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

March 14, 2005

Held in conjunction with the Fourth International Conference on
Aspect-Oriented Software Development (AOSD 2005)

Chicago, USA

College of Computer and Information Science
Northeastern University
Boston, Massachusetts 02115
360 Huntington Avenue, 161CN
NU-CCIS-05-03

Table of Contents

Using a Low-Level Virtual Machine to Improve Dynamic Aspect Support in Operating System Kernels

Michael Engel
Dept. of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Str.
D-35032 Marburg, Germany
engel@informatik.uni-marburg.de

Bernd Freisleben
Dept. of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Str.
D-35032 Marburg, Germany
freisleb@informatik.uni-marburg.de

ABSTRACT

Current implementations of software providing dynamic aspect functionality in operating system (OS) kernels are quite restricted in the possible joinpoint types for native code they are able to support. Most of the projects implementing advice for native code use basic technologies adopted from instrumentation methods which allow to provide *before*, *after* and *around* joinpoints for functions. More elaborate joinpoints, however, are not available since support for monitoring native code execution in current CPUs is very restricted without extensive extensions of the compiler toolchain. To realize improved ways of aspect activation in OS kernels, we present an architecture that provides an efficient low-level virtual machine running on top of a microkernel system in cooperation with an aspect deployment service to provide novel ways of aspect activation in kernel environments.

1. INTRODUCTION

Current implementations providing dynamic aspect functionality today typically are based on either modifications of a high-level virtual machine (VM) like the JVM or modifications of the instruction stream that is executed on demand on the machine code level. Systems based on high-level VMs, like e.g. Steamloom [2], provide a rich set of functionality and are able to supply rather complex joinpoint models since they have the ability to intercept the execution of virtual machine instructions. Due to general characteristics of these VMs – no pointers are available in Java, languages running on top of the VM (Java, C#) are not widely used as system implementation languages – providing aspect support for an operating system kernel running on top of the VM is not feasible[1]. In contrast, aspect activation directly on the machine instruction level, provided by systems like TOSKANA [10] and Arachne [17], is restricted to the amount of interaction that is possible by dynamically rewriting the instruction stream using the available symbolic information in the executable code.

In this paper, `TOSKANA-VM`, a novel way to provide aspect support for legacy operating systems is presented. Based on experiences with the NetBSD-based implementation of TOSKANA using native code manipulation [10], an intermediate approach between direct control of the native instruction execution and employing a high-level virtual machine is chosen. Since the abstraction level of a virtual machine is required in order to gain improved control over the execution of instructions, an existing low-level machine, LLVM [14], was chosen as the basis for TOSKANA-VM. In addition to the virtual machine itself, LLVM consists of a complete compiler toolchain providing support for compiling C and C++ programs that use GNU extensions. This is the basis for running a kernel on top of LLVM with only a moderate amount of modifications.

LLVM, however, is not well suited to run as a virtualization layer on top of the bare hardware. Thus, TOSKANA-VM uses the L4 microkernel as basis for executing LLVM instances. Here, L4 provides only minimal kernel functionality like memory management, task and thread abstractions and a fast implementation of inter-process communication. On top of L4, a mix of LLVM instances with their associated programs in LLVM bytecodes and native code can run concurrently. One of the native programs running on top of L4 is the weaver, which is responsible for activating and deactivating joinpoint shadows in the particular LLVM instance using IPC notifications.

This paper is organized as follows. Section 2 describes the overall structure of a system based on the L4 microkernel and an operating system personality, followed by an overview of low-level virtual machines in section 3. Section 4 describes ways to implement aspect-oriented functionality in a virtual machine. An overview of the L4- and LLVM-based system structure is contained in section 5. Section 6 summarizes related work. Section 7 concludes the paper and outlines areas for further research.

2. MICROKERNEL-BASED SYSTEMS

Compared to traditional monolithic operating system kernels, a microkernel-based system divides the functionality it provides into several system components that are cleanly separated from each other. At the base of the system, the

[1]There are, however, operating systems written in Java. These have never become mainstream, though.

.

microkernel itself provides only the absolute minimum functionality of a kernel – in the case of L4 used in this paper, this is restricted to memory management, task management and interprocess communication primitives. All other functionality usually contained in a monolithic kernel is delegated to so-called *kernel personalities*, which are essentially user-mode tasks running as microkernel applications.

2.1 L4 System Structure

The basis for all interaction with the hardware in a L4-based system [15] is the microkernel itself. L4 has complete control over the hardware and is the only process in the system running in privileged ("kernel" or supervisor) CPU mode. All other parts of the system run under control of the microkernel in non-privileged user mode. This includes all operating system personalities described in the next paragraph.

L4 is optimized for fast inter-process communication, which is extensively used throughout the system. IPC messages can be sent and received by any task in the system; messaging in L4 is synchronous, so the delivery of IPC information is guaranteed by L4 as soon as the IPC call returns to the caller. In addition, L4 supports a method to share address spaces between different tasks running on top of L4. Using the *flexpages* system, one task can share parts of its virtual address space with another task with page-sized granularity.

In order to support virtualization of OS instances, L4 provides abstractions for timers and interrupts as well as virtual memory management and encapsulates these as IPC messages.

2.2 OS Instances

Since L4 only provides basic kernel functionality, an additional layer of software is required to implement the features required by application programs that L4 is lacking. This is realized in the form of an *operating system personality* that provides the standard interfaces of a traditional monolithic OS kernel to applications running on top of it. In the case of L4, a port of Linux as a personality is available, called *L4Linux* [11]. L4Linux is a modified version of Linux 2.6 in which all critical hardware accesses (interrupt control, page table handling, timer control) are removed and replaced by IPC calls to the underlying L4 microkernel that performs these operations on behalf of the Linux personality (if permitted by the security guidelines).

L4Linux runs as an ordinary user mode process, thus several L4Linux instances are unable to interfere with each other. As a consequence, a virtualization on the level of L4Linux instances running in parallel is feasible and already used in several applications [19, 9].

3. THE LOW-LEVEL VIRTUAL MACHINE

LLVM is a virtual machine infrastructure consisting of a RISC-like virtual instruction set, a compilation strategy designed to enable effective program optimization during the lifetime of a program, a compiler infrastructure that provides C and C++ compilers based on the GNU compiler collection and a just-in-time compiler for several CPU architectures. LLVM does not implement things that one would expect from a high-level virtual machine. It does not require garbage collection or run-time code generation. Optional LLVM components can be used to build high-level virtual machines and other systems that need these services.

3.1 LLVM Instruction Set

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler (providing the intermediate representation IR), as an on-disk bytecode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. All three different forms of LLVM code representation are equivalent.

