adaptation at runtime is supported by exposing multiple times and places in the code base for which adaptive code can be inserted. Unfortunately, coordinating this code is crosscuts the dominate decomposition and new patterns were needed to keep the code maintainable and to enable the dynamic adaptation at runtime. While current AOP techniques hold promise for improving the maintainability of the crosscut code, they offer very little support for runtime adaptation. We hope that this paper will show a real world application of crosscutting and an interesting pattern for dealing with it. We hope that future programming languages will support dynamic crosscutting

## 7. Bibliography:


[1] Bergmans L, Aksit M. "Composing Multiple Concerns Using Composition Filters," Communications of the ACM, special issue on AOP, October 2001.

[2] Lieberherr K, OvlingerJ, Mezini M, and Lorenz D, "Modular Programming with Aspectual Collaborations", College of Computer Science, Northeastern University , Tech report NU-CCS-2001-04, March 2001

[3] Ossher H. and Tarr P, "Using Multidimensional Separation of Concerns to Reshape Evolving Software. CACM ) Oct 2001, pp 43. http://www.research.ibm.com/hyperspace

[4] Ossher H, Kaplan M, Katz A, Harrison W, Kruskal V. "Specifying Subject-Oriented Composition," Theory and Practice of Object Systems, Vol. 2, No. 3, Wiley & Sons, 1996.

[5] Kiczales G, Hilsdale E, Hugunin J, Kersen M, Palm J, Griswold W. "An overview of AspectJ," Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2001.

[6] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Grisold W, "Getting Started with AspectJ" CACM Oct 2001, page 59.

[7] Cougaar Distributed agent system, open source at http://cougaar.org

[8] Zinky JA, Bakken DE, Schantz RE. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, April 1997. http://www.dist-systems.bbn.com/tech/QuO

[9] Ultra*Log DARPA Program on Logistics Information System Survivability, http://www.ultralog.net/

[10] Shapiro R, Zinky J., Rupel P. The Aspect Pattern. OOPSLA 2002 Workshop. Patterns in Distributed Real-time and Embedded Systems, November 5, 2002, Seattle, Washington.

# Invasive Composition Adapters: an aspect-oriented approach for visual component-based development

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

Viviane Jonckers
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 67
viviane@info.vub.ac.be

## ABSTRACT

In this paper, we build on previous work that combines ideas from visual component-based software development with aspect-oriented software development. We introduced a composition adapter to modularize crosscutting concerns in our visual component-based methodology developed in earlier work. A composition adapter can be visually applied onto a composition pattern and the changes it describes are automatically inserted using finite automaton theory. The expressive power of a composition adapter is however limited to concerns that alter the exterior behavior of a component. To overcome this limitation, we propose to employ a new aspect-oriented implementation language, called JAsCo, tailored for the component-based context. An invasive composition adapter, which has an implementation in the JAsCo language, is able to express concerns that require more than mere filtering and re-routing. The changes dictated by an invasive composition adapter are automatically inserted into the components and composition patterns.

## 1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) argues that some concerns exist that can not be confined to one single module. Typical examples of such concerns are logging and synchronization. The research to deal with this problem is under constant evolution. Most of this research however is targeted to Object-Oriented Software Development (OOSD). As a consequence, these approaches are not very well suited to be reused in a component-based context. This paper describes our approach to introduce aspect-oriented ideas in Component-Based Software Development (CBSD) from design to implementation.

In previous research [12-15], we developed a component-based approach that lifts the abstraction level for visual component composition. This research resulted in a visual component composition environment called PacoSuite. PacoSuite improves on standard visual composition tools as it allows components to be wired together based on generic interaction protocols, called "composition patterns", rather than simple event/method pairs. To introduce aspect-oriented ideas into PacoSuite, we proposed a "composition adapter". A composition adapter transforms the original composition patterns to introduce the specified aspects. Technically, a composition adapter is applied by introducing the aspects in the glue code of a component-based application. As a result, it is impossible to introduce aspects in the components themselves. However, several experiments revealed that it should be possible to adapt the components' interior to express aspects that require more than mere filtering or rerouting. To solve this problem, we introduce a new aspect-oriented programming language targeted at component-based development, called JAsCo. An "invasive" composition adapter is an enhanced version of a regular composition adapter implemented in the JAsCo language. In this way, concerns that require adaptations to the interior of components can also be expressed.

This paper presents a complete overview of our approach. As a result, technical details of algorithms and formal foundations are not discussed. Section 2 briefly describes our component-based methodology and presents the composition adapter model using run-time checking of timing constraints as a concrete example. Section 3 briefly presents the JAsCo aspect-oriented programming language and the invasive composition adapter model is introduced in section 4. Section 5 presents the tool support we created to support our methodology. Finally, we present some related work and state our conclusions.

## 2. RESEARCH CONTEXT

### 2.1 CBSD in PacoSuite

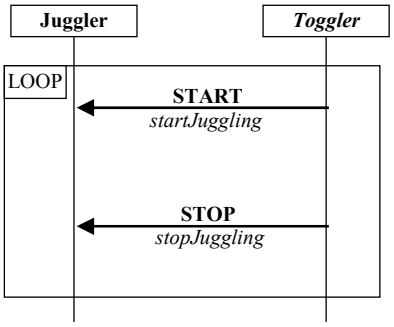



**Figure 1: Usage scenario of a Juggler component.**

We mainly focus our component-based research on lifting the abstraction level for component-based development. We want to realize the plug and play idea of component-based development. Therefore, we propose to document components with usage scenarios that specify how to employ them. A usage scenario is expressed by a special kind of Message Sequence Chart (MSC) [4]. The main difference with a regular MSC is that the signals are taken from a limited set of pre-defined semantic primitives. Each of these signals is mapped on the concrete API

that performs them. As a result, the documentation of a component is both abstract and concrete at the same time. Figure 1 illustrates a usage scenario of the well-known *Juggler* bean. One participant of a usage scenario represents the component itself and the other participants represent the environment the component expects. In this case, only one environment participant is specified, namely the *Toggler* participant. This usage scenario documents that the *Juggler* component expects consecutive start and stops. The START primitive is implemented by *startJuggling* and *stopJuggling* implements the STOP primitive.

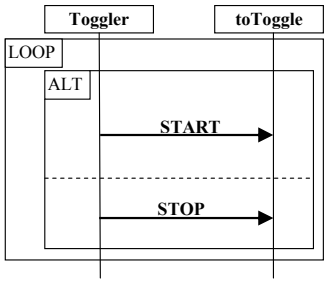



**Figure 2: Toggling composition pattern.**

We introduce explicit and reusable composition patterns that are also expressed using MSC's. A composition pattern is an abstract specification of the interaction between a number of roles. The signals between the roles originate from the same limited set of semantic primitives. This allows comparing the signals in a usage scenario of a component with these in a composition pattern. Figure 2 illustrates a generic toggling composition pattern. This composition pattern specifies that the *Toggler* participant consecutively sends either a START or a STOP to the *toToggle* participant. A possible application of this composition pattern is a simple visual interface that allows toggling the *Juggler* component from a single *JButton* component. To build this application, the *Juggler* component is mapped on the *toToggle* role and the *JButton* component is mapped on the *Toggler* role. Notice that even this simple collaboration can not be wired by most visual composition environments because the collaboration itself requires state.

The documentation of components and composition patterns allows checking the compatibility of a component with a role. The glue-code that constraints the behavior of the components and that translates syntactical compatibilities is generated automatically. Both the algorithms are based on finite automaton theory. In this paper we do not go into the details of these algorithms. The interested reader is referred to [14, 15].

## 2.2 Composition Adapters

Some concerns can not be cleanly modularized using composition patterns and components as are spread into different entities. As a result, editing, adding and removing such a concern becomes a cumbersome and error-prone task. To solve this problem, we propose composition adapters. The next paragraphs present this solution using the run-time checking of timing constraints as a concrete example. If we want to check timing constraints dynamically using our current concepts, every composition pattern needs to be adapted in the same way. Of course, when the application goes into the production phase, the dynamic timing aspect needs to be removed from the application. Consequently, the involved composition patterns need to be altered again to remove the timing aspect.

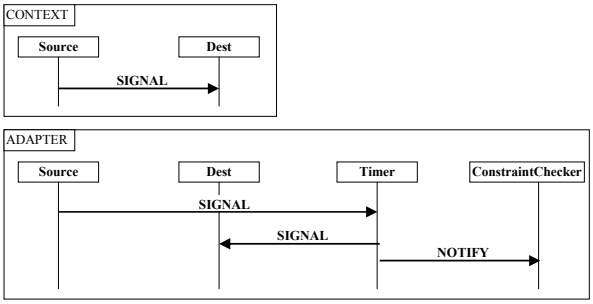



**Figure 3: Dynamic timing verification composition adapter**.

In order to modularize crosscutting concerns in PacoSuite, we introduce a new concept, called a composition adapter. A composition adapter is able to describe adaptations of the external behavior of a component independently of a specific API. A composition adapter is again documented using a special kind of MSC and consists of two parts: a context part and an adapter part. Figure 3 depicts the composition adapter that is used to modularize the timing aspect. The context part of a composition adapter describes the behavior that needs to be adapted. This can be a simple signal send as in Figure 3, but can very well be a full protocol. The adapter part specifies the adaptation itself. In the case of the dynamic timing composition adapter, every signal between the *Source* and *Dest* role will be rerouted through a *Timer* role. The *Timer* role is responsible for taking a timestamp and notifying the *ConstraintChecker* role. The *ConstraintChecker* role is responsible to verify whether every signal it is notified of, does not violate a timing constraint. The component that is mapped on the *ConstraintChecker* role could do the verification process offline and/or run on a different CPU to minimize the disruption of the system.

When a composition adapter is applied onto an existing composition pattern, the context roles of the composition adapter need to be mapped onto roles of the composition pattern. For example, suppose we want to time the communication between the *Toggler* and *toToggle* roles of the composition pattern in figure 2. The *Source* role of the timing composition adapter of Figure 3 has to be mapped onto the *Toggler* role of the composition pattern. Likewise, the *Dest* role has to be mapped onto the *toToggle* role. As a result, the START and STOP signals are not send directly to the *toToggle/Dest* role but are re-routed through the *Timer* role. After sending the START or STOP signal to the *toToggle/Dest* role, the *ConstraintChecker* role is notified.

To automatically apply a composition adapter onto a given composition we developed an algorithm based on finite automata theory. In this paper, we do not discuss this algorithm, a full explanation can be found in [13].

## 2.3 Discussion

The critical reader might have noticed that the composition adapter approach to enable run-time checking of timing constraints is not very accurate. Currently, the timestamp of the event is taken when it arrives at the component mapped on the Timer role. So, there is at least an inaccuracy because of the delay of this message send. If the application works distributed, this delay can not be neglected. Certain sophisticated component

systems use a scheduler to pass messages to components. This scheduling process imposes yet another delay, making the timestamp even less accurate. As a result, our composition adapter approach to check timing constraints at run-time is not very well suited if a high precision is desired. The only way to achieve a correct timestamp is to alter the mapped components themselves so that the timestamp is taken before a message is sent or received. However, a composition adapter is only able to alter the exterior behavior of a component by ignoring or re-routing messages. Aspects that require other adaptations can not be described using a composition adapter, which is a major limitation. To solve this problem, we enhance our current model using an implementation in an aspect-oriented programming language. The next section describes the language we designed for allowing a composition adapter to specify invasive changes of a component. Section 4 discusses how this new language is used to realize an *invasive* composition adapter.

# 3. JASCO LANGUAGE

For enhancing the composition adapter model, an implementation in an aspect-oriented programming language is required. Several AOSD-approaches, such as AspectJ [2], composition filters [3] and HyperJ [15], are available. These technologies however, mainly aim at describing crosscutting concerns in an object-oriented context. As a result, they are very well not suitable for being deployed in a component-based context, this because of several restrictions:

- Nearly all AOSD-approaches describe aspects with a specific context in mind, which limits reusability.
- The deployment of an aspect within a software-system is at the moment rather static, as aspects loose their identity when they are integrated within the base-implementation. As a result, aspects are not able to exhibit the same plug-and-play characteristic of components.
- The communication between components depends on the employed component model. Current AOSD-technologies however do not support to specify aspects on these specific kinds of interactions.

For overcoming the problems mentioned above, we propose a new aspect-oriented implementation language called JAsCo. JAsCo has been developed with CBSD, and in particular PacoSuite, in mind. The JAsCo-language stays as close as possible to the regular Java syntax, and introduces two new concepts: *aspect beans* and *connectors*. An aspect bean is a regular Java bean that describes one or more logically related hooks as a special kind of inner classes. A hook is a generic and reusable entity and can be considered as the combination of the AspectJ's pointcut and advice. A connector on the other hand, is used to initialize several logically related hooks with a concrete context. To make the JAsCo language operational, we propose an "aspect-enabled" component model, where components do not require any adaptation whatsoever for aspects to be deployed.

The following two subsections describe the syntax of both the aspect- and connector-language. For more information about JAsCo and the JAsCo Beans component model, we refer to [9].

## 3.1 Aspect Syntax

Aspect beans are used for describing functionality that would normally crosscut several components from which the system is composed. The run-time checking of timing constraints, introduced in section 2, is an example of such a crosscutting concern. Whenever a specific method is executed, a timestamp should be taken such that the defined timing constraints can be checked. Figure 4 illustrates the implementation of this dynamic timer aspect. Aspect beans usually contain one or more hook-definitions (line 17 till 32), and are able to include any number of ordinary Java class-members (line 3 till 15), which are shared amongst all hooks of the aspect. A hook is used for defining **when** the normal execution of a method should be cut, and **what** extra behavior there should be executed at that precise moment in time. For defining when the behavior of hook should be executed, each hook is equipped with at least one constructor (line 21 till 23) that takes one or more *abstract method parameters* as input. These abstract method parameters are used for describing the context of a hook. The *TimeStamp*-hook specifies that it can be deployed on every method that takes zero or more arguments as input. The constructor-body defines how the join points of a hook initialization are computed. In this particular case, the constructor-body (line 22) specifies that the behavior of the *TimeStamp*-hook should be triggered whenever *method* is executed. The behavior methods of a hook are used for specifying the various actions a hook needs to perform whenever one of its calculated join points is encountered. Three kinds of behavior methods are available: *before, after* and *replace*. The *TimeStamp*-hook specifies two behavior methods (line 25 till 31). The before behavior method describes that a timestamp should be taken prior to the execution of *method*. In addition, the after behavior method specifies that all the interested observers should be notified of the timestamp.

```
1  class DynamicTimer {
2
3    private Vector obs = new Vector();
4    void removeTimeListener(TimeListener o) {
5      obs.remove(o);
6    }
7    void addTimeListener(TimeListener o) {
8     obs.add(o);
9   }
10   void notifyListeners(Method m, long t) {
11      for (int i = 0;i < obs.size();i++) {
12        ((TimeListener)obs.elementAt(i)).
13            TimeStampTaken(m,t);
14      }
15   }
16
17   hook TimeStamp {
18
19     private long timestamp;
20                                          When?
21     TimeStamp(method(..args)) {
22       execute(method);
23     }
24                                          What?
25     before() {
26       timestamp=System.currentTimeMillis();
27     }
28
29     after() {
30       notifyListeners(method,timestamp);
31     }
32   }
33 }
```

**Figure 4: The JAsCo-aspect for dynamic timing**.

## 3.2 Connector Syntax

Connectors are used for initializing a hook with a specific context (methods or events). A hook initialization takes one or more methods or event signatures as input. Figure 5 illustrates the *TimeConnector*. This connector initializes a *TimeStamp*-hook *timer* with the throwing of the *actionPerformed*-event of the *JButton*-component (line 5), and with the *startJuggling* and *stopJuggling*-methods of the *Juggler*-component (line 6 till 7). After initializing this hook, the *TimeConnector* specifies the execution of the before and the after behavior methods. Consequently, the *TimeConnector* has following implication: take a timestamp and notify all observers of the *DynamicTimer* aspect bean whenever the JButton throws an *ActionEvent* and whenever the Juggler starts or stops juggling.

```
1  connector TimeConnector {
2                                                    Where?
3    DynamicTimer.TimeStamp timer =
4      new DynamicTimer.TimeStamp ( { onevent
5        JButton.actionPerformed(ActionEvent),
6        void Juggler.startJuggling(),
7        void Juggler.stopJuggling() } );
8
9    timer.before();
10   timer.after();
11 }
```

**Figure 5: The JAsCo-connector for dynamic timing of the JButton and the Juggler.**

# 4. INVASIVE COMPOSITION ADAPTERS

## 4.1 Documentation

One of the problems encountered with our current composition adapter model is that it is not able to express aspects that require interior adaptations of a component. To solve this problem, we propose to employ the JAsCo language as an implementation for a composition adapter. Hence, the composition adapter model needs to be altered slightly.

Figure 6 illustrates the *invasive* composition adapter that documents the *DynamicTimer* aspect bean of Figure 4. Messages in the context part of an invasive composition adapter can be mapped on a hook. In the case of Figure 6 the SIGNAL message is mapped on the *TimeStamp* hook. As a result, every message between the component that is mapped on the *Source* role and the component that is mapped on the *Dest* role will be given to the *TimeStamp* hook constructor. As a consequence, those messages are changed to take a timestamp and to notify interested observers. The adapter part of an invasive composition adapter includes a new role that represents the aspect bean in the JAsCo language. In the case of Figure 6, the *DynamicTimer* role represents the aspect bean with the same name of Figure 4. The adapter part documents what the effect of the application of the *DynamicTimer* aspect bean will be. In the example of Figure 6, every signal between a certain source and destination component is still sent in the same way. However, the *DynamicTimer* aspect bean declares that a timestamp has to be taken before an adapted method is executed (see Figure 4**,** line 28-30). This behavior is not documented in the composition adapter as it is internal to the aspect bean and no communication with other components is involved. As a consequence, this behavior is not relevant for verifying compatibility and to generate glue-code. After the original method is executed, the *DynamicTimer* aspect bean notifies a *ConstraintChecker* component that verifies whether certain timing constraints are violated (see Figure 4**,** line 32-34). This behavior however, is documented in the composition adapter because it requires communication with other components. Messages that are sent or received by a JAsCo component require an implementation mapping. In Figure 6, the NOTIFY message of the *DynamicTimer* aspect bean is implemented by throwing the *timeStampTaken* event. The implementation mapping is required to be able to generate glue-code that will call the correct method of the component that is mapped on the *ConstraintChecker* role when the *DynamicTimer* throws the *timeStampTaken* event. Notice that the component that will be mapped on the *ConstraintChecker* role does not have to understand the *timeStampTaken* event. Glue-code that translates the *timeStampTaken* event into one or more methods of the mapped component can be automatically generated using the documentation of Figure 6.

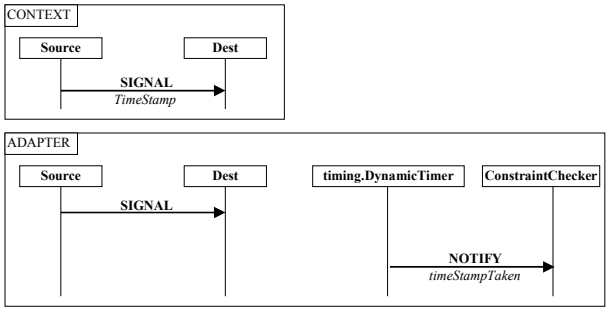



**Figure 6: Invasive Composition Adapter model for the DynamicTimer aspect bean**.

## 4.2 Applying an invasive composition adapter

An invasive composition adapter changes the composition patterns in the same way a regular composition adapter does. As a result, we can still use the same algorithm that was developed for regular composition adapters to determine the effect of an invasive composition adapter on a composition pattern.

An invasive composition adapter however also changes the components themselves through the implementation in the JAsCo language. The adaptations to a component caused by an invasive composition adapter might affect the external behavior of the component. As a consequence, the documentation of a component becomes inconsistent. To be able to still verify the compatibility of an adapted component with a given composition pattern, the documentation of this component needs to be modified. This is easily achieved by a similar algorithm as the one used for adapting composition patterns to the specification of a composition adapter [13]. The specification of an invasive composition adapter is used to alter the documentation of the components that are mapped on the context roles of the composition adapter. In this way, we are still able to check compatibility and automatically generate glue-code. In the case of Figure 6, the documentation does not have to be altered because the original behavior of the components that are mapped on the *Source* and *Dest* roles is not changed.

As a last step, a connector in the JAsCo language is generated to be able to apply the JAsCo implementation of the invasive composition adapter onto the correct components. In

order to locate the concrete methods and events the aspect has to be applied to, we have to calculate where the context part of the composition adapter occurs. Luckily, this was already determined in the previous phase. So, only the parts of the documentation of a component where the context part occurs need to be analyzed. In case of Figure 6, this means that all messages that are mapped onto the signal with the *TimeStamp* hook as an implementation, have to be altered by the composition adapter. For instance, if the *Juggler* component of Figure 1 is mapped onto the *Dest* role of the composition adapter of Figure 6, both the *startJuggling* and *stopJuggling* methods would have to be adapted. Figure 5 illustrates the connector generated when the *Juggler* component is mapped onto the *Dest* role and the *JButton* component is mapped onto the *Source* role. The onevent keyword is used because outgoing communication of Java Beans occurs through event posting. When the connector is generated, the JAsCo compiler is executed and the regular glue-code generation process of our visual component composition environment is started. As a result, the *startJuggling* end *stopJuggling* methods and the *actionPerformed* event are timed. Timing constraints that act on these points can be verified at run-time with a more accurate precision than when using a non-invasive composition adapter.

## 4.3 Small Case Study

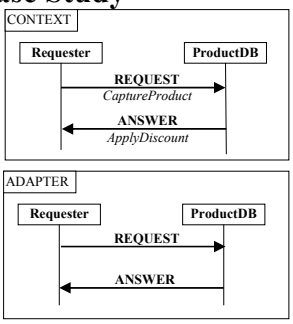



**Figure 7: OldProductDiscount invasive composition adapter.**

It can be argued that using an invasive composition adapter for specifying timing constraints validation is not really necessary. Indeed, a regular composition adapter is also able to describe this concern, only the accuracy of the timestamps differs. Therefore, we shortly present a small case-study that introduces crosscutting concerns that really require an invasive composition adapter. The case study at hand is a digital photo printing laboratory. The system consists of two sub-applications: a client that allows browsing and previewing pictures and a server application that is responsible for printing and calculating the price of an order. We identified four crosscutting concerns and successfully modeled them using an invasive composition adapter. Due to space constraints, only one of them is introduced, namely a business rule that specifies a discount for obsolete products. In this case, the obsolete product is a photo paper format that is no longer in use. To introduce this concern, extra behavior has to be inserted in the product database to be able to persistently store and use the old product information. As a result, the product database returns a discounted price for older products. Figure 7 illustrates the *OldProductDiscount* invasive composition adapter. The context part declares that this invasive composition adapter is applicable on a consecutive REQUEST and ANSWER. Notice that a different hook is mapped on both the primitives of the context part. The *CaptureProduct* hook is responsible for capturing all relevant information of the price request of a certain product. The *ApplyDiscount* acts on the answer of the request and changes the result if the product is considered obsolete. The adapter part of the *OldProductDiscount* invasive composition adapter declares that the request and answer are sent in the same way as before. Notice that the *OldProductDiscount* aspect bean itself is not documented because it does not participate in the interaction. Indeed, this invasive composition adapter only changes the interior behavior of the component that is mapped onto the *ProductDB* role.

# 5. TOOL SUPPORT

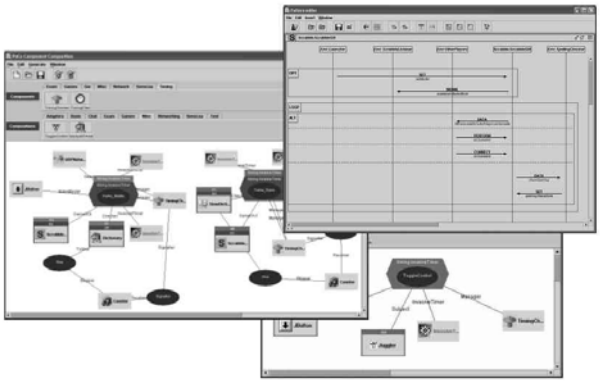


**Figure 8: Screenshots of PacoSuite. The middle left and bottom right screenshots illustrate the visual component composition environment PacoSuite. The rectangles represent components, the ovals stand for composition patterns and the hexagonal shapes symbolize invasive composition adapters. The top-right screenshot shows the documentation of a Scrabble component in the PacoDoc tool.**

The ideas introduced in this paper are implemented in a visual component composition environment called PacoSuite. PacoSuite consists of two visual tools, called PacoDoc and PacoWire, and the command-line tools required by the JAsCo language. PacoDoc is a visual editor for documenting components, composition patterns and composition adapters. PacoWire is our actual component composition environment that allows visually applying a component onto a role of a composition pattern. The drag and drop action is refused when the component is detected to be incompatible with the composition pattern. Composition adapters can also be visually applied on a given composition of components. The changes dictated by a composition adapter are automatically applied using the algorithms mentioned in this paper. In case of an invasive composition adapter, the JAsCo tools are executed transparently to the user. When all the component roles are filled, the composition is checked as a whole and glue-code is generated automatically. Figure 8 illustrates some screenshots of this tool suite.

# 6. RELATED WORK

One of the first approaches to integrate aspect-oriented software development and component-based software development is the aspectual component model of Lieberherr et al [11]. The JAsCo language was partly inspired by this work and

quite some similarities exist between both languages. They both employ a separate connector language to deploy an aspect within a specific context. On a technical level, the aspectual components approach uses byte code weaving, while we propose a new component model. Our approach improves on aspectual components by lifting the abstraction for applying aspects from the implementation level to a visual composition environment.

Filman [7] proposes dynamic injectors to introduce aspects into a given component configuration. He incorporates dynamic injectors into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. The dynamic injector approach is very similar to our non-invasive composition adapter idea because both approaches employ a wrapping and filtering technique to insert crosscutting concerns into a composition of components.

Another more recent approach to recuperate aspect-oriented ideas in component-based software development is event based aspect-oriented programming (EAOP). EAOP [4] allows specifying crosscuts on events and event patterns using a formal language. Similar to the composition adapter approach, EAOP allows specifying aspects on a full protocol of events instead of a set of methods. Since EAOP is based on a formal model, EOAP is able to improve on our approach because of the advanced detection and resolution of aspect interactions [5]. Our approach extends EAOP by lifting the abstraction level for aspect application from the implementation level to a visual composition environment.

Duclos et al [6] focus on separating crosscutting concerns in legacy systems built using CCM [3]. Similar to PacoSuite, they specify crosscutting concerns at the architectural level. They also employ two languages, one for declaring an aspect and one for describing how the aspect should be used. Aspects are applied by generating individually tailored CCM containers that include the aspect's logic. In that sense, their approach is similar to wrapping because they do not allow interior changes to the components.

# 7. CONCLUSIONS

In this paper, we introduce an invasive composition adapter in order to specify crosscutting concerns that require interior adaptations of a component on a component-based design level. An invasive composition adapter is an extended version of a regular composition adapter and has an implementation in the JAsCo aspect-oriented language. A component composer is able to visually apply an invasive composition adapter on a given component composition. The invasive composition adapter is verified to be compatible with the composition and is automatically deployed using algorithms based on finite automaton theory. Likewise, an invasive composition adapter can be easily removed from a collaboration when the concern is not desired any longer. The main drawback of this approach is that it is domain dependent. It is possible to agree on a set of semantic primitives to document component interactions for a limited application domain. However, it is unfeasible to come up and agree on a general set of semantic primitives. Another drawback is that this approach is resource intensive. Our current algorithms are of exponential nature and in worst case scenarios this could lead to state explosions. In addition, the glue-code to translate syntactic incompatibilities between components adds an extra level of indirection.

# 8. ACKNOWLEDGMENTS

We owe our gratitude to Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Also, we like to thank Dr. Bart Wydaeghe who developed the component based methodology during his PhD research. Since October 2000, Wim Vanderperren is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in Flemish: "Fonds voor Wetenschappelijk Onderzoek").