The LLVM representation aims to be as light-weight and low-level as possible while being expressive, typed, and extensible at the same time. It aims to be a "universal IR", by being at a low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IRs", allowing many source languages to be mapped to them). By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function.

3.2 The LLVM Infrastructure

As fig. 1 illustrates, the LLVM compilation strategy exactly matches the standard compile-link-execute model of program development, with the addition of a runtime and offline optimizer. Unlike a traditional compiler, however, the .o files generated by an LLVM static compiler do not contain any machine code, but rather LLVM code in a compressed format.

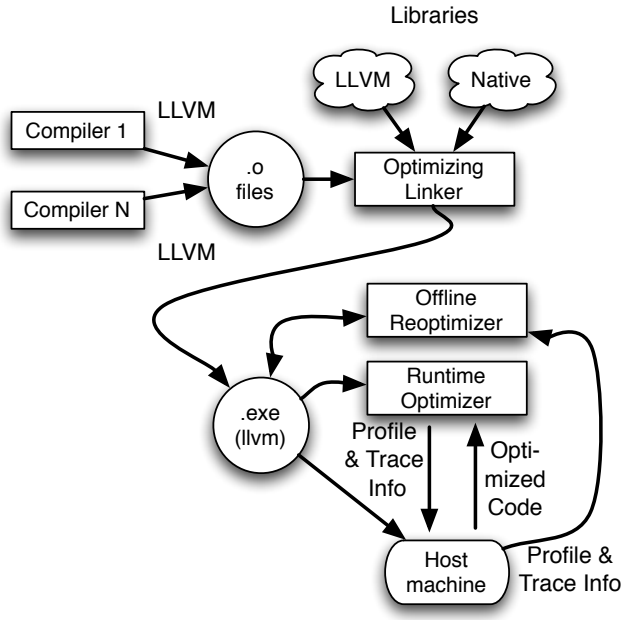

Figure 1: LLVM Compilation Infrastructure

The LLVM optimizing linker combines LLVM object files, applies interprocedural optimizations, generates native code, and links in libraries provided in native code form.

Once the executable has been produced, the developers (or end-users) of the application begin executing the pro-

gram. Tracing and profiling information generated by its execution can be optionally used by the runtime optimizer to transparently tune the generated executable.

The system heavily depends on having a code representation with the following qualities:

The bytecode is high-level and expressive enough to permit high-level analyses and transformations at linktime and in the offline optimizer when profiling information is available. Without the ability to perform high-level transformations, the representation does not provide any advantages over optimizing machine code directly.

Also, the code has a dense representation, to avoid inflating native executables. Additionally, it is useful if the representation allows for random access to portions of the application code, allowing the runtime optimizer to avoid reading the entire application code into memory to do local transformations.

Finally, the code is low-level enough to perform lightweight transformations at runtime without too much overhead. If the code is sufficiently low-level, runtime code generation has low overhead in a broad variety of situations. A low-level representation is also useful because it allows many traditional optimizations to be implemented without difficulty.

4. ASPECT ACTIVATION IN A VM

When handling aspect activation in native code, the interception of the execution of arbitrary machine instructions is usually not supported extensively by standard CPUs. Consequently, the joinpoint model is quite restricted and dynamically inserting joinpoint shadows is an intricate task, since self-modifying code is used in the process.

Using a virtual machine instead of executing code directly on the CPU provides improved methods of detecting possible joinpoints, since all instructions are now either directly interpreted by the VM or translated into short native code segments by the just-in-time (JIT) compiler.

The following subsection describes the restricted "traditional" approach using code splicing, followed by a description of the advantages of the instruction-manipulating VM-based approach.

4.1 Code Splicing in Native Code

When directly working with native code, like in the TOSKANA project on NetBSD, the basic method for inserting dynamic joinpoint shadows into native code is *code splicing*. Code splicing is a technology that replaces the bit patterns of instructions in native code with a branch to a location outside of the predefined code flow, where additional instructions followed by the originally replaced instruction and a jump back to the instruction after the splicing location are inserted.

TOSKANA uses fine-grained code splicing, which is able to insert instrumentation code with the granularity of a single machine instruction. As shown in fig. 2, splicing replaces one or more machine instructions at a joinpoint with a jump instruction to the advice code with the effect that the advice code is executed before the replaced instruction.

This method has some significant drawbacks. When modifying native code, some complications show up that have to be taken care of in order to avoid corrupting activities currently running in kernel mode.

Since the execution of kernel functions may usually be interrupted at any time, it is desirable to make the splicing

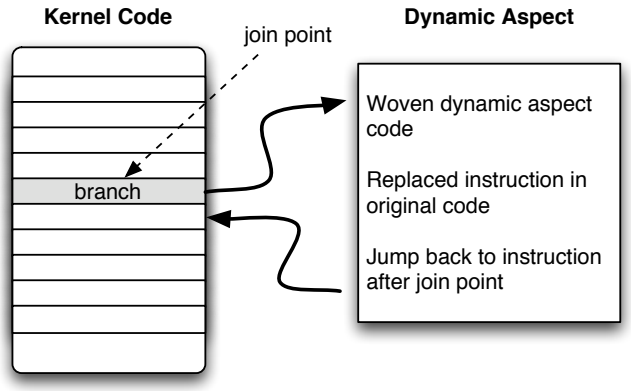

Figure 2: Code Splicing

operation atomic.

Another problem may be that more than one instruction has to be replaced by the jump instruction in the splicing process and the second of these replaced instructions is a branch target. A branch to that address would then end up trying to execute the bit pattern that is part of the target address as operation code with unpredictable results. In this case, either a workaround has to be activated – e.g. by rewriting an instruction before the joinpoint itself, thereby reducing the precision of the joinpoint location – or the replacement of an instruction must be avoided here, leading to a slightly restricted amount of available joinpoints.

The biggest problem with code splicing is that it only provides a very restricted joinpoint model. Since no information on register contents and values of pointers is available to the weaver, any operations involving dynamically calculated or loaded values are not eligible as possible joinpoint types.

Thus, essentially, only support for before, after, and around joinpoints is available by splicing in jumps to advice code at the beginning and return points of the respective function. A wider variety of joinpoint types is desirable, but can not be achieved using splicing.

4.2 Manipulation of Instruction Execution

Using LLVM gives the aspect weaver enhanced control over the execution of (VM bytecode) instructions. Instead of having to rely on self-modifying code at program runtime, the infrastructure executing the (byte-)code itself can now be instructed to intercept instruction execution, thereby providing much more detailed information about the current state of the machine.

Based on LLVM, TOSKANA-VM is able to supply a broader range of joinpoint types. The types currently implemented are described in the following paragraphs, accompanied with an explanation of the basic VM functionality executed to support them.

Execution and Call

LLVM provides two function call instructions, which abstract the calling conventions of the underlying machine, simplify program analysis, and provide support for exception handling. The simple call instruction takes a pointer to a function to call, as well as the arguments to pass (by value). Although all call instructions take a function pointer

to invoke (and are thus seemingly indirect calls), for direct calls, the argument is a global constant (the address of a function). This common case can easily be identified simply by checking if the pointer is a global constant. The second function call instruction provided by LLVM is the invoke instruction, which is used for languages with exception handling.

When using code splicing, no clean distinction can be made between *execution* and *call* joinpoints, as the only point at which the weaver can be certain that the function in question was actually executed is within the code of the function itself.

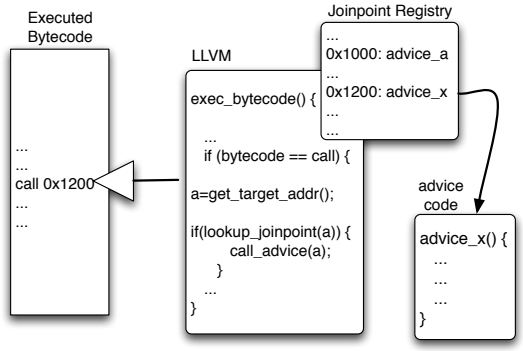

Figure 3: A *Call* Joinpoint

Implementing *call* joinpoints, as depicted in fig. 3, requires an interception of the call instruction described above. The target of the call instruction has to be compared against a list of active joinpoints and the related advice code is called appropriately.

Execution joinpoints – illustrated in fig. 4 – however, have a different semantics, since they exist at a point when the body of code for an actual method is executed. Thus, interception of the call is not sufficient; rather, the bytecode instruction pointer of the currently executing instruction has to be monitored. As soon as program execution enters (or leaves) the range of addresses defined by the function in question, advice code can be called. While monitoring the instruction pointer seems expensive at first look, the current implementation uses a fast hash-table mechanism to speed up the lookups. In future versions, annotated bytecodes could accelerate this functionality even further, requiring enhanced tool support.

Variable Assignment and Access

Capturing variable assignment and access was not possible using splicing, since accesses to variables in native code not only occur using a direct address reference (which could be detected), but more commonly using pointers to variables contained in registers or calculated target addresses that were not available to the splicing process.

In LLVM, however, the final address of every read or write instruction executed by the VM is well-known. Hence, interception of read accesses (i.e., variable read) and write accesses (i.e., variable assignment) can be intercepted and corresponding advice code can be executed as the address of the variable is well-known from the symbol table.

A problem with joinpoints triggering on variable accesses

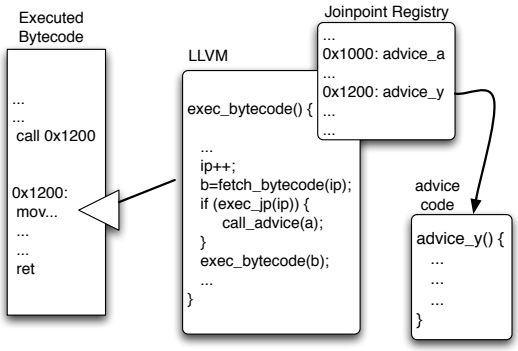

Figure 4: An *Execution* Joinpoint

lies in variables that are stored in registers for the sake of faster access times. Here, the VM provides a mapping from memory addresses to register contents; however, a tracking of variables in registers in the VM is necessary which is time-consuming. Thus, currently only *volatile* variables and variables to which an address operator (`&`) has been applied can be used as variable access joinpoints.

5. SYSTEM STRUCTURE: L4+LLVM

An overview of the system is given in figure 5. On top of the microkernel, the infrastructure consists of one or more instances of LLVM running in their own address spaces and the weaver as a separate process that handles communication with the VM instances.

Running on top of L4, one or more instances of LLVM execute, these in turn can run L4Linux instances compiled to bytecode or other L4-based applications compiled with LLVM as their target.

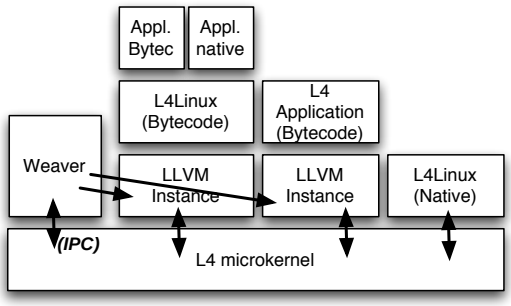

Figure 5: L4/LLVM-based TOSKANA-VM System

The dynamic aspect weaver is a separate component running on top of L4. It receives joinpoint description and weaving/unweaving requests from other tasks in the system and instructs a LLVM instance to add or remove the joinpoint description to respectively from its table of active joinpoints. Then, the weaver inserts the specified advice code in the address space of the LLVM instance in question using shared memory obtained via L4 flexpages.

In this L4-based system, tasks compiled to native code can execute in parallel to the LLVM instances. Of course, these tasks provide no support for dynamic aspects.

6. RELATED WORK

6.1 Steamloom

The Steamloom virtual machine [2] is an extension to an existing Java virtual machine. It follows the approach to tightly integrate support for AOP with the VM implementation itself instead of relying on a set of Java classes. Steamloom comprises changes to the underlying VM's object model and just-in-time compiler architecture. Its approach to dynamic weaving is to dynamically modify method bytecodes and to schedule those methods for JIT recompilation afterwards.

Steamloom was built to evaluate the benefits that can be gained by implementing AOP support in the execution layer instead of at application level. Various measurements [12] have shown that both flexibility and performance significantly benefit from such an integrative approach.

6.2 Arachne

Arachne [17] is a dynamic weaver for C programs. It uses the μDyner AOP infrastructure for writing and dynamically deploying aspects into running user mode C applications without disturbing their service. The implementation of joinpoints, however, requires source instrumentation of the program to reduce the cost of dynamic weaving. μDiner provides a special *hookable* source-level annotation with which the developer of the base program annotates points in the program at which dynamic adaptation is permitted. Here, only functions and global variables can be declared to be hookable. The aspect code is written using a special extension of C that provides the syntax for specification of joinpoints and advice types.

6.3 a-kernel

Research in AOP in OS kernels was initiated by [7], where problems crosscutting a common layered operating system structure were identified using FreeBSD as an example. Based on AspectC, an AOP extension of C, the a-kernel project tries to evaluate the usability of aspects to improve OS modularity and reduce the complexity and fragility associated with the implementation of an operating system.

In [5], [8], [6], and [4], various cross-cutting concerns are implemented as static aspects using AspectC. In addition, an analysis of code evolution implementing cross-cutting concerns between different versions of FreeBSD is undertaken and the evolution is remodelled using static aspects.