# 9. REFERENCES

[1] AspectJ Website. http://www.aspectJ.org.

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

[3] Corba Component Model: see http://www.omg.org.

[4] Douence, R., Motelet, O. and Südholt, M. A formal definition of crosscuts. In Proceedings of the 3rd International Conference on Reflection. (Kyoto Japan, September 2001)

[5] Douence, R., Fradet, P. and Südholt, M. A framework for the detection and resolution of aspect interactions. In Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (Pittsburgh PA, October 2002)

[6] Duclos, F., Estublier, J. and Morat, P. Describing and Using Non Functional Aspects in Component-based Applications. In Proceedings of the 1st international conference on Aspect-oriented software development. (Enschede The Netherlands, April 2002)

[7] Filman, R.E. Applying Aspect-Oriented Programming to Intelligent Synthesis. Workshop on Aspects and Dimensions of Concerns, ECOOP, Cannes, France, June 2000.

[8] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, September 1993.

[9] Suvée, D., Vanderperren, W., and Jonckers, V. JAsCo: an Aspect-Oriented approach tailored for Component-based Software Development .In Proceedings of the second international conference on AOSD, Boston, USA, march 2003.

[10] Lieberherr, K., Lorenz, D. and Mezini, M. Programming with Aspectual Components. Technical Report, NU-CCS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[11] Ossher, H. and Tarr, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Proc. of the Symposium on SACT: The State of the Art in Software Development. Kluwer, 2000.

[12] Vanderperren, W. A pattern based approach to separate tangled concerns in component-based development. ACP4IS workshop at AOSD 2002.

[13] Vanderperren, W. Localizing crosscutting concerns in visual component-based development.In proceedings of Software Engineering Research and Practice (SERP) international conference, Las Vegas, USA, june 2002.

[14] Vanderperren, W. and Wydaeghe, B. Towards a New Component Composition Process. In Proceedings of ECBS 2001, April 2001.

[15] Wydaeghe, B. and Vandeperren, W. Visual Component Composition Using Composition Patterns. In Proceedings of Tools 2001, July 2000.

# Aspect Component Based Software Engineering[1]

Pedro J. Clemente and Juan Hernández
Quercus Software Engineering Group. http://quercusseg.unex.es
University of Extremadura. Spain

{jclemente, juanher}@unex.es

## ABSTRACT

Component Based Software Engineering (CBSE) and Aspect Oriented Programming (AOP) are two disciplines of software engineering, which have been generating a great deal of interest in recent years. From the CBSE point of view, the building of applications becomes a process of assembling independent and reusable software modules called components. However, the necessary dependencies description among components and its latter implementation causes the appearance of crosscutting, a problem that Aspect Orientation resolves effectively. Aspect Oriented Programming allows programmers to express in a separate form the different aspects that intervene in an application which are composed adequately at a later stage. This paper analyses the problem of *crosscutting* which is produced during component development, and a component based development extension using Aspect Oriented techniques is proposed. This Component based Software Engineering (CBSE) extension has been named Aspect Component Software Engineering (ACBSE).

## Keywords

Aspect-Oriented Software Development (AOSD), Component Based Software Engineering (CBSE)

## 1. INTRODUCTION

Component-Based Development is gaining recognition as the key technology for the construction of high-quality, evolvable, large software systems in timely and affordable manners. Constructing an application under this new setting involves the assembly/composition of prefabricated, reusable and independent pieces of software called components. A component should be able to be developed, acquired and incorporated into the system and composed with other components independently in time and space[1].

The ultimate goal, once again, is to be able to reduce developing costs and efforts, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated. Component Oriented Programming aims at producing software components for a component market and for later composition (composers are third parties). This requires standards to allow independently created components to interoperate, and specifications that put the composer into the position to decide what can be composed under which conditions[1]. This approach moves organizations from application development to application assembly.

This situation has given birth to the existing commercial component models and platforms such as CCM, EJB or DCOM. However, most of the publicity surrounding these component models and platforms is oriented towards gaining the race way under between middleware architects and vendors to establish their products as standards for developing open distributed systems. Thus, whilst companies are focused on highlighting the benefits of software developing using the plug and play mechanism of their products, there is little or no discussion in the media of how to really design reusable, flexible and adaptable components.

In this sense, there are reasons in Component Based System (CBS) which cause a lack of reusability and adaptability: CBSE imposes a structure on the programs that makes it difficult to have different concerns well-modularized: code-tangling is inherent to CBSE programs[2;3]. The *uses statement* (like CCM statement) during component specification may be considered harmful. The purpose of these statements is on the specification of receptacles (i.e., a component reference in order to use the operations it provides). However, these references express an aggregation relation between components, thus establishing strong (and hard-coded) dependencies among components which make them difficult (or even impossible) to reuse, adapt and evolve.

Aspect-Oriented Software Development (AOSD) is an emerging technology that provides direct support for separating and weaving concerns that crosscut the functional components in a typical software system. Aspect Oriented Programming (AOP) has been created with the objective of allowing programmers to express separately the different concerns of an application, in order to be composed adequately at a later stage. The main characteristics of software developed with AOP are flexibility, adaptability and reusability of the elements used to compose the system [4]. AOP provides various mechanisms to express, adapt, isolate and reuse *crosscutting concerns* in the software development to obtain these main characteristics.

This paper focuses on the study of the current problems of Component Based Systems. On the one hand, each phase of CBD; that is specification, implementation, package, assembly and deployment, is revised. On the other hand, an Aspect Oriented Programming methodology to develop Component Based Systems business rules is proposed. The final software systems are composed using Aspect Oriented techniques as a “glue” among components, giving that the main advantages provided by aspect orientation to component-based systems.

The rest of the paper is as follows: in section 2, the problems arising during Component-based development will are identified. In section 3, our proposal is presented. Finally, we feature a set of conclusions.

[1] This project has been financed by CICYT, project number TIC02-04309-C02-01.

## 2. CURRENT CBS PROBLEMS

Currently, common component platforms like CORBA Component Model (CCM)[5] or Enterprise Java Beans (EJB)[6] are based in the idea of D'Souza in Catalysis[7]. This idea is simple: to build software systems using modules (components) like a builder builds a house, using independent modules. Each module has a specification and an implementation, and then, each is composed to build the final software. For this objective, the interfaces which a component provides and requires are used like the connectors in a "*lego piece*".

In this context, building applications are based on a process to compose/assemble *plug&play* components. Therefore, building an application requires the following phases: on the one hand, the description and implementation of *plug&play* components is needed (*specification and implementation CBD phases*). On the other hand, a process to interconnect and deploy components is required (*package, assembly and deployment CBD phases*). Initially, CBD methodology increases the quality of software by providing flexibility, adaptability and reusability through the assembly/composition of independent software components.

However, individual components are not as reusable and adaptable as may appear in a first place because the *crosscutting* phenomenon arises in a actual way, as we explain in the following paragraphs.

### 2.1 CBS Adaptability and Reusability

From the *adaptability* point of view, a system must be adapted when new requirements appear. This means that changes in the business rules can be applied to systems already built with minimal changes. Let us proceed to analyze how we can adapt the functionality of CBS to new requirements. First, a business rule is a process in a software system; therefore, business rule changes introduce software systems changes. These system changes can include ones in *functional* or *non-functional* properties.

**Updating Non-functional properties.** Common component platforms offer a container to manipulate the *non functional* system properties like security, persistence, distribution, etc. The container properties facilitate the development of components, because the containers offer common services for all components[5;6]. The container configuration can be changed during the *package phase.* During this phase the developer can specify various kind of policies for each component. For example, if we are developing a system using CCM, the security, transactions, and persistence for each component can be configured in *Package Phase*[5].

**Updating Functional properties.** The container can not be used to change a functional system behavior. CBS functional behaviors are specified by business rules which describe the interconnection among components. This interconnection among components will form the final system. Besides, business rules change for each system, for each domain, etc. Business rules respond to the specific problem to be solved, and they establish the specification and interconnection components in the design phase [8].

When a component A declares that uses the services offered by other component B, that declaration affects both the specification and implementation of A. This is due to the fact that the *uses statement* expresses an aggregation relation between A and B. Furthermore, in the implementation of A, direct calls to B methods appear and they are hard-coded. Consequently, changes in business rules involve updating both the specification and implementation of software components. In addition, specification changes affect to package, assembly and deployment phases of CBD.

For example, let us assume that we have developed a CBS controlling a market. Then, a new business rule for clients' authentication is required in our system. This rule describes an aspect called the *Authentication Aspect.* Suppose that the *Authentication Aspect* is implemented in a specific component (Authentication Component). In order to introduce this new aspect (it is implemented by Authentication Component) in the system we have to perform the following steps:

- First, to identify the components affected by the Authentication Aspect.
- Second, the specifications and implementations of the identified components should be updated, as it has been shown above.
- Finally, the software architecture of the system should be updated; that is to say, package, assembly and deployment phases must be reviewed.

This means that component systems are not easily adaptable to new requirements, because the introductions of new requirements involve changes in the all phases of the component life cycle, namely specification, implementation, package, assembly and deployment.

This situation happens frequently because business rules evolve and, continuously the software systems need to implement new business rules. Consequently, new mechanisms for developing more adaptable and reusable component based systems are needed, and this is the main contribution of this paper, which it is explained below.

## 3. ACBSE: BUILDING CBS USING AOP

In this section a new component based development methodology is presented. This methodology combines the principles of Component Based Software Engineering and the flexibility, adaptability and reusability characteristic are provided by Aspect Orientation. This methodology is called ACBSE (Aspect Component Based Software Engineering).

In the following paragraphs we are going to express the changes necessary to apply aspect oriented programming in each component based system development phase (design and specification, implementation, package, assembly and deployment).

### 3.1 System Design and Components Specification Phases

During the component-based system specification, the interfaces that a component provides and requires must be described. For example, in the specification of a CCM[5] component, the interfaces that it provides (facets) and those that it requires (receptacles) are described. We will focus our attention on the dependencies introduced by the *uses clause*, that is to say, the interfaces it requires from other components. There are two alternatives to define the dependencies between components: doing it during the specification phase, or leaving it for subsequent phases.

Both approaches have their advantages and disadvantages:

- If the dependencies of a component are described during the specification, they belong to that component and, therefore, they must be maintained by all implementations of that component specification [8]. With this alternative, the component provides a clear and concise idea of its behavior. However, the use of this component is quite limited. For example it is possible that in a specific framework the handling of some of the dependencies is unnecessary or, even worse, the introduction of new dependencies becomes a difficult task.
- If the dependencies between components are not represented during the specification phase. As an advantage, the components can be easily adapted to the requirements of each context. However, CBD phases concerns system architecture (package, assembly and assembly) should be reviewed to apply the component dependencies.

We propose a new way to specify component dependencies avoiding the crosscutting at the implementation phase. The different implementations of a component specification do not need to implement those dependencies.

Component dependencies are classified as *intrinsic* and *non-intrinsic*[9], since not all the dependencies have the same degree of dependency regarding to the component:

- A *non-intrinsic* dependency is when its use depends on the framework or the context in which a component is to be used. That is, if we delete a dependency from the component description, the component maintains its initial functionality without those facilities that provide the deleted dependency.
- An *intrinsic* dependency is when its description and use is vital for the component itself. In others words, if this dependency is deleted then the component loses its meaning.

## 3.2 System Implementation Phase

The implementation phase allows for implementing the component functionality. The component developer should use only the *intrinsic* dependencies, and in the component implementation there are many calls to methods to *intrinsic* interfaces. This means that each component only implements the basic business rules.

Throughout the implementation phase of the components, each component implements the interfaces it provides, as well as all the methods needed to carry out its functionality. During the implementation of these methods, dependencies can be used in the component implementation but it can only use *intrinsic* component dependencies. However, all those dependencies defined as *non intrinsic* dependencies are applied throughout the package phase of software components. Therefore, *crosscutting* is not being introduced in the implementation of the component due to *non-intrinsic* dependencies.

## 3.3 System Package Phase

During the *Package Phase*, XML descriptors (for example, Component Descriptor in CCM) are used to describe the component properties which form a part of the component system. In this XML description each component identifies the interconnections with other components; that is to say, the system architecture is described through the connection among interfaces which are provided or required by components.

The *Package Phase* allows us to apply the *non intrinsic* dependencies on the new component based system. The steps are the following:

- First, the *non intrinsic* dependencies must be defined during *System Design Phase*[9] using a graphic representation like UML.
- Second, the dependencies which have been described in UML are translated to XML Component Descriptor Specification.
- Third, this XML Component Descriptor Specification describes the *non-intrinsic* dependencies for each component. This means identifying the new business rules or new dependencies in new contexts or new domains.
- Finally, the information that a specific component describes in its XML Component descriptor is pre-processed in order to recompile the component code, and add the restrictions and dependencies that are specified in this XML Component Descriptor. These dependencies are expressed as aspect implementations through a generic aspect-oriented programming language, such as AspectJ[10], AspectC++[11], etc.

Why is the packaging phase the most suitable one to describe when and where the new business rules or dependencies should be applied? One of the principal advantages of our proposal is the flexibility that is obtained in the component design, precisely due to the fact that the dependencies with other components are not expressed in the component implementation, but indeed are once the component has been implemented according to the necessities of the context.

## 3.4 System Assembly and Deployment Phase

At this moment of the lifecycle of the CBD we have defined the component interfaces, the *intrinsic* and *non intrinsic* dependencies and the mode to interconnect the components to obtain the final system (previous section). However, given that we are designing component-based systems, which are possibly distributed, we must also represent the location of every one of the components that form the final system.

During the packaging phase a description of when and how the dependencies that are defined in the specification phase must be applied is provided. However, the location of the component that implements or provides an interface is still not specified. The UML diagrams for assembly and deployment of component-based systems[9] are translated to XML Assembly Descriptor.

An XML Assembly Descriptor (similar to CCM Assembly Descriptor) permits us to connect the component implementations, which are part of our application. It is made up of a set of component descriptors and property descriptors. That is to say, it allows for the description of components and component instances which make up a final application. This descriptor permits us to specify the location of the components or instances.