Further development resulted in the RADAR [18], a low-level infrastructure using dynamic aspects in OS code. Currently, no experience with implementing the system seems to exist.

6.4 Singularity

Singularity [13] is a new research operating system developed by Microsoft focussing on the construction of dependable systems through innovation in the areas of systems, languages, and tools. Based on executing operating system code using a VM (MSIL) running on top of a microkernel, this system shows similarities to the approach presented in this paper. However, support for AOP is not mentioned in the related publications – a combination of dynamic AOP approaches for .NET and Singularity might be an interesting alternative to TOSKANA-VM.

6.5 DTrace

DTrace [3] is a toolkit developed by Sun Microsystems to dynamically insert instrumentation code into an unmodified, running Solaris OS kernel. Unlike other solutions for dynamic instrumentation that execute native instrumentation code, DTrace implements a simple virtual machine in kernel space that interprets byte code generated by a compiler for the "D" language, which is an extension of C specifically developed for writing instrumentation code.

D provides safe instrumentation of the kernel. To avoid endless loops in instrumentation code, only forward branches are permitted by the VM. Thus, the functionality of D programs is relatively restricted. While this provides a lot of security when dynamically inserting code into random spots in the kernel, the execution model provided by DTrace is too restricted to implement general advice code.

6.6 Xen

Xen [1] is a virtual machine monitor for x86 that supports execution of multiple guest operating systems with high levels of performance and resource isolation using *paravirtualization* technology. On top of Xen, different operating systems are able to run concurrently.

Xen requires modifications to kernels running on top of it, but applications run unmodified. Due to the paravirtualization, which only virtualizes and intercepts certain privileged instructions, Xen is not capable of interception instruction flow at arbitrary points in the code, so deploying advice code is not possible using this virtualization approach.

6.7 VVM

Instead of designing and implementing a new virtual machine for each application domain, the goal of VVM is to virtualize the virtual machine itself. VVM supports so-called "VMlets" that contain a specification of a virtual machine implemented using VVM.

No large-scale operating system has been developed to run on top of either of these virtual machines, however, so the feasibility of this approach still has to be determined.

6.8 z/VM

Another low level virtual machine that is explicitly used to virtualize a single physical machine into distinct partitions each running its own operating system instance is IBM's z/VM[16], used on z-Series mainframe systems. z/VM supports a large number of operating systems running in parallel on top of the virtual machine[2] and would be an ideal target for implementing dynamic aspect support in the execution layer.

7. CONCLUSIONS

This paper presented a novel approach to providing enhanced aspect-oriented programming technology in the context of an operating system kernel. First results show that using a low-level virtual machine as a thin layer above the hardware while relegating basic system functionality to a microkernel directly executing in native code on the CPU provides a reasonable architecture to provide and experiment with joinpoint models that implement novel concepts

[2]A test has shown that 40,000 parallel Linux instances are possible

or operate on a more fine-grained level than the joinpoints available through code splicing.

The current implementation of an aspect-enhanced LLVM and the weaver task on top of L4 is capable of running test programs written in C compiled as LLVM bytecodes. The next step is the port of a complete kernel personality (e.g. L4Linux) to run in bytecode on top of L4 and LLVM. This should be feasible, since L4Linux already provides a separate architecture component for L4, which subsequently has to be augmented with the necessary low-level adaptations and compile infrastructure changes for a LLVM target.

The performance of a complex system like a kernel personality running in bytecode instead of native code will be an interesting focus of optimization. The currently available simple test cases give the impression that running a kernel in bytecode will be feasible performance-wise, though no exact data is available at this time.

Future work also includes the provision of an improved security model. The method currently available in L4 to restrict communication between processes – "clans and chiefs" – does not provide sufficient control over which processes are permitted to communicate. This model will be replaced by a new security infrastructure of an upcoming L4 release which will be the basis for future TOSKANA-VM releases.

8. REFERENCES

[1] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, and R. Neugebauer. Xen and the Art of Virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. "Virtual Machine Support for Dynamic Join Points". In *"Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD'04)"*, pages 83–92. ACM Press, 2004.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. "Dynamic Instrumentation of Production Systems". In *"Proceedings of the USENIX Annual Technical Conference"*, pages 15–28. USENIX, 2004.

[4] Y. Coady and G. Kiczales. "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code". In *"Proceedings of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)"*, pages 50–59. ACM Press, 2003.

[5] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. "Structuring System Aspects: Using AOP to Improve OS Structure Modularity". In *"Communications of the ACM, Volume 44, Issue 10"*, pages 79–82. ACM Press, 2001.

[6] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. "Exploring an Aspect-Oriented Approach to OS Code". In *"Proceedings of the 4th Workshop on Object-Orientation and Operating Systems at the 15th European Conference on Object-Oriented Programming (ECOOP-OOOSW) "*. Universidad de Oviedo, 2001.

[7] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. "Position Summary: Aspect-Oriented System Structure". In *"Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOTOS-VIII)"*, page 166. IEEE Press, 2001.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code". In *"Proceedings of of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)"*, pages 88–98. ACM Press, 2001.

[9] M. Engel and B. Freisleben. Wireless Ad-Hoc Network Emulation Using Microkernel-Based Virtual Linux Systems. *Proceedings of EuroSIM 2004*, pages 198–203, 2004.

[10] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. *"Proceedings of the Fourth Conference on Aspect-Oriented Software Development (AOSD'05)"*, 2005 (to appear).

[11] H. Härtig, M. Hohmuth, and J. Wolter. "Taming Linux". In *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*. IEEE Press, September 1998.

[12] M. Haupt and M. Mezini. "Micro-Measurements for Dynamic Aspect-Oriented Systems". In M. Weske and P. Liggesmeyer, editors, *Proceedings of Net.ObjectDays*, volume 3263 of *LNCS*, pages 81–96. Springer Press, 2004.

[13] G. C. Hunt and J. R. Larus. Singularity Design Motivation. *Microsoft Technical Report TR-2004-105*, 2004.

[14] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In *"Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California"*, pages 75–84. ACM Press, 2004.

[15] J. Liedtke. "μ-Kernels Must And Can Be Small". In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA*. IEEE Press, October 1996.

[16] R. A. Mullen. "z/VM: The Value of zSeries Virtualization Technology for Linux". In *"Proceedings of the IBM z/VM, VSE and Linux on zSeries Technical Conference, Miami, Florida"*. IBM, 2002.

[17] M. Segura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. "Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution". In *"Proceedings of the 2nd International Conference on Aspect-oriented Software Development"*, pages 110–119. ACM Press, 2003.

[18] O. Stampflee, C. Gibbs, and Y. Coady. "RADAR: Really Low-Level Aspects for Dynamic Analysis and Reasoning". In *"Proceedings of the ECOOP Workshop on Programming Languages and Operating Systems"*. ECOOP, 2004.