The objective of this XML descriptor is to generate a specific properties file. This file can be read for all components and the rest of the components along the net can be located. This file can use various types of locations: URL, IOR, NameService, etc.

### 3.5 ACBSE advantages

- **Reusability.** The component is not recoded when the components are used in other domains or contexts, because the component implementation can be adapted to new business rules by changing the *non intrinsic* dependencies. Then these components can be coupled with others components.
- **Adaptability.** Programmers are offered the possibility of modifying the component descriptor by altering the final component functionality.
- **Scalability.** The system can be easily scalable because we obtain new component implementations and new component specifications. Then these new components with their *intrinsic* and *non intrinsic* dependencies can be used to compose new systems.
- **Compressibility.** Developing a new system is based on following a set of structured phases (Design and Specification, Implementation, Package, Assembly and Deployment). All information about the system is stored by using the common schemas (UML or XML). Besides, the interconnection code is generated by XML translation.

## 4. CONCLUSIONS

In this paper we have presented a joined CBSE and AOP proposal in which two of the recent tendencies in software system development are united. We have expanded the life cycle of a component-based system through techniques of aspect-oriented programming with the aim of making good use of the advantages of both tendencies and obtaining more flexible, adaptable and reusable software systems.

In a component based system, the business rules establish and determine the components specification and their relations. However, these relations or dependencies provoke the appearance of *crosscutting* as we have seen in this paper.

Therefore we have detached every one of the stages in the component based development. Every one of these stages is expanded so that a new description model of dependencies between components, which are materialized during the system composition phase, is implanted.

These interconnection descriptions in XML permit us to save time and cost, due to the fact that almost the entire code necessary is generated automatically. Finally, it should be emphasized that currently the interconnection between components is totally transparent to the programmer.

This ACBSE methodology has been developed with success in the CORBA Component Model domain [12].

## 5. REFERENCES


[1] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[2] Duclos, F., Estublier, J., and Morat, P., "Describing and using non functional aspects in component based applications," *Proceedings of the 1st international conference on Aspect-oriented software development* Enschede, The Netherlands: ACM Press, 2002, pp. 65-75.

[3] Lieberherr, K. J., Lorez, D., and Mezini, M. *Programming with Aspectual Components*. 1999. Technical Report, NU-CCS-99-01, Northeastern University.

[4] Kiczales, G. *Aspect-Oriented Programming*. 1997. Procedings of ECOOP, Springer Verang. LNCS 1241.

[5] Object Management Group (OMG). *Specification of Corba Component Model (CCM)*. 1999. Web Site: http://www.omg.org/cgi-bin/doc?orbos/99-07-01.

[6] Sun Microsystems, *Enterprise JavaBeans (EJB) Specification 2.1*, Web site: http://java.sun.com, 2003.

[7] D'Souza, D. *Objects, Components and Frameworks with UML*. 2000. Web site: http://http://www.trireme.u-net.com/catalysis/.

[8] Chessman, J. and Daniels, J., *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.

[9] Clemente, P. J., Sánchez, F., and Perez, M. A. *Modelling with UML Component-based and Aspect Oriented Programming Systems*. 8-6-2002. Seventh International Workshop on Component-Oriented Programming(WCOP 2002) at European Conference on Object Oriented Programming (ECOOP). Málaga, Spain. Web Site Download: http://www.research.microsoft.com/%7Ecszypers/events/wcop2002/.

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. *An Overview of AspectJ*. 2001. Proceedings of ECOOP, Springer Verang. LNCS 2072.

[11] Gal, A., hroeder-Preikschat, W., and Spinczyk, O. *AspectC++: Language Proposal and Prototype Implementation*. 2001. OOPSLA. Web Site.http://www.cs.ubc.ca/\~kdvolder/Workshops/OOPSLA2001/submissions/17-gal.pdf.

[12] Clemente, P. J., Hernández, J., Murillo, J. M., Perez, M. A., and Sánchez, F. *AspectCCM: An aspect-oriented extension of the Corba Component Model*. 2002. EUROMICRO Conference. Euromicro Component Based Software Engineering Track. Dortmund, Germany, IEEE Press.

# Learning from Components: Fitting AOP for System Software *

Andreas Gal, Michael Franz
Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{gal,franz}@uci.edu

Danilo Beuche
School of Computer Science
Otto-von-Guericke-University of Magdeburg
Magdeburg, D-39106, Germany
danilo@ivs.cs.uni-magdeburg.de

## ABSTRACT

Aspect-oriented programming (AOP) and implementation of system software are fairly complex tasks on their own. Combining these two severe challenges seems to be not very inviting to operating system and system software builders. To make AOP more appealing to the system software community, we limit the sometimes pervasive nature of AOP by applying lessons learned from component-oriented programming to make AOP more manageable and easier to verify. The result is AspectLagoona, a featherweight aspect language with very simple semantics and easily understandable and well specified aspect and component code interaction.

## 1. MOTIVATION

At first, aspect-oriented programming [7] and component-oriented programming (COP) seem to be totally irreconcilable as far as their fundamental design principles are concerned.

Components are defined as a unit of composition with contractually specified interfaces and explicit context dependencies only [13]. A component can be understood as a black box for which it is only known how to connect to it to request a certain service (Figure 1). *How* the request is processed internally is hidden from the client. This property makes components interchangeable, as all dependencies are on the interface level only.

Aspects have a very different way of interconnecting with components. Instead of communicating with components through well defined interfaces, they have to reach directly inside the component to extract and modularize crosscutting concerns (which might very well even crosscut across component boundaries).

In general, commercial operating system builders tend to be much more reluctant to adapt new technologies than software writers in other domains. Operating system implementors are this conservative for a good reason: it is the responsibility of the OS to facilitate all I/O operations and to manage all hardware resources. While a faulty application might affect a certain single task performed by that particular application, a software error inside the operating system can easily interfere with all applications at once, causing damage to a much greater extent. Operating systems are in general also much more difficult to debug than standard applications as usually little control over the machine is left once the OS crashed.

Probably the most repulsive property of AOP for system software programmers is its crosscutting nature. While a line of component code tends to be connected to one particular action resulting in some predictable local effect, aspect code can sometimes have a subtle, sometimes even "ghostly", influence on the whole system by bypassing established communication protocols and interfaces between components and directly manipulating internal component state.

The interdependence between aspect code and component code is particularly hard to grasp as existing general purpose aspect languages like AspectJ [6] and AspectC++ [12] offer a wide variety of mechanisms for aspects to attach to component code.

To make AOP more attractive for system software developers, it is required to make aspect and component code interaction more obvious by simplifying the involved mechanisms and offering the programmer a mechanism to control where aspect code is allowed to interfere with component code and where not.

In this position paper, we present the language Lagoona [3] and its aspect-oriented extension AspectLagoona. Lagoona is an object-oriented language based on the idea of stand-alone messages and message forwarding. In Section 2 we will discuss the fundamentals of the Lagoona language. Section 3 highlights why stand-alone messages greatly simplify the understanding of aspect-component interaction while still preserving the similar level of expressibility as other general-purpose aspect languages. Section 4 discusses related work while Section 5 finally contains our conclusions and possible future extensions.

## 2. LAGOONA

The most obvious feature that sets Lagoona apart from established object-oriented programming languages is *stand-alone messages*. In Lagoona messages are bound to (declared in) modules instead of types, whereas most other object-oriented statically-typed languages subordinate messages to classes (Figure 2). Stand-alone messages prevent

*This work was partially supported by the National Science Foundation under grants EIA-9975053 and CCR-0105710.

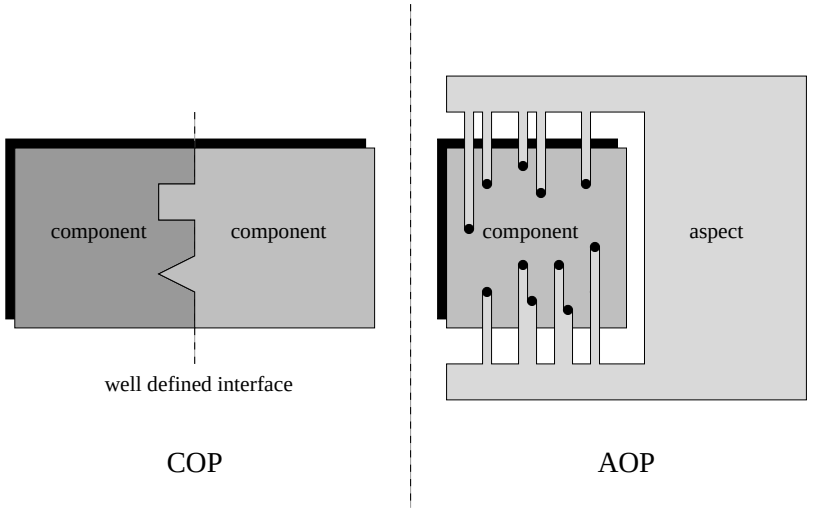



**Figure 1: Interdependencies between components and aspects**

"accidental" conformance relationships, where for example a `Cowboy` type and a `Shape` type both understanding a message `draw` with different semantics. Similar to the assumption made about interfaces in COM [10], it is assumed that messages and their specification are *immutable* once published and thus have the same meaning to any potential receiver object independently from its type.

Messages are the basis for *interface types*, (`interface` in our concrete syntax) which represent references to objects that implement a certain set of messages. Conformance to interface types is structural. The pervasive interface type `any` represents the empty message set and is the top element in the resulting type lattice. Note that the name we give to an interface type is only a convenient abbreviation; instead of using such a name, we could also declare isomorphic interface types repeatedly. Conceptually, interface types in Lagoona are used to decouple independent components [4], similar to the use of interfaces in both COM [10] and to a certain extent Java [5].

Implementation types (`class` in our concrete syntax) host methods and declarations for instance variables. While messages are *abstract operations* that describe *what* effect they achieve, methods are *concrete operations* that describe *how* an effect is achieved. In other words, messages are specifications for methods, and methods are implementations of messages. Each method implements exactly one message and is triggered when an object of the associated type receives that particular message.

While an interface type is simply a set of messages, an implementation type consists of a set of methods and associated storage definitions. In contrast to messages, methods *are* declared in the scope of an implementation type. This asymmetry is intentional, since we want to support multiple implementations of identical specifications on the level of messages and methods as well as on the level of interface types and implementation types to foster component-oriented programming. As with messages and methods, interface types and implementation types serve as specifications and implementations for each other.

To relate interface types and implementation types (including their instances), we need to define some notion of *conformance*.

First, an interface type $B$ denoting a set of messages $M_B$ conforms to an interface type $A$ denoting a set of messages $M_A$ if and only if $M_B$ is a superset of $M_A$:

$$A \trianglelefteq B \iff M_A \subseteq M_B \tag{1}$$

In other words, we employ *structural subtyping* between interface types.

Second, an implementation type $C$ with a set of methods implementing a set of messages $M_C$ conforms to an interface type $B$ denoting a set of messages $M_B$ if and only if $M_C$ is a superset of $M_B$:

$$B \trianglelefteq C \iff M_B \subseteq M_C \tag{2}$$

Third, an interface type never conforms to an implementation type. Of course, Lagoona allows interface types to be *cast* to implementation types, guarded by a dynamic check.

Finally, two implementation types only conform if they are the same type. In other words, we employ *occurrence equivalence* between implementation types.

As instance variables are internal to the associated object, at runtime, Lagoona's object model essentially reduces to a web of independent instances that communicate through messages.

Assume we are sending a message $m$ to a receiver $r$, which can be an interface or an implementation reference, whose type $R$ denotes a message set $M_R$. We distinguish two *message send operators* with different semantics.

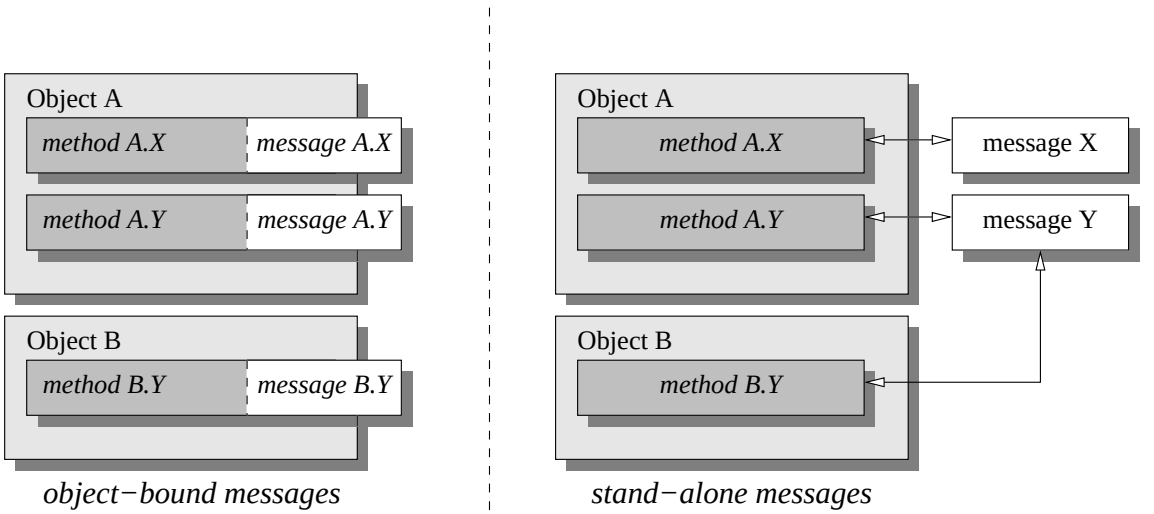



**Figure 2: Messages in Lagoona and traditional object-oriented languages**

The first operator . is *strict* in the sense that the expression $r.m$ is valid if and only if $m$ is an element of $M_R$:

$$r.m \Longleftrightarrow m \in M_R \tag{3}$$

In other words, this operator statically ensures that the message $m$ will be 'handled" by the instance bound to $r$.

The second operator ! is *blind* in the sense that the expression $r!m$ is *always* valid. Of course, we have to guard the application of this operator by a dynamic check, similar to the one for casts mentioned above.[1]

Implementation types can define a *default method* which is triggered for messages that do not have an explicit method associated with them. Inside this default method, messages can be *resent* or *forwarded* to other instances (Figure 3).