[19] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. "Towards Scalable Multiprocessor Virtual Machines". In *Proceedings of the 3rd Virtual Machine Research & Technology Symposium (VM'04)*. IEEE Press, May 2004.

Avoiding Incorrect and Unpredictable Behaviour with Attribute-based Crosscutting

Donal Lafferty
lafferty@engineer.com
Distributed Systems Group
Department of Computer Science
Trinity College Dublin

ABSTRACT

For consistency with component-oriented programming, the implementation of aspect-based software product families should be decoupled from the components that they influence. One solution is to implement such families with language-independent aspect-oriented programming (AOP) in combination with property-based crosscutting. Language-independent AOP decouples the implementation language of such a family from that of the components to which family members are applied. Using property-based crosscutting avoids coupling the software product family with a specific set of components, because join points are selected according to some implementation commonality such as a containing type name rather than precise details of their implementation. However, experimental evidence shows that compiling a component's implementation to a language independent format can introduce new join points that match a property-based crosscut. These matches can result in unexpected behaviour. To assure correct and predictable behaviour when a software product family member is applied to an application it is better to use attribute-based crosscutting, in which join points are selected according to the appearance of attributes, called annotations in Java, on the join point's implementation.

1. INTRODUCTION

The use of aspects to implement software product families has been demonstrated by work on spontaneous containers [13]. Here, different aspects are responsible for implementing persistence, transaction processing and access control properties of a container. Moreover, different implementations of each property can exist. These aspects can be assembled to create a container that enforces network policies applied to a particular network node. Such a product family might find practical application in tradeshow venues [13], where services vary according to the status of the attendee. The containers that allow visitor PDAs to interact with each other will differ from those provided to exhibitors. For instance, access control capabilities may be required by exhibitors that make use of a tradeshow registration database that is not available to visitors.

Language independent AOP and the use of property-based crosscutting should reconcile the implementation of aspect-based software product families with component-oriented programming. Software components emphasize deployment and composition characteristics that allow components provided by one organization to be combined with components of another by a third-party unrelated to either organization [14]. Specifically, "a third party is one that cannot be expected to have access to the construction details of all the components involved." [14]. Thus, implementation source code should not be a factor in the ability of an aspect-based software product family member to bind with components in an application. Language-independence addresses this requirement by allowing aspects and components to be written in a variety of languages and freely intermixed [9]. Using property-based crosscutting [6], an aspect selections join points in application according to some commonality such as a shared containing type, some naming convention, or common parameter types in the case of methods. Sufficiently general property-based crosscutting allows aspects to be used with a variety of components without the need to customize the aspect on a component by component basis. Using language independent AOP and property-based crosscutting should allow third parties to treat aspect-based software product families and application components being composed as black boxes.

The vehicle for testing this theory was Weave.NET [7], a language independent aspect weaver that supports a pointcut-advice mechanism [10] and allows clear-box crosscutting. Weave.NET targets Microsoft's Common Language Infrastructure (CLI) [3]. The CLI simplifies the task of implementing Weave.NET by providing a language-independent substrate to which components, regardless of implementation language, are compiled. The CLI standardizes the metadata descriptions of types and type members implemented by CLI components, regardless of implementation language, and component behaviour is written in a language-neutral binary format. With respect to the clear-box / black-box distinction [4], a black-box technique manipulates components in terms of their public interfaces, while a clear-box technique manipulates the parsed language structures used to write these interfaces. Clear-box techniques, such as those available with the pointcut-advice mechanism AspectJ [15], often offer a richer set of join points, because they provide a better representation of all the structures of a programming language used to write the component. In the case of components, language constructs that are expressed directly in byte code, such as accesses to type members, can be modified [4].

In our experiments, initially reported in [7], we applied members of a simple diagnostics software product family to a Fibonacci series enumerator, and we noted an inability to make strong assurances of correct and predictable behaviour of the resulting application. In our experiments, implementations of a recursive Fibonacci series enumerator algorithm written in SML.NET [5], VisualBasic.NET [12] and C# [11] were composed with logging and profiling functionality implemented using aspects. Using language independent AOP and property-based crosscutting, the same logging and profiling aspects could be applied to each Fibonacci enumerator component without change. However, we noted that in the case of the component implemented in

SML.NET it was not possible to predict the behaviour of the final application based on the source code of the Fibonacci algorithm. During the translation to a language neutral format, the SML.NET compiler introduced addition join points to the internal implementation of component interfaces that matched the logging aspect.

Rather than change to black box crosscutting, we avoided inadvertent join point matches using attributed-based crosscutting [7, 8]. The additional join points added during component compilation did not influence the component implementation, and so predictability could have been restored by limiting the weaver being used. However, it was possible to restore predictability with attribute-based crosscutting rather than by crippling the weaver. With *attribute-based crosscutting*, join point selection is written in terms of attributes. Attributes [3], also called annotations [1], offer a programming-language mechanism for associating additional information with the metadata descriptions of types and their members. Language support for attributes typically includes the ability to define new attributes and the ability to place attribute declarations along side the definition of types and their members. Annotation of a program element with an attribute causes additional data to become associated with the metadata description of that program element; however, annotating code with attributes does not influence program behaviour. For instance, an attribute might append the metadata of a method with the name of the programmer implementing the method. This information would be associated with the metadata description of the method. Rather than using the method name in a pointcut specification, the attribute name can be used instead. Since an attribute applied to source code appears only once in the compiled assembly, attribute-based crosscutting avoids inadvertent join point selection and thus results in more predictable behaviour.

In the remainder of this paper, we present the experimental results that identified problems with property-based crosscutting and that identified attribute-based crosscutting as a possible solution. In section 2, we summarize the composition scenario in terms of the software product family functionality being implemented and the components to which family members are being applied. In section 3, we present an implementation of the software product family based on property-based crosscutting solution and evaluate its drawbacks. In section 4, we do the same for an attribute-based crosscutting solution, and explain what drawbacks in the previous section are avoided. Finally, we summarize our findings in section 5.

2. EXPERIMENTAL SCENARIO

To evaluate the usefulness of property-based crosscutting in the context of language-independent AOP, we chose to use a software product family that provided application diagnostics. The software product family provides profiling and logging functionality features, and it is written using the aspect model provided by Weave.NET. One, the other, or both diagnostics features can be applied to methods of an application. However, in this paper we focus on the application of logging functionality to components implementing a common Fibonacci series algorithm. The algorithm enumerates series elements, and it was chosen based on the observation that Fibonacci algorithms are a common means of demonstrating AOP techniques [2].