Lagoona does not contain any implicit fall back rules for the message dispatch such as inheritance. However, the programmer can easily emulate inheritance, both class-based as well as prototype-object based, using the default method and a simple forwarding statement. We use the term *generic message forwarding* to express that the actual message remains opaque during the forwarding process. The default method is implemented in a generic way and describes the forwarding action for all otherwise unhandled messages for the implementation type. Figure 4 shows a concrete code example where objects of type $A$ forward all unhandled messages to an object of type $B$. Thus, basically $A$ behaves as it would be derived from $B$ in traditional object-oriented languages.

## 3. AOP WITH LAGOONA

Adding support for aspect-oriented programming to Lagoona is surprisingly simple. As we have mentioned earlier, instance variables are internal to the associated object in Lagoona and thus objects communicate exclusively using message send operations. This removes the need for *get/set* pointcut functions, which exist in other aspect languages to capture read and write actions to attributes.

[1]For sensible assignment semantics, it is also necessary to perform a dynamic check for the generation of return values in case of messages, which are expected to produce a return value.

```
module Example {
  ...
  class A {
    B b = new B();

    void X() {
      ...
    }

    void default(message m) {
      m.forward(b);
    }
  };

  class B {
    void Y() {
      ...
    }
  };
};
```

**Figure 4: Emulating inheritance with stand-alone messages**

There is also no need to distinguish between different types of invocations such as *call* and *execution* as all complex dispatch algorithms are explicit in Lagoona. If the programmer decides to use inheritance by emulating it with forwarding, that emulation code is embedded within the default method and can be directly accessed by aspect code.

In AspectLagoona advice code is bound to messages and executed every time that particular message is being send

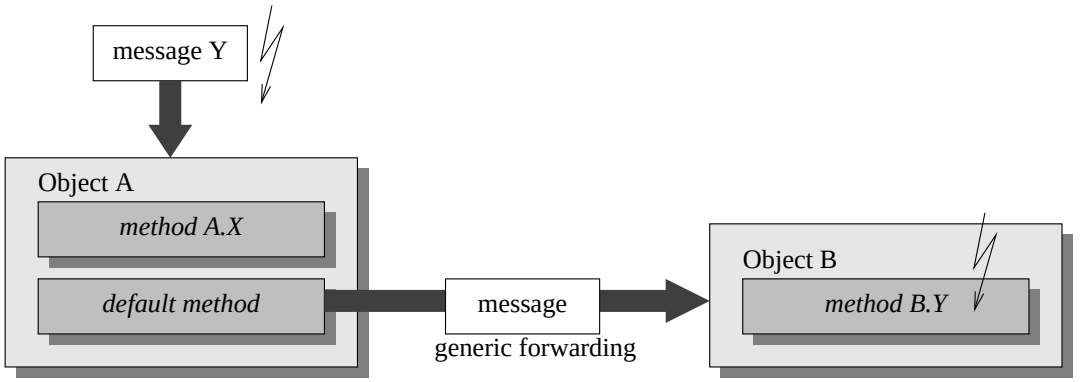



**Figure 3: Using generic message forwarding to extend objects**

to an object. As messages are unique in Lagoona, there is no need to introduce the concept of named pointcuts to describe and select parts of the class hierarchy. This has a subtle but significant impact on the code maintainability.

Languages like AspectJ and AspectC++ rely on explicit class names or name patterns to select the join points for which advice is being specified. To maintain consistency, these pointcut expression in the aspect code have to be updated every time a new class is added to the system which should receive advice from that particular aspect. Name patterns like "my*" can be used to ease this maintenance burden, allowing to capture all classes or methods having a name matching the pattern. However, this approach is error prone, if the name of a new class accidentally matches already existing name patterns in the program.

In AspectLagoona, aspects are bound to messages and not to concrete implementation types. If advice is specified for a certain method, it will apply to all methods implementing that particular message. When new implementation types are added to the system, the pointcut expression and the aspect definition do not have to be changed, because advice code is specified for messages instead of concrete methods. This feature is vital to apply aspect-oriented programming in a component-based environment, where new components can be added on the fly.

There is also no need to introduce a novel construct for pointcuts and pointcut expressions. A pointcut is a set of join points. In Lagoona the only meaningful join points are messages and Lagoona already supports the notion of a set of messages: *interfaces*. Figure 5 demonstrates how advice code is declared in Lagoona.

In contrast to other aspect languages where privileged aspects can inject code inside components, in Lagoona public interfaces exported by components serve as join points. This decouples the aspect code from the component code and allows the component to hide its internal structure to the degree it is necessary to provide interchangeability. Effectively, interfaces become not only the well-defined specification for inter-component communication, but also for the interaction between aspects and components (Figure 6).

```
module Application {
  interface NeedLocking {
    void print(char c);
    void flush();
  };
  class Screen {
    void print(char c) {
      // lock must be held
      ...
    }
    void flush() {
      // lock must be held
      ...
    }
  };
};

module Locking {
  advice Application.NeedLocking {
    void before() {
      // aquire lock
    }
    void after() {
      // release lock
    }
  }
};
```

**Figure 5: Defining advice code in Lagoona**

## 4. RELATED WORK

A number of powerful aspect-oriented languages exists today, including AspectJ [6], AspectC++ [12], and more recently AspectC# [8]. Unfortunately, none of them is geared to accommodate component-oriented programming and aspect-oriented programming at the same time.

AOP has been successfully applied in the operating systems domain by a number of projects. The PURE family of operating systems uses AspectC++ to implement interrupt synchronization for deeply embedded systems [9]. The a-kernel [2] project uses AspectC, a subset of AspectJ to modularize certain crosscutting concerns in the FreeBSD kernel.

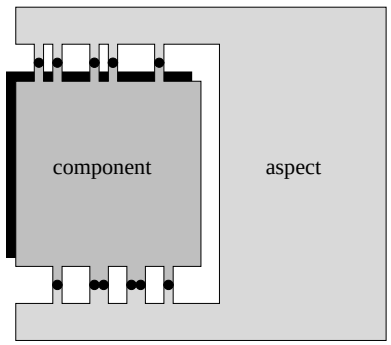



**Figure 6: Well-defined aspect/component interface**

Both projects operate on a monolithic operating system kernel and do not address component-based systems.

The Aspect-Modulator Framework [11] also permits to modularize C++ OS code with aspects. An aspect modulator is responsible to execute advice code where appropriate. The aspect modulator is invoked from join points generated manually through code insertion. Bossa [1] uses event-based AOP to modularize OS schedulers. Events are inserted manually into the source code and aspects can choose to subscribe to them. An interesting feature of the event-based AOP model is the possibility to dynamically enable and disable aspect code.

## 5. CONCLUSIONS AND FUTURE WORK

Implementors of system software are reluctant to adopt novel programming mechanisms and paradigms, unless the new technology is handed to them in manageable pieces. AspectLagoona aims exactly at these hard-to-convince users by offering a lightweight mechanism for aspect-oriented programming that still allows to deal effectively with all applications of aspects that the authors of this paper have encountered so far.

On the other hand AspectLagoona does not try to compete directly with AspectJ or AspectC++, as it is in contrast to the latter two aspect languages a complete new language design with a different object-oriented core language. This disqualifies AspectLagoona for the re-engineering of legacy operating systems, which seems to be currently the most pursued approach in pushing AOP into the commercial OS market.

The lesson we are trying to learn from AspectLagoona is what the smallest and least invasive approach to extend a languages with AOP capabilities would be. We are confident that such a minimalistic approach has a good chance of being accepted in certain areas of the system software domain.

Our current implementation of AspectLagoona includes a complete compiler and runtime system, all written in Lagoona. Recently we started to re-engineer parts of the compiler and especially the runtime system to make use of the aspect-oriented features of the language, which were initially not present.

As far as future work is concerned, we plan to rework the advice activation infrastructure, which is currently compile-time driven. Our goal is to fully integrate aspects into the component framework of Lagoona, allowing to compose systems from binary components *and* precompiled aspects at deployment time or even dynamically.

## 6. REFERENCES

[1] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming os schedulers with domain-specific languages and aspects: New approaches for os kernel engineering. In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Apr. 2002.

[2] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects: using AOP to improve OS structure modularity, 2001.

[3] M. Franz. The Programming Language Lagoona: A fresh Look at Object-Orientation. *Software - Concepts and Tools*, 18(1):14–26, 1997.

[4] P. Fröhlich and M. Franz. On certain basic properties of component-oriented programming languages. In D. H. Lorenz and V. C. Sreedhar, editors, *Proceedings of the Workshop on Language Mechanisms for Programming Software Components (at OOPSLA)*, pages 15–18, Tampa Bay, FL, Oct. 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[6] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*. Springer-Verlag, June 2001.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[8] H. Kim. AspectC#: An AOSD implementation for C#. Master's thesis, Department of Computer Science, Trinity College Dublin, Sept. 2002.

[9] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schrder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the pure operating system family. In *Proceedings of the 5th ECOOP Workshop on Object-Orientation and Operating Systems*, Malaga, Spain, June 2002.

[10] Microsoft Corporation. *The Component Object Model (Version 0.9)*, Oct. 1995.

[11] P. Netinant, T. Elrad, and M. E. Fayad. A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization. *Communications of the ACM*, 44(10):83–85, 2001.

[12] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)*, volume 10 of *Conferences in Research and Practice in Information Technology*. ACS, 2002.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM, 1998.

# AOP Support for C#

M. Devi Prasad
Manipal Center for Information Science
Manipal Academy of Higher Education
Manipal – 576119
Karnataka, India
Telephone: +91- 08252-571914
devi.prasad@mahe.manipal.edu

B.D. Chaudhary
Department of Computer Science
Motilal Nehru National Institute of Technology
Allhabad – 211004
Uttar Pradesh, India
Telephone: +91-532-2541006
bdc@mnrec.ac.in

## ABSTRACT

We have extended the C# compiler available under Microsoft's Shared Source Common Language Infrastructure (SSCLI) to facilitate Aspect Oriented Programming. The resulting compiler targets Microsoft .NET architecture. Our implementation introduces new ideas into the aspect language and the aspect-weaving mechanism. Our AOP extensions emulate AspectJ programming model and augment it with constructs that harness facilities provided by the Microsoft's .NET architecture. In particular, our framework allows aspect definitions to introduce 'attributes' on base C# module elements. This allows .NET runtime to provide container services transparently to marked modules and module elements.

Our aspect weaver brings novelty to the weaving phase. It allows configurable aspect ordering and selective aspect weaving. Selecting aspects of interest from a set of defined aspects and specifying suitable physical order for advice weaving is externalized in an XML file. Aspect scheduler determines a weave plan based on this specification and the weaver carries out this plan.

Modifications to the original compiler involved reorganizing the source code so that we obtain better modularization promoting reuse, or reduce coupling with the base code. In some cases, we extracted classes used as implementation helpers in the base compiler and turned them into new reusable abstractions.

In order to implement efficient traversals between abstract syntax graph and a graph representing semantics-verified method bodies, we had to extend data structures for syntax and method-body graphs. The solution introduced cyclic references across graphs. Therefore, we had to define new protocols for memory management so that these graphs representing method bodies are preserved even after their semantic checking. We altered the memory ownership scheme so that the base compiler and our extension subsystem coordinate memory management concerns.

Current (incomplete) implementation is around 2 thousand lines of C++ code over and above Microsoft's base compiler code. This implementation performs only a source to source translation. It has taken about four months for four people to make stable extensions. There is no public release of the implementation available as yet.

## 1. INTRODUCTION

This paper summarizes the novel features of our AOP extensions to C# language [3]. It also reports the experience gained while restructuring and enhancing a shared source compiler. Here we describe a general global view of this project, named CAMEO. The initial aim of CAMEO is to implement AspectJ like language support enabling aspect-oriented modularization in C#. Other goals include support for structural aspects that harness Common Language Runtime (CLR) features [1], incremental or partial aspect weaving, and configurable advice weaving. A preliminary source-to-source translator implementation is available for internal use. We intend to evolve this framework for exploring new ideas in AOP and metaprogramming.

Determining a collection of joinpoints in the base source involves performing a detailed control flow analysis of the code. Conducting flow analysis directly on the source text is an expensive operation in most practical cases. Since traditional compilers routinely parse source text and build Abstract Syntax Graphs (ASGs), it makes sense to make use of available infrastructure from implemented translators. In the CAMEO project, we counted on Microsoft's Shared Source Common Language Infrastructure (SSCLI) implementation to meet these requirements.

SSCLI [2] is an implementation of Microsoft's CLR architecture. Apart from the implementation of a Virtual Execution Engine and host of other tools, it includes C++ implementation of a C# language translator. The latter is a complete implementation of the ECMA standard [3] and its source code is available for modifications only for academic and research purposes. The SSCLI provided compiler generates Microsoft's Intermediate Language (MSIL) code. Because the compiler is a tiny part of the larger framework, it is inextricably tied to the infrastructure. SSCLI itself is spread around some 9000 files and the compiler source occupies nearly 200 files. We decided, very early in the project, to separate compiler code per se from rest of the SSCLI. That way, building the compiler did not require 'make'ing the entire SSCLI and it also eliminated dependencies on other tools.

The base compiler contains a collection of classes and Component Object Model (COM) components implemented in C++. It comprises lexical analysis, parsing, semantic checking, and code generation phases. The lexer reads entire source file into the memory and creates a token stream. Eventually all source files are tokenized and stored in core. Parser operates on this token stream to create an Abstract Syntax Graph (ASG). The semantic analyzer builds a Symbol Graph (SG) for namespaces, classes, methods, and other structural language-elements while performing semantic checks on types and inheritance hierarchies. It also builds an Expression Graph (EG) while binding method bodies. From there, the code generator emits MSIL executables. Totally, there are about one hundred different kinds of syntax

nodes, symbol nodes, and expression nodes. By referring to these three graphs, it is possible to regenerate the original source without any loss of information.

We turn off code generation from the base compiler and redirect control to CAMEO subsystem. CAMEO will then go through a series of phases that operate using three different data sources: the abstract graphs generated by the base compiler, aspect definition files, and an aspect configuration file. The output of CAMEO is a morph of original C# source program with aspects woven into it. Figure 1 shows the normal flow of data and control between the subsystems and phases within them.

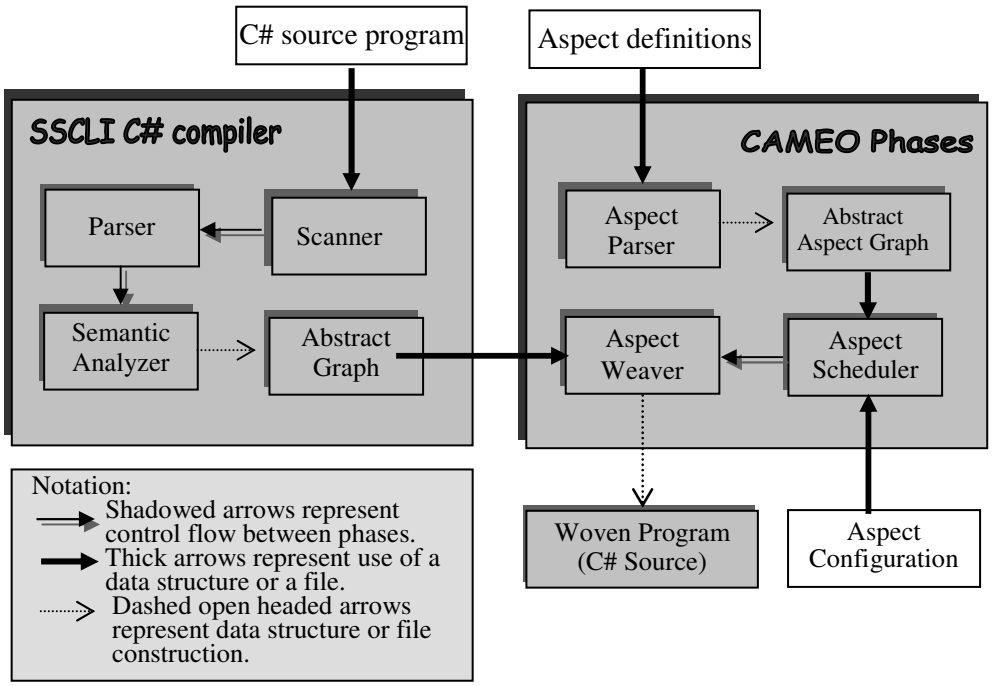



**Figure 1 - Flow of control across CAMEO aspect weaver**

The aspect parser builds an abstract aspect tree (AAT) from aspect definitions. This tree represents all inter-type introductions, pointcut declarations, or local member definitions found in the body of an aspect declaration. This tree is used in aspect selection and weaving stages.

CAMEO weaver takes an approach fundamentally different from that of AspectJ [4]. The latter uses '**dominates**' clause to define precedence relationships among advice encapsulated by different aspects. We feel such "concerns" about advice ordering must be separated from aspect definition per se. In CAMEO, the description of precedence relations is externalized in an XML based configuration file, much in the spirit of descriptions supplied to the popular 'make' tool. Each 'rule' in the configuration file specifies dominating rules and aspects that are part of this rule. Aspect scheduler plans a weaving order based on this information. Aspect weaver uses this plan to weave advices into the base code.

In AspectJ all aspects from an aspect definition file are automatically included for weaving. The only way to avoid particular aspects is to physically separate their definitions into different files and exclude these files from the compilation unit. In CAMEO, we have an improved method for excluding aspects from a weave step. CAMEO uses an external configuration file containing aspect composition rules. It is much simpler to specify and maintain appropriate rules for combining required aspects than to configure physical compilation units. This simplified method for separating aspect selection concern from its definition helps in building variants of a base system in a side-by-side, configurable fashion.

The rest of this paper is organized as follows: in section 2, we explain some novel features of our aspect language and the weaver. In Section 3, we discuss the challenges faced in implementing these features on top of the existing translator. In Section 4, we compare the overlapping ideas between CAMEO and a popular dynamic AOP framework. In the final section, we provide a summary of the work.

## 2. ASPECT LANGUAGE SUPPORT AND WEAVING MODEL

In order to maintain conformance with contemporary aspect technology, we decided to make CAMEO weaver emulate AspectJ. Moreover, C# is quite similar to Java at various levels. Hence, the syntax for aspect definition, pointcut declaration and advice declaration in CAMEO remains similar to AspectJ with minor differences. We plan to support all primitive pointcuts of AspetcJ in CAMEO. There are certain idiosyncrasies in C# and .NET that call for additional pointcuts to pick up special joinpoints and special weaver behavior. We provide a representative list of specialties in CAMEO here:

— The '**get (…)'** and '**set (…)'** primitive pointcuts in AspectJ pick up those joinpoints that access a non-private member of a Java class. On the other hand, C# has a special syntax for defining property accessors and mutators, in addition to the ordinary field accessors. In C#, property members resemble methods of a class, both in syntax and semantics. Therefore, we should treat property members of class, as well as non-private fields of a class, orthogonal. These two pointcuts, therefore, need different treatment from that of AspectJ.

— C# programs can use "unsafe" pointers. The '**unsafe**' primitive pointcut picks up joinpoints from the base code that use pointers.

— The inter-type declaration can introduce new attributes on existing classes, members of a class, return type and formal parameters of methods.

We feel these joinpoints are interesting to both designers and developers under the .NET architecture. Code using unsafe pointers could be compromising reliability for functionality. We can control tensions between safety and functionality using '**declare error**' or '**declare warning**' constructs within aspect definitions, in a case-by-case fashion.

The next three subsections bring out important ideas that distinguish CAMEO from other contemporary aspect weavers.

## 2.1 Structural Aspects Targeting .NET Architecture

AOP frequently uses inter-type 'introduction' mechanism to affect structural and behavioral changes in classes. An aspect definition can extend some class declaration to implement one or more interfaces, and introduce new methods and fields into that class. When components participate in specific patterns of interactions, AOP helps in implementing protocols in a non-invasive way by introducing required members and interfaces on partaking classes. Introduction in CAMEO works transparent to the target classes.

In some cases, stylistic naming conventions are strictly followed in naming methods and fields of classes. An external framework can later use reflection or static analysis techniques to provide certain services to methods that are stylistically named. Many software-testing frameworks follow this approach to automate testing. The use of declarative 'attributes' in .NET avoids some well-known problems with stylistic naming conventions [6]. Nunit Version-2 [7], a unit-testing framework for .NET languages, extensively uses attributes. Applications can also define custom attributes to annotate classes and methods and reflect upon them to provide services or modify behavior at runtime [5].

In CAMEO, we have a provision to introduce attributes on classes, methods, parameters, and fields. A statement such as the following one, defined inside an aspect declaration, introduces new attribute 'Test' on 'PushTest' method of 'Stack' class that is visible in the 'org.example' namespace.

```
introduce [Test ] on public org.example.Stack.PushTest (…);
```

Following construct introduces a new private field named 'url' into the class 'org.example.Machine'

```
introduce private String url on org.example.Machine;
```

The base compiler maintains a sequential in-core token stream representing the source program. Inter-type declarations modify physical structure of a class after the creation of token stream. Consequently, we cannot efficiently update the sequential buffer when introductions arbitrarily modify the token stream. In CAMEO, we create new abstract syntax nodes to represent the synthesized attribute or member declaration. Then we hook them into the target (which can be a class, method, formal parameter or return type) node in the original ASG. Synthesized nodes lack matching source text in the token stream.

This poses an interesting problem while unparsing, which is the last phase in CAMEO. There we need to reconstruct source text from woven ASG and EG. Because we cannot directly map synthesized nodes to text tokens, we should explicitly implement an 'Unparse()' method to generate source text from their abstract internal representation. We have generalized this technique and applied it to all types of abstract nodes. An important benefit of this approach is that in the future we can do away with the token stream making way for flexible solutions.

## 2.2 Configurable Aspect Ordering

CAMEO uses an XML file containing weave rules. Each rule logically stands for a concern that is implemented in terms of related aspects. In some cases, it also specifies requirements regarding the order of execution of advice. Each rule has two parts. One part lists aspects central to the concern represented by that rule. We treat a rule to 'own' aspects listed under it. The other (optionally empty) part specifies precedence relation among aspects owned by this rule, and other 'interfering' aspects. For example, a logger concern may take precedence over a security concern. A typical rule reads:

```
<rule name = 'XmlSerializer'>
  <precedence type = 'strict'>
    <dominatingRule name = 'Logger' />
    <dominatingRule name = 'SecurityManager' />
```

```
    </precedence>
    <weave>
      <aspect name = 'XmlWriter' />
    </weave>
  </rule>
```

This example says that rules 'Logger'and 'SecurityManager' dominate the rule 'XmlSerializer'. Moreover, the attribute *type = 'strict'* on the 'precedence' element specifies that the aspects listed by "Logger" (which is not shown here) must be weaved before the aspects under "SecurityManager" (again, not shown). If *type* attribute is 'lax', the weaver is free to weave the aspects under the rules 'Logger' and ''SecurityManager' in arbitrary fashion. In the example, aspect 'XmlWriter' is weaved only after the aspects owned by 'Logger' and 'SecurityManager' rules are completely considered.

The aspect scheduler computes the closure of rules starting from a 'head rule' representing the root of precedence relationships. This step should yield an acyclic directed graph. A cyclic graph implies circular dependency of rules and aspect relations, which is illegal. Consistency checks on this graph detect conflicts in aspect precedence relations. We obtain a flattened list of owned aspects from the closure of rules. This list represents a complete weave order. If there is any non-consecutive repetition of aspect names in this list, we flag the precedence requirement illegal. In the current implementation, we always assume strict ordering for aspects. A non-consecutive repetition of an aspect $\mathbf{a}_j$ across two different rules implies that at least two aspects for distinct concerns have incompatible expectation from $\mathbf{a}_j$. When $\mathbf{a}_j$ appears consecutively, it is folded into one instance.

## 2.3 Selective Aspect Weaving

While integrating software developed using AOP techniques, testing becomes easier if aspects are propagated in a staged manner. By guiding the weaver to deal with few selected aspects, it is possible to avoid aspect 'interference'. In CAMEO, we can introduce fresh set of rules into the configuration file, any time it is necessary to control aspect impact on the base code. These new rules should capture only the desired aspects and their inter-relationships. The command line argument to compiler should indicate the new head rule for aspect selection. As discussed in the previous section, we obtain a flattened list of owned aspects from the closure of head rule.

The aspect scheduler refers the aspect closure graph, described in section 2.2, to determine the list of aspects relevant to the current weave step. In the most general case, the scheduler constructs a list that honors aspect precedence relations. The current implementation handles only strict precedence rules, mentioned in section 2.2.

The weaver receives abstract aspect tree and precedence list from the aspect scheduler. The precedence list represents the weave order for aspects of interest. At first, the weaver considers inter-type introductions from each aspect. The weaver performs field introductions, followed by method and property introductions and then attribute introductions, in that order, on the target classes. Introductions modify the in-memory ASGs. Then the weaver processes pointcut declarations. This step performs a flow analysis of the transformed ASG in order to identify potential sites for advice weaving. Finally, the advice weaver carries out required modifications by inserting code at appropriate locations in the ASG.

# 3. OTHER IMPLEMENTATION CONCERNS

Some of the modifications discussed in the previous sections posed complex engineering problems. We mention these problems in this report because they represent a class of concerns that are best candidates for aspectual modularization. We hope that an understanding of such concerns helps better design of traditional software.

A careful study of the SSCLI C# translator source reveals numerous crosscutting concerns. Important among them include incremental compilation, thread synchronization (during incremental compilation), symbol table creation, memory management, compiler options influencing different phases, error reporting, executable file creation, and COM support

It is clear that the above concerns contribute to the efficiency of translator implementation. In general, efficiency concerns are scattered and tangled inside compiler's implementation. A minor modification to one feature escalates changes in many modules, either across multiple methods within a class or across class hierarchies.

For instance, the base compiler uses a class named 'CLSDREC' that maintains information necessary to declare a single class. This class has a method named 'compileMethod' that builds a parse tree for a method definition and generates the intermediate language (IL) instructions for it. For obvious efficiency reasons, its implementation releases memory held by the parse tree after IL generation. Though we have switched off the code generation phase, we would still like to exploit interior parse tree for its rich semantic content. In the base compiler, a COM class implements the interior parse tree. Each object of this type maintains references to other classes that represent source module and related abstractions. We changed the implementation so that CAMEO takes up ownership of allocated memory. This change in memory management policy manifests as two scattered blocks of code. First, we modify the flow within 'compileMethod' to avoid deallocation on successful compilation. We allow a deallocation only when compilation fails. Next, as a last step in CAMEO, we iterate every method node in the ASG and release the reference to the parse tree component it holds. This is necessary because we have to honor COM's reference counting requirements.

In some cases, we were able to refactor code from base implementation and reuse it effectively. For example, there is a class named 'CSourceText' that was used in the base compiler as an implementation helper for buffering C# source programs. This class can read

content encoded in UTF8, UNICODE or ASCII. We extracted its class declaration and implementation into different physical files and reused it for reading aspect definitions. Although this class uses an inflexible buffering strategy, its design simplicity facilitated good reuse.

## 4. RELATED WORK

At present, the choice of weaving strategy appears to be one of the important distinctions among different AOP languages and frameworks. Languages based on AspectJ model employ static weaving. Java Aspect Components (JAC) [8] represents an approach that exploits runtime infrastructure to provide dynamic composition of aspects. Some enhancements claimed by CAMEO overlap with that of JAC [9]. However, there are important differences in realizing these enhancements. Therefore, in this section we briefly bring out essential distinctions between JAC and CAMEO.

JAC is a powerful framework comprising of a runtime infrastructure for dynamic aspect composition. JAC does not extend the Java language. Instead it works directly on the java bytecodes. Its programming model consists of four important elements: a base object, an aspect component, an application specific weaver object, and a wrapping controller. All four are pure java objects. JAC uses Javaassist [10] to intercept requests to load a class and constructs a run-time environment suitable for composing objects with aspects. Each aspect component is written using a reflective API that helps in coordinating the execution of multiple aspects wrapped around a base object. This API also helps an aspect component to probe runtime call stack and determine method call context. The weaver object uses reflection to wrap base objects with aspect objects. An application specific wrapping controller assists in dynamically verifying the consistency among aspects, in defining precedence among aspects in a modular way. JAC derives its effectiveness from the powerful reflection based API.