2.1 Target Application

The specific algorithm targeted for our language-independence tests is a component implementing a recursive algorithm that enumerates members of the Fibonacci series. The algorithm, shown in Figure 2.1, includes two methods, one that generates elements in the Fibonacci series, and a second that reports a series of elements generated using the former method. Components containing these methods have been written in C#, VB.NET and SML.NET. The C# version shown in Figure 2.1 is typical of the algorithm, which is recursive regardless of the programming language used.

```
public class FibonacciSeries {
  public void FibSeries(int seriesLen){
    for (int i = 0; i<= seriesLen; i++) {
      long result = Fibonacci(i);
      System.Console.WriteLine("Element\t"+
                      i+ "\tvalue \t"+result);
    }
  }

  public long Fibonacci(int n){
    if (n > 1)
      return this.Fibonacci(n-1)
            + this.Fibonacci(n-2);
    return 1;
  }
}
```

Figure 2.1: C# source for algorithm to enumerate Fibonacci series elements.

2.2 Test Aspect

Our Fibonacci algorithm lacks an explicit indication of its complexity, but this is remedied by adding diagnostics functionality that provides logging. This logging is implemented via an aspect that reports the start and end of execution join points. Logging is a fairly simple concept made simpler by limiting the aspect to reporting the start and end of a method execution to the application console rather than logging to a file. A sample implementation of logging behaviour written in SML.NET is shown in Figure 2.2. The method names in the source allude to the kind of advice they implement. The appearance of multiple methods with the prefix `LogAfterJoinPointXXX` corresponds to the use of different return types in the methods of the Fibonacci series algorithm.

3. LOGGING VIA PROPERTY-BASED CROSSCUTTING

Property-based crosscutting is consistent with language independent AOP, and it allows an aspect to be created that can be applied to various components without the changes to the crosscutting specification However, the crosscutting specifications suffer the drawback of being error prone. Also, obtaining assurances of correct program behaviour is difficult, as program behaviour cannot be determined from inspection of component source code.

When the logging aspect is written using property-based crosscutting, the crosscutting specification is the same regardless of the language implementing the Fibonacci algorithm with which logging is woven. The logging aspect's XML-based crosscutting specification is shown in Figure 3.1 and it defines a named pointcut called `SomeMethodExecution` that identifies method invocations that take an integer as a parameter, regardless of the

return type. The crosscutting specifications are reusable without modification in that they can be applied to components without change. Reuse then relies on the component's types being the same, in terms of members and member signatures, regardless of implementing language.

```
 structure Aspect_ML_Logging =
 struct
 _classtype Logger() : TCD.CS.DSG.Weave.Reflect.Aspect()
  with
   LogBeforeJoinPointInt (param:int) =
   let
    val                    jptInfo                    =
valOf(this.##get_JoinPointStaticPart());
   in
    print "Join point: ";
    print (valOf(jptInfo.#toShortString())); print "\n";
    print "Execution parameter: ";
    print (Int.toString(param)); print "\n"
   end
  and
   LogAfterJoinPointLong(param:int, result:Int64.int)=
   let
    val                    jptInfo                    =
valOf(this.##get_JoinPointStaticPart());
   in
    print "Join point: ";
    print (valOf(jptInfo.#toShortString())); print "\n";
    print "Execution parameter: ";
    print (Int.toString(param)); print "\n";
    print "Execution result:    ";
    print (Int64.toString(result)); print "\n"
   end
  and
   LogAfterJoinPointVoid (param:int) =
   let
    val                    jptInfo                    =
valOf(this.##get_JoinPointStaticPart());
   in
    print "Join point: ";
    print (valOf(jptInfo.#toShortString())); print "\n";
    print "Execution parameter: " ;
    print (Int.toString(param)); print "\n";
    print "Execution result:    NONE!"; print "\n"
   end
  end
 end
```

Figure 2.2: Implementation of logging behaviour written in SML.NET

During weaving trials in which the logging aspect was applied to each implementation of the Fibonacci algorithm, we noted that the specification of types in XML was error prone, because mapping from language-based type names to CLI type names must be done manually. Writing a custom crosscutting specification in XML involves using metadata descriptions to select join points. To make Weave.NET aspects language independent, the crosscutting specifications are specified in terms of CLI types, and not the development language types with which a programmer will be familiar. The need to map from development language types to CLI types is acute in the case of primitive types, whose CLI names vary considerably from those used in the source code of a component. For example, Table 3.1 shows the mappings between SML.NET primitive types, their C# equivalent and their CLR name. These tables show no overlap between the programming language type names and those used by the CLI.

```
 <item>
   <named_pointcut>
     <modifier><public/></modifier>
     <name>SomeMethodExecution</name>
     <local_var_ref>
       <var_type>Int32</var_type>
       <var_name>data</var_name>
     </local_var_ref>
     <pointcut>
       <and>
         <pointcut><primitive>
           <execution>
             <method_signature>
               <return_type>
                 <type_name>*</type_name>
               </return_type>
               <join_point_type>
                 <type_name>*</type_name>
               </join_point_type>
               <method_name>*</method_name>
               <parameters>
                 <parameter>
                   <type_name>Int32</type_name>
                 </parameter>
               </parameters>
             </method_signature>
           </execution>
         </primitive></pointcut>
         <pointcut><primitive>
           <args>
             <parameter>
<formal_parameter_name>data</formal_parameter_nam
e>
             </parameter>
           </args>
         </primitive></pointcut>
       </and>
     </pointcut>
   </named_pointcut>
 </item>
```

Figure 3.1: A pointcut identifying method execution join points to which logging is applied.

Table 3.1: Mapping between CLI (.NET) types and C# / SML.NET equivalents, taken from [5].

.NET type	C♯ type	SML.NET type
System.Boolean	bool	bool
System.Byte	byte	Word8.word
System.Char	char	char
System.Double	double	real
System.Single	float	Real32.real
System.Int32	int	int
System.Int64	long	Int64.int
System.Int16	short	Int16.int
System.SByte	sbyte	Int8.int
System.String	string	string
System.UInt16	ushort	Word16.word
System.UInt32	uint	word
System.UInt64	ulong	Word64.word
System.Exception	System.Exception	exn
System.Object	object	object

```
structure App_Noninvasive_ML_Fibonacci
 : sig val main: string option array option->unit
    end =
struct
_classtype FibonacciSeries()
 with
  Fibonacci (n) =
    case(n) of
      0 => (Int64.fromInt(1))
    | 1 => (Int64.fromInt(1))
    | n => (this.#Fibonacci(n-1) +
            this.#Fibonacci(n-2))
 and
  FibSeries (n) =
    case(n) of
     ~1 => ()
    | n => (this.#FibSeries (n-1);
            print "Element\t";
            print (Int.toString (n));
            print "\t value \t";
        print(Int64.toString(this.#Fibonacci(n)));
            print "\n" )
 end
 fun SelfTest (elements, times) =
  let
         val fibML = FibonacciSeries()
  in
         case(times) of
           0 => ()
         | n => (fibML.#FibSeries(elements);
                 SelfTest(elements, times-1))
  end
 fun main  (a : string option array option) =
  let
    val elements = 10
    val times = 1
  in
    SelfTest(elements, times)
  end
end
```

Figure 3.2: SML.NET implementation of application to calculate Fibonacci Series elements.