In contrast, CAMEO employs static weaving, provides C# extensions, and works with source code. It does not use any runtime infrastructure or reflection, nor does it deal with bytecode weaving. The CAMEO programming model does not directly deal with consistency and precedence among aspects. Instead such issues are handled by externalized declarative specifications. These preferences have to be known at compile time.

## 5. CONCLUSION

CAMEO employs Microsoft's SSCLI code base for building experimental AOP infrastructure for C#. As an ongoing project, it intends to serve as a tool for advanced separation of concerns for .NET application development. The current implementation does not handle all pointcuts available in AspectJ. Nor does it support byte code level weaving. We have a few ideas on the paper for adding new joinopints to the existing repertoire of CAMEO that simplify constructing a class of structural design patterns. We have plans to refine the ideas of aspect configuration to handle lax ordering among different aspects. We also intend to replace existing in-memory token stream by a more flexible scheme so that inter-type introduction, advice interlacing becomes more efficient.

The main contribution of this project is in exploring mechanisms to separate aspect definitions from aspect weaving concerns. We have demonstrated that selective aspect weaving and configuring their weave order is both attractive and useful. We believe such separation of aspect weaving concern helps in better reuse of aspect definitions.

## 6. ACKNOWLEDGEMENTS

We would like to thank the CAMEO team: Aravind, Gopichand, Jagadish, and Lalitha. Thanks to our colleagues Mohan and Veena for their support at various stages. We appreciate the AspectJ mailing list members for their critical analysis of selective aspect weaving and for comparing it with the strategies employed in AspectJ and Hyper/J.

## 7. REFERENCES

[1] Microsoft .NET architecture and resources – www.microsoft.com/net

[2] SSCLI – http://www.microsoft.com/licensing/sharedsource/default.asp

[3] ECMA C# specification - http://www.ecma.ch/ecma1/STAND/ECMA-334.htm

[4] AspectJ download, documentation – http://www.eclipse.org/aspectj/

[5] Dharma Shukla, Simon Fell, and Chris Sells. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. MSDN magazine, March 2002 http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/AOP.asp

[6] Martin Fowler. How .NET's Custom Attributes Affect Design. IEEE Software September/October 2002. http://www.martinfowler.com/articles/netAttributes.pdf

[7] Nunit. http://www.nunit.org

[8] Java Aspect Components. http://jac.aopsys.com

[9] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Reflection 2001, pages 1-24, 2001. LNCS 2192.

[10] Javaassist. http://www.csg.is.titech.ac.jp/~chiba/javassist

# Idioms for Building Software Frameworks in AspectJ

Stefan Hanenberg[1] and Arno Schmidmeier[2]

[1]Institute for Computer Science
University of Essen, 45117 Essen, Germany
shanenbe@cs.uni-essen.de

[2]AspectSoft,
Lohweg 9, 91217 Herbruck, Germany
A@schmidmeier.org

**ABSTRACT**

Building applications using AspectJ means to design applications build upon the new language features offered in addition to Java. The usual argumentation that AspectJ permits a better separation of concerns in contrast to the traditional static typed object-oriented code might be valid, but does not prevent developers to misuse these language features. What's needed is a discussions of how to apply the language features of AspectJ to achieve good designed applications. In this paper we propose four idioms whose application turned out to result in good designed application in an appropriate context.

## 1. INTRODUCTION

AspectJ comes with a number of (more or less) new language features which try to tackle the problem of crosscutting code. Although the impact of these features on the object-oriented code is already analyzed like for example in [6] it is not clear how to apply these features to new problems. However, as noted by R. W. Floyd : "*To persuade me of the merit of your language, you must show me how to construct programs in it. I don't want to discourage the design of new languages; I want to encourage the language designer to become a serious student of the details of the design process*" [2, p. 460].

In this paper we continue to describe idioms which turned out to be used in good designed AspectJ applications and try in this way to encourage the usage of AspectJ in large scale applications. In [4] we already proposed some idioms which were closely related to the language features of AspectJ. Here we propose idioms which seem to be somehow more advanced. The idioms where successfully used in a large scale AspectJ project on Enterprise Application Integration (EAI) systems [9]. In [4] we already discussed the relationship between the proposed idioms and patterns. One of the major points in this discussion was, that the proposed idioms did not have the pattern format. However, in this paper we still neglect to put the idioms in such a format because of two reasons. First, we feel that it is still more important to discuss typical design decisions in aspect-oriented languages than to claim that a number of good patterns are found. And second, it is still not yet clearly determined what language features an aspect-oriented language will provide in the future: the provided language features still evolve from version to version. Hence, a collection of good design decisions might be no longer valid in the future because of language changes in AspectJ.

Furthermore, we do not give an example for each proposed idiom. Instead, we discuss the design of an existing real-world example which was influenced by the here proposed idioms at the end of the paper.

In section 2 to 5 we discuss four different idioms. We concentrate in this discussion on the core ingredients of those idioms, their advantage and their consequences. In section 6 we discuss a real-world example where such idioms were used. Since the example comes from quite a large context we just can give a small glimpse of it and we only concentrate on the considerations which lead to the final design. In section 7 we conclude the paper.

## 2. TEMPLATE ADVICE[1]

A template advice is used whenever additional behavior which should be executed at a certain join point contains some variabilities. That means the code to be executed consists of a fixed and a variable part whereby the variable part changes from application to application. A template advice corresponds directly to the *template method design pattern* [3] whereby the template is specified inside an advice instead of a method.

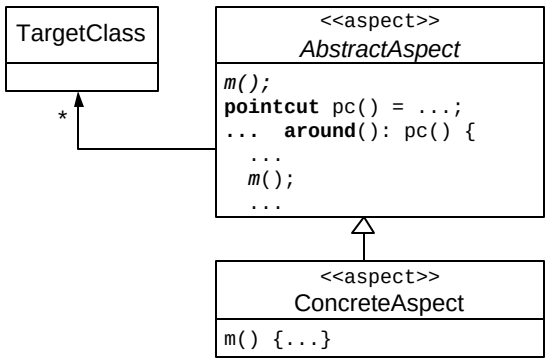



**Figure 1. Template Advice**

Whenever the problem is given that the code to be executed at certain join points is partly known and fixed, but might contain (depending on the concrete join point) some variabilities, the designer has to decide how to consider such variabilities. In case

[1] It should be noted that the name *template advice* has been already used in [4] for a different idiom. We regard the name of the idiom in [4] a little bit misleading and renamed it to *import pointcut* since this metaphor seems to describe its usage more appropriate.

the join points are well-known, it is possible to implement a number of different aspects for each of those different kinds of join points. The disadvantage of this approach is, that all those different join points usually have some commonalities. This commonalities (which means, that some common pointcut specifications are used) depict redundant code inside the application. The problem can be reduced by using the *composite pointcut idiom* [4]. However, the problem is still, that the advice contains redundant code. This redundant code depicts those part which are constant in each occurrence of the advice. To reduce this redundancy the stable part of the advice is used as a template and the variable part is put into a method. The property of AspectJ that advice cannot be refined in subaspects means in such a situation that the decision which part of the code is fix is final: it is not possible to refine advice incrementally. Instead, the aspect needs to be refactored.

The ingredients of the template advice are (in correspondence to *template method* [3]):

- *Abstract aspect*: the aspect that contains declarations of a number of primitive operations which are defined in sub-aspects. Usually, these methods are abstract, that means they are just declared but not defined. Furthermore, the abstract aspect contains the advice (which is called the template advice) which contains the invocations of the primitive operations. Usually the pointcut referred by the template advice is abstract.
- *Concrete aspect*: The concrete aspect defines or overrides the primitive operations of the abstract aspect and implements in that way the aspect-oriented adaptation of the target classes.

It seems questionable if the template advice is really a specific AspectJ idiom since it is very similar to the *template method*. However, we regards it as an specific idiom, because the consequences of using a template advice are quite different than the usage of a template method. First, in AspectJ only abstract aspects can be extended by further aspects. That means, when the corresponding aspect is written it must be clear whether or not a contained advice inside the aspect tends to be a template advice or not. Using pure object-oriented language features in a language supporting late binding this question does not have to be answered. For example in Java or Smalltalk almost every method can be overridden by a subclass. That means every method inside the superclass which contains invocations of an overridden method depicts a potential template method. That means methods might become template methods because of an incremental modification of the class structure. Hence, the preplanning problem of design patterns as mentioned in [1] is not that significant for a *template method*. On the other hand, because of the limited possibilities of incremental aspect refinement in AspectJ this problem is more present in a template advice. Hence, the consequences of using a template advice are much more restrictive.

The consequences of using a template advice are:

- *Separation of fixed and variable part of crosscutting code*: the advice depicts the fixed part of the crosscutting code, while the abstract method depicts the variable part which can be refined according to the special need.
- *Limited incremental refinement*: since AspectJ does not permit to refine advice directly (via overriding) the advice implementation is usually fixed.
- *Conflict handling*: if there are more than one concrete aspect which refer to at least one common join point the developer need to determine which advice should be executed. This can be either realized by further idioms, or by an explicit usage of dominate relationships between aspects.
- *Limited knowledge on aspects internals required*: the adaptation of the aspect behavior just depends on the concrete method definition. Hence, the developer performing the aspect adaptation only needs little knowledge about the concrete pointcut or the advice internals. However, a detailed description of the contract belonging to the abstract method is needed.
- *Lost access to introspective facilities*: since the reflective facilities of AspectJ are just available inside an advice there is no possibility to refer inside the method to the execution context. This must be considered during the design. In case the execution context might be needed, it has to be passed as a parameter.

Template advice usually occur together with *composite pointcuts* [5] where the concrete aspect defines the component pointcuts. Also, template advice are often used in conjunction with *pointcut methods* and *chained advice* (see section 3 and 4) where in both cases the concrete aspect refines the pointcut definition. Hence, different implementation of template advice usually differ in their handling of the corresponding pointcut.

It should be noted that template advice is a very generic idiom which builds in conjunction with template method and composite pointcut the fundament of aspect-oriented frameworks in AspectJ. It can be compared to [7] whose analysis of software frameworks is based on the distinction between hook and template coming mainly from the template method design pattern.

## 3. POINTCUT METHOD

A pointcut method is used, whenever a certain advice is needed whose execution depends on runtime specific elements which cannot or only with large effort expressed by the underlying pointcut language.

The pointcut language of AspectJ is quite expressive. Dynamic pointcuts like `args(..)` permit to specify join points which are evaluated during runtime and permit in that way to specify a large variety of crosscuttings. Typical examples where dynamic pointcuts are used are the simulating dynamic dispatching on top of Java (cf. e.g. [9]). However, sometimes the decision of whether or not a corresponding advice should be executed is not that easy to specify inside a pointcut definition. Such a situation is usually

given if the advice execution depends on a more complex computation or includes a invocation history of the participating objects.

The usage of `if` pointcuts can reduce this problem. However, `if` pointcuts are somehow ugly since they permit only to call static members of the aspect. Furthermore, the usage of `if` pointcuts usually reduces the reusability of the enclosing aspect, because they are usually very specific to a small set of join points. Usually, the usage of the pointcut language seems to be inappropriate when the decision whether or not a corresponding advice should be executed can be better expressed by methods than the pointcut language. In these cases the usage of a pointcut method is appropriate.

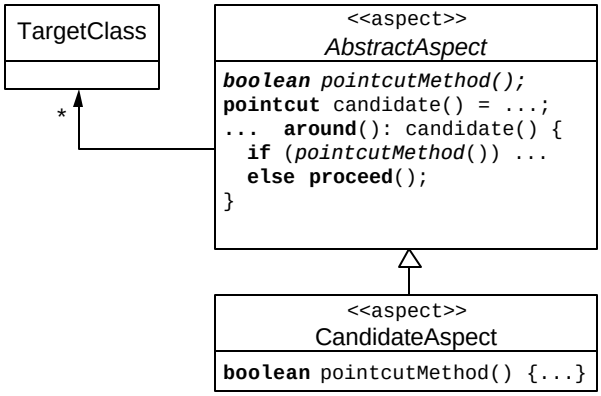



**Figure 2. Pointcut Method**

The ingredients of a pointcut method are:

- *Candidate pointcut*: the pointcut which determines all potential join points where additional behavior might take place. However, the pointcut definition includes more join points than needed to perform the aspect specific behavior.
- *Pointcut Method*: the method which is invoked from inside the advice to determine whether or not the advice should be executed. Typically the return type of a pointcut method is boolean.
- *Conditional Advice*: the advice which contains the behavior which might be executed at the specified join points. The additional behavior is conditional executed depending on the result of the pointcut method.
- *Candidate Aspect:* the aspect which refines the pointcut method.

Implementations of pointcut methods vary in a number of ways. First, usually a pointcut method's return type is boolean. That means a pointcut method only determines whether or not the additional behavior specified inside an advice should be executed. On the other hand, a pointcut method can also include just any computation whereby the conditional execution of the advice depends on the pointcut methods result (and any other context information). That means the decision whether or not the advice should be executed not only depends on the pointcut method itself.

Another important issue is how the computation of the pointcut method depends on the execution context of the application. Usually context information are directly passed by the advice to the pointcut method. That means the referring pointcut either passes some parameters to the advice or the advice extracts context information using the introspection capabilities of AspectJ like `thisJoinPoint` or `thisStaticJoinPoint`. Another possibility is, that the aspect itself has a state that is set by the application's execution context. The pointcut method can decide because of this state whether the advice should be executed or not.

An advantage of using a pointcut method is its adaptability by aspects: it is possible to specify further advice which refine the pointcut method outside the aspect hierarchy. That means, the condition whether or not an advice should be executed can be modified incrementally. In case the pointcut is hard-coded by using the pointcut language such an extension is not that easy. It assumes a corresponding underlying architecture or *rules of thumb* like discussed in [5].

The consequences of using a pointcut method are:

- *Hidden pointcut definition*: the user which specifies the pointcut method does not need to understand the implementation of the whole pointcut. He just needs an acknowledgement that at least all join points he is interested in are specified by the pointcut.
- *Parameter passing*: to determine whether or not the advice should be executed, the pointcut method needs some inputs. This might be for example property files, or (which is more usual) parameters which are passed from the pointcut to the advice and then from the advice to the pointcut.
- *Possible late pointcut refinement*: the pointcut method can be refined by further aspects.
- *Default advice behavior*: in case the conditional advice is an after or around advice, it is necessary to specify any default behavior. Around advice usually call `proceed`, while after advice usually pass the incoming return value.
- *Little knowledge about advice internals needed*: when specifying the pointcuts it is not necessary to understand all internals of the advice. Usually it is enough to have a description in natural language what kinds of join points can be handled by the advice and what kind of impact the advice has on the join point.

The pointcut method idiom is similar to the *composite pointcut* [5]. Both divide the pointcut into a stable and variable part (usually a composite pointcut it used in conjunction with a inheritance relationship between aspects). The difference between both is, that for adapting a composite pointcut the application of an inheritance relationship between aspects is necessary. This also implies that a composite pointcut has some preplanned variabilities (which are usually component pointcuts). A pointcut method does not directly depend on an inheritance relationship. The refinement might be either achieved via inheritance or by an advice. In the first case a pointcut method plays the role of an

abstract method inside a template advice. In the latter case, a pointcut method is often refined by a chained advice.

## 4. CHAINED ADVICE

Whenever there is (extrinsic) behavior of objects which is regarded to be somehow fragile what means it seems as if these methods might change because of a number of different decisions and furthermore by a number of different aspects the usage of chained advice is recommended.

Object-orientation already offers to extend the behavior of objects via the inheritance mechanism. Often this extension is based on a *template method* [3] where the pattern's abstract method already contains a concrete implementation. However this does not really solve the adaptation problem: the adaptation is achieved by inheritance and that implies a new class has to be created which overrides and adapts a known one. Furthermore, it must be guaranteed that the request for creating new objects must be redirected to the new class in certain situations. If (for the original classes) no *creational patterns* [3] where used such a task tends to be error-prone and the resulting design is usually unacceptable. In such cases, where an application's behavior at (at least) one join point depends on a number of concerns those concerns are usually not orthogonal, but interact in some way. That means, the new behavior should be modularized in separate aspects, but the relationship between such non-orthogonal concerns must be considered. In such cases we propose to apply the chained advice idiom.

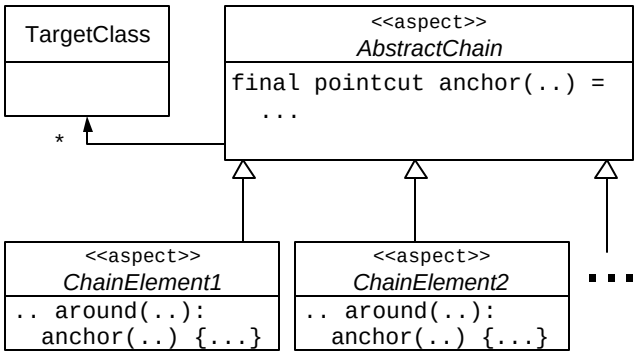



**Figure 3. Chained Advice**

The ingredients of a chained advice are:

*Abstract chain*: the aspect containing the anchor pointcut.

*Anchor pointcut*: the pointcut which is used by every advice within the chain. We call this the anchor pointcut, because each chain of advice is anchored at each join points part of this pointcut definition.

*Chain element*: The aspects extending the abstract chain and containing the advice which refers to the anchor pointcut. The chained advice have a predefined order. Usually each advice contains a mechanism to redirect the execution to a different advice.

In contrast to the previous mentioned idioms, a chained advice comes with a number of different implementations. On the one hand it is not necessary that the pointcut is inherited from a super-aspect. Instead, we found either the usage of static pointcuts, or even more complex aspect hierarchies than illustrated in figure 3. We found implementations where the chain was realized by an ordinary `proceed`-call, in other cases we found more complex pointcut definitions (that means each chain element offers a join point used by the following chain element). Also, in many cases the execution of chained advice is mutually exclusive, than means at most one chained advice is executed. But there are situation where more than one chain element is executed. What kinds of chained advice should be used depends on the concrete situation.

The way how the mutually exclusive advice were realized differ in different applications. On the hand (as we will illustrate in the final example) *pointcut methods* were used, in other cases ordinary advice in combination with *composite pointcuts* [5] were used. Both implementations have their pro and cons. The advantage of the first approach is that aspects do not need to have any knowledge about each other, i.e. their implementations do not depend on each other. But this also means that the advice execution order has to be controlled in some way. The latter approach assumes an explicit dependency of each advice.

The consequences of using the chained advice idiom are:

- *Separate concerns for each advice*: each advice represents certain behavior coming from different concerns within its own module.
- *Independent composability*: certain elements within the chain can be composed independent of each other. The level of independence of each chain elements depends on the underlying implementation. The major benefit is usually, that new chain elements can be added without the need to perform any destructive modifications within existing chain elements.
- *Parameter passing*: a mechanisms is needed to pass the responsibility from one chain element to another.
- *Default behavior needed*: often chained advice need to provide a default behavior at the anchor join points.

Chained advice make often use of *pointcut methods* to determine whether or not a chain element should be executed. Furthermore, chained advice often make use of *composite pointcuts* to reduce redundant pointcut definitions.

## 5. FACTORY ADVICE

Whenever the object creation of certain object depends on specific aspects which might vary from application to application or the execution context of an application, the usage of a *factory advice* is recommended.

A factory advice is an advice which is responsible for the object creation. It looks similar to the well-known design pattern *factory method* [3]. The argumentation why we still regard this a specific idiom in AspectJ is similar to the argumentation in section 2: the consequences of using a factory advice differ widely from the factory method.

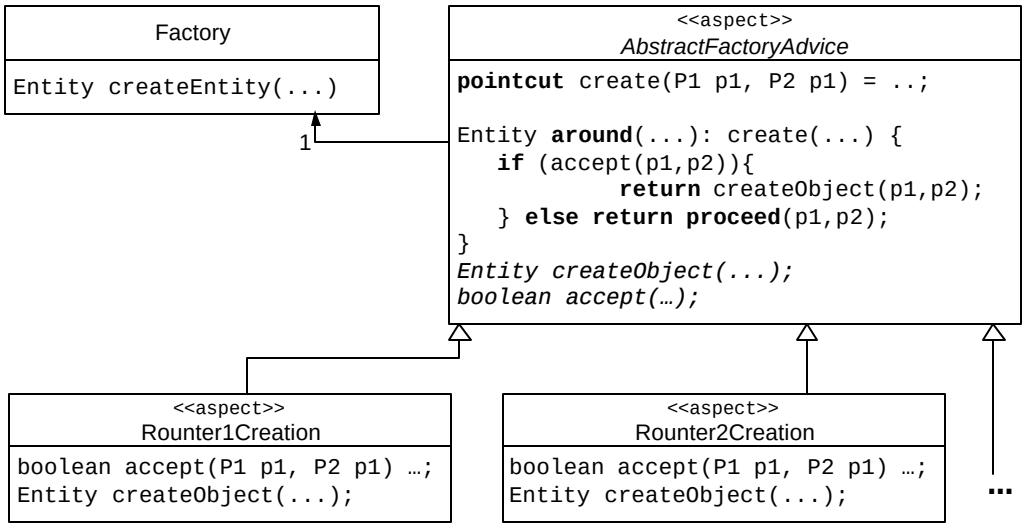



**Figure 5. Example: Object creation in large scale frameworks**

Whenever the creation of objects depends on certain aspects (and there might be more than one aspect) and such object creation might differ in different applications or different execution contexts it is usually not appropriate only to intercept the object creation using a pointcut to the constructor and then redirecting the creation using an around advice. The problem in such a context is usually the restriction that around advice need to return the same type than its join points.

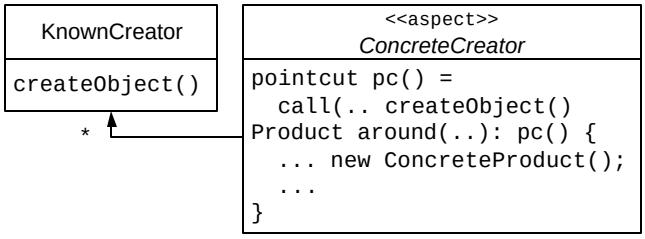



**Figure 4. Factory Advice**

The ingredients of a factory advice are:

- *default create method*: the method which is invoked by a client to request a new object. Usually, the method just return a null object.
- *a concrete creator*: the aspect which contains a pointcut to the default create method and the specification what product should be created.
- *abstract product*: the product expected by the client. Usually the factory advice redirects the creation of the abstract product to a different class extending the abstract product.

The relationship between a factory advice and the usual application of advice can be seen like the relationship between the *factory method design pattern* [3] and the *template method* [3]: although both are similar in their relationship of hook and template their differ mainly in the way their intention.

The consequences of using a factory advice are:

- *deferred object instantiation*: the aspect instantiation is no longer hard coded inside the object structure, but moved to the aspect definition. That means the instantiating aspect must be woven to the application to guarantee its correctness.
- *specified default behavior*: an advice factory assumes a specification of a default behavior of the default create method. Usually, the advice overrides the whole behavior specified there. But there are situations where the default create method contains some meaningful code and the aspect code is just executed in "special situations".
- *composability*: The advice factory permits to exchange the object creation process without performing destructive modifications within the object structure.

Factory are often used as chained advice in cases where the object to be instantiated depends on some execution context. In this way the factory advice looks even more like the abstract factory design pattern [4].

## 6. Example

Object oriented component frameworks suffer always from the problem of the construction of new component instances. The creational patterns in [4] reduce but do not really solve the problem. Each combination of these patterns violates at least in one point the principle of "need to know", which leads to somehow non transparent dependencies. Each component can know everything from the framework but not the other way round. When the framework is responsible to constructing new component instances, the framework needs to know the component. Delegating this kind of knowledge to framework configuration files doesn't solve that problem either. This approach contains several other drawbacks: it is impossible to implement the component in plain Java, a combination of Java and XML is needed, several checks which modern compiler can perform during compile time are no longer possible, code patterns

which enforce all configurations are not possible, this approach is not valid for high performance applications, because of the additional overhead caused by the required use of the Java Reflection API.

It is desirable, that every component connects itself to the construction mechanism. We present a solution of this problem as an example of a combination of the discussed idioms, which has been applied in the EOS-product family [10].

The core functionality is that dependent of the passed parameters the component decides on its own if it should be instantiated or not. That means, it depends on the framework configuration what objects have to be instantiated an in such a situation the application of an advice factory is appropriate. That means the request of an object creation is passed to a certain *default creation method* (we neglect here the implementation of corresponding pointcut create). However, the decision of what concrete product should be created depends on the one hand on the passed parameters and on the other hand on the available components inside the framework.

Since it is possible to specify all join points and it depends on the installed components whether or not they should be instantiated we decided use a *poincut method* inside the *advice factory* as illustrated in figure 5. Clients request a new abstract product (of type `Entity`) from the factory (which is in the concrete example an object). The factory object's default create method contains a dummy implementation. The concrete creator defines a pointcut for this method and defines a template advice and a pointcut method. The pointcut method `accept` specifies whether or not a concrete aspect should be responsible for the object creation or not. The abstract method `createObject` is overridden by concrete aspects and creates a concrete product.

In the here mentioned context we realized the concrete aspects as *chained advice* where each installed component comes with its own chain element for object creation. The reason for it is, that the fixed part of the template advice can be easily implemented as a chained advice and the responsibility which chain element creates the object lies in each chain element's `accept` method. Since the template advice either invokes `createObject` or proceeds with the join points execution all chain elements are mutually exclusive. Under the assumption that each element's pointcut method `accept` is adequately implemented there is no need determine any domination of the aspects.

Since the creational process differed widely from entity to entity we decided the implement the object creation as the abstract method inside the template method idiom.

## 7. CONCLUSION

In this paper we demonstrated a small collection of idioms we found frequently inside AspectJ applications and demonstrated an example which illustrated the usage of the idioms. The intention of the paper is to demonstrate "good design decisions" in AspectJ and discuss their advantages and disadvantages.

Although we found implementations of the here described idioms in current AspectJ projects we are aware of the fact, that there is no such clear distinction between the here described idioms and the known GoF design patterns *template method*, *chain of responsibility*, *abstract factory* and *factory method*. In that way it looks like the here proposed idioms are more a implementation of known design patterns as e.g. proposed in [6]. On the other hand the consequences of each of the idioms is quite different from the consequences of using the GoF patterns. Such consequences are mainly determined by the restriction that concrete aspects cannot be extended and advice cannot be overridden.

Nevertheless, the *pointcut method* seems to be an idiom which is highly related to an aspect-oriented language features and seems in that way rather a "pure aspect-oriented idiom" than the others. However, we think that the here described idioms are good examples of good AspectJ design which were successfully used and should be therefore considered when designing AspectJ applications if the application's context matches the idioms contexts.

## 8. REFERENCES


[1] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods Tools and Applications*, Addison-Wesley, 2000

[2] Floyd, R. W.; *The Paradigms of Programming*, Communications of the ACM, Volume 22, No. 8 (1979), pp. 455 – 460.

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

[4] Hanenberg, S.; Costanza, P.: *Connecting Aspects in AspectJ: Strategies vs. Patterns*, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD'01, Enschede, April, 2002

[5] Hanenberg, S., Unland, R.: *Using and Reusing Aspects in AspectJ*. Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA, 2001

[6] Hannemann, J., Kiczales, G., *Design Pattern Implementations in Java and AspectJ*, OOPSLA 2002.

[7] Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, 1995.

[8] Schmidmeier, A.; Hanenberg, S.; Unland, R.: *Implementing Known Concepts in AspectJ*, 3rd Workshop on Aspect-Oriented Software Development of the German Informatics Association, March, 2003

[9] Sirius GmbH, Enterprise Object System, *EOS, Functional Product Overview*, EOS Core System Version 3.5, 2002