We did experiment with making it easier to simplify type specification by allowing the use of truncated versions of CLI type names in which the namespace is removed. Hence, the use of `Int32` rather than `System.Int32` in the XML of Figure 3.1. While these truncated versions are shorter to write, they make mistakes easier to make. For example, in writing "`System.String`", we found the capitalization of `System` to be a reminder to capitalise the '`String`' portion. When the namespace was removed, it was easier to forget that the CLI type was being used, and so we reverted to using language-specific monikers. For example, '`string`', all lower case, was used instead of '`String`' with the capital first letter. These mistakes are hard to spot, since it appears that the type is correctly written

Generally, user types present less difficulty, as their name and namespace holds across language boundaries. There are still quirks when user types are exported as nested classes. For instance, class types exported by SML.NET are nested in their respective module. Thus, a class `Logger` defined in module `ML_Logger` would be accessed using the moniker `Aspect_ML_Logger+Logger`.

Our evaluation also noted a severe problem with the accidental selection of join points that could not be overcome using source code analysis tools. In Figure 3.1, the regular expressions are used in the pointcut designator's argument to create a property-based crosscut. However, such regular expressions can make unexpected join point selections. Before evaluation, we made the general assumption that these extra join points could be spotted in source code. If this were the case, then with a careful examination of component source could be used to predict the behaviour of the final application and on the basis of this prediction correct behaviour could be ascertained. However, evaluation tests involving components written in SML.NET indicate superfluous join points are not always visible from source. Assemblies generated by the SML.NET compiler can contain considerably more types than could be inferred from the source code. For example, Figure 3.2 defines a SML module with methods `main` and `SelfTest` at the module level and methods `Fibonacci` and `FibSeries` in the class

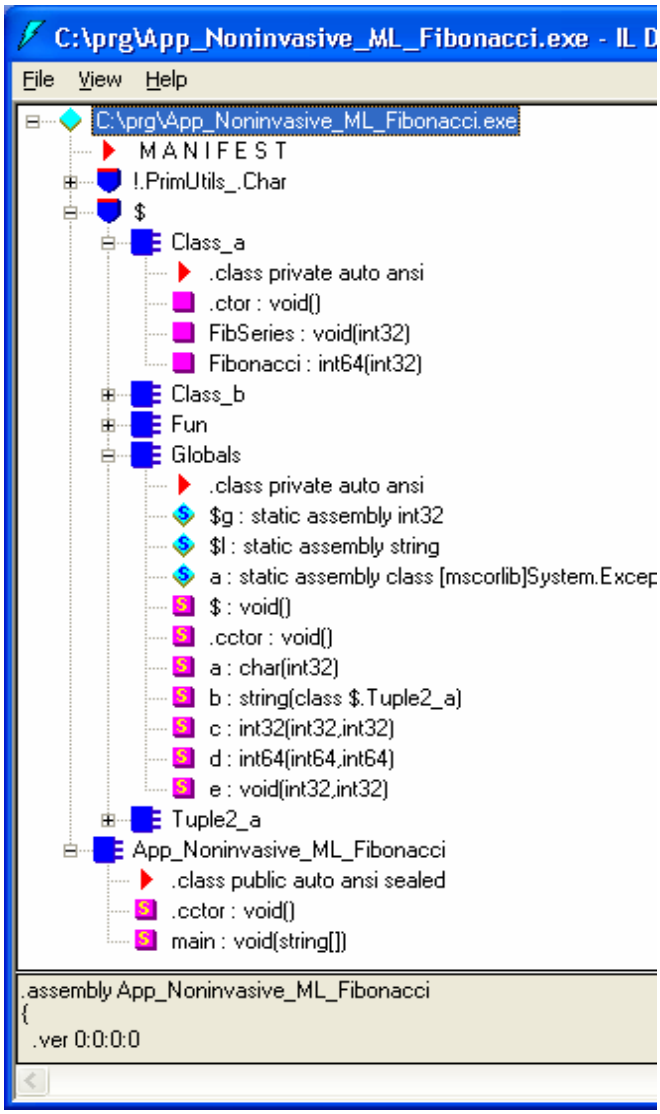

Figure 3.3: Disassembler view of types contained in a component written in SML.NET source in Figure 3.2.

`FibonacciSeries`. Using the directive "`export App_Noninvasive_ML_Fibonacci`" to compile this source results in a CLI component containing a surprising number of additional types. As shown in Figure 3.3, a disassembler view of

the type definitions in the component uncovers a large number of types for which there are no explicit declarations in the source code. As expected, there is a type corresponding to the module that contains the implementation of `main` and `SelfTest`, and there is a class corresponding to the `FibonacciSeries` class declaration that contains the implementations of `Fibonacci` and `FibSeries`. The difficulty is that there are other types such as `Globals` with methods such as "`static char a(int32 A_0)`" that would match the property-based crosscut for logging shown in Figure 3.1.

Our tests verified that property-based crosscutting has useful reusability characteristics, but its drawbacks make it quite difficult to use when obtaining strong assurances of correct and predictable behaviour is a concern. Difficulties in specifying XML using a language independent type system made it difficult to write crosscutting specifications by hand. While these could be overcome with diligence, the inadvertent join point selection could not. New join points introduced by the compiler when a component was compiled could not be determined through source code analysis, whether done by a human or via an automated application.

4. LOGGING VIA ATTRIBUTE-BASED CROSSCUTTING

To contrast attribute-based crosscutting, we revised the logging aspect to exploit attribute types for join point selection, and composed this new logging aspect with the Fibonacci series algorithm implementations. Our evaluation noted the use of attribute-based crosscutting offers a more succinct and accurate means of applying crosscutting functionality, and attributed-based crosscutting avoided the unexpected join point selection that prevented application behaviour from being predicted from source code analysis.

Attribute-based crosscutting specifications complement an attribute type [8] that allows access to aspect functionality through annotation of component source. In contrast to property-based crosscutting, attribute-based crosscutting uses attribute type names in place of join point implementation details such as types and type member signatures. When using attribute type names, the grammar for pointcut specifications is unchanged when it comes to the primitive pointcut designators available, but the parameters used for designators are changed. Rather than signature or type name arguments, primitive pointcut designators are parameterized with attribute tags describing the attribute type name. In the case of the Weave.NET, these attributes are implemented by the CLI's custom attribute types.

The contrast between property-based and attribute-based crosscutting can be seen in Figure 4.1. The top pane of the figure contains the execution pointcut specification used in Figure 3.1 to select execution join points for logging. In this pane, the selection of method execution join points is based on a partial method signature. In the bottom pane, the specification is revised to select methods tagged with an attribute type with the name `Logging`. This second version contains considerably fewer terms than the first, but it is reliant on the ability to annotating method source with an attribute type.

An example application of attribute types is shown in Figure 4.2, where methods of the Fibonacci series algorithm are bound to logging functionality. This example emphasizes the attribute annotations by marking them in bold.

```
<execution>
  <method_signature>
    <return_type>
      <type_name>*</type_name>
    </return_type>
    <join_point_type>
      <type_name>*</type_name>
    </join_point_type>
    <method_name>*</method_name>
    <parameters>
      <parameter>
        <type_name>Int32</type_name>
      </parameter>
    </parameters>
  </method_signature>
</execution>
```

```
<execution>
  <attribute>Logging</attribute>
</execution>
```

Figure 4.1: Contrast between property-based crosscutting (top) and an aspect-based crosscutting (bottom).

```
public class FibonacciSeries {
 [Logging]
  public void FibSeries(int seriesLen) {
    for (int i = 0; i<= seriesLen; i++) {
      long result = Fibonacci(i);
      System.Console.WriteLine("Element \t"+
                        i+ "\tvalue \t"+result);
    }
  }
 [Logging]
  public long Fibonacci(int n) {
    if (n > 1)
      return this.Fibonacci(n-1)
           + this.Fibonacci(n-2);

    return 1;
  }
}
```

Figure 4.2: Fibonacci series enumerator annotated with attributes to identify methods for logging.

In our tests, we noted attribute-based crosscutting provides an alternative means of identifying CLI metadata that avoids mistakes made with property-based crosscutting specifications that are extremely difficult to detect. Recall that writing property-based crosscutting involves specifying join points in terms of metadata that is native to the CLI. There is little help available from the weaver for detecting erroneous type specifications, as it is hard to design a weaver that can distinguish between types that are specified correctly and those that are specified in error. For instance, the method parameters in Figure 4.2 are of type `int`. `int` is the C# moniker for the CLI type `System.Int32`, and thus the short form `Int32` appears in the property-based crosscut in Figure 4.1. Should the type `int` appear accidentally in the crosscutting specification, one would expect the weaver to complain. However, it is legitimate for a programmer to define a custom CLI type by the name of `int` in a different namespace. Even if we require that type names in the crosscutting specifications include a full namespace, `int` is still a valid user defined type. Attribute-based property selection avoids the issue of detecting errors made when the language type name is mapped to the CLI type name mappings, since the placement of attributes on types or type members avoids the need to deal with join point

selection in terms of CLI-specific type names. In effect, the use of attributes represents the introduction of language-independent monikers for types and type members.

```
structure App_Invasive_ML_Fibonacci
  : sig val main: string option array option -> unit
    end =
struct
_classtype FibonacciSeries()
 with
  {Aspect_CS_Logging.Logging()} Fibonacci (n) =
    case(n) of
      0 => (Int64.fromInt(1))
    | 1 => (Int64.fromInt(1))
    | n => (this.#Fibonacci(n-1) + this.#Fibonacci(n-2))
 and
  {Aspect_CS_Logging.Logging()} FibSeries (n) =
    case(n) of
     ~1 => ()
    | n => (this.#FibSeries (n-1);
            print "Element\t"; print (Int.toString (n));
            print "\t value \t";
            print (Int64.toString(this.#Fibonacci (n)));
            print "\n" )
 end
      ...
end
```

Figure 4.3: SML.NET implementation of Figure 3.2 updated to exploit custom attributes.

Also, attribute-based property selection avoided unexpected join point selection, since attributes follow the implementation of the tagged method. With revisions to include attributes, the SML-based Fibonacci series algorithm in Figure 3.2 takes on the appearance of that of Figure 4.3, where attributes appear in bold. Note that the definitions of `main` and `SelfTest` have been removed for brevity. As before, additional helper types will appear in the compiled assembly. However, an examination of the metadata of the assembly indicates that only those methods explicitly tagged at the source code level will have their metadata description annotated by the logging attribute in the compiled assembly. Thus, applying logging on the basis of attributes rather than method signature, constrains logging to the `Fibonacci` and `FibSeries` methods.

So, in addition to making it simpler to specify aspect-based software product families, attribute-based property eliminated problems with predictability that had prevented strong assurances of correct application behaviour from being made from an examination of component source.

5. CONCLUSIONS

In the context of language-independent AOP, attribute-based crosscutting specifications have two important advantages over property-based crosscutting. Use of attributes represents the introduction of language-independent monikers for types and type members that simplify the specification of crosscutting in a language independent fashion. Second, attribute-based crosscutting will not inadvertently match unwanted join points introduced during the translation of a component source to a language-independent substrate. This occurs because only those component structures explicitly annotated with an attribute at the source code level will have their metadata description annotated by that logging in the compiled component.

6. REFERENCES

1. Bloch, J. JSR-000175 A Metadata Facility for the JavaTM Programming Language, http://jcp.org/aboutJava/communityprocess/review/jsr175/, 2003.
2. Costanza, P. Dynamically scoped functions as the essence of AOP. *ACM SIGPLAN Notices, 38* (8). 29 - 36.
3. ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI), http://www.ecma-international.org/publications/standards/ecma-335.htm, Geneva, 2003.
4. Filman, R.E. and Friedman, D.P. Aspect-Oriented Programming is Quantification and Obliviousness *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, Minneapolis, USA, 2000.
5. Kennedy, A., Russo, C. and Benton, N. SML.NET 1.1 User Guide, http://wwww.cl.cam.ac.uk/Research/TSG/SMLNET/smlnet.pdf, Cambridge, U.K., 2003.
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. Getting Started with AspectJ *Communications of the ACM*, 2001, 59-65.
7. Lafferty, D. Aspect-Based Properties *Dept of Computer Science*, Trinity College Dublin, Dublin, 2004.
8. Lafferty, D. and Cahill, P.V., Attribute Types. in *Submitted for review to ECOOP 2005*, (Glasgow, Scotland, 2005), Springer.
9. Lafferty, D. and Cahill, V., Language-Independent Aspect-Oriented Programming. in *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, (Anaheim, California, USA, 2003).
10. Masuhara, H. and Kiczales, G., Modeling Crosscutting in Aspect-Oriented Mechanisms. in *European Conference on Object-Oriented Programming (ECOOP 2003)*, (Darmstadt, Germany, 2003), Springer-Verlag.
11. Microsoft. Standard ECMA-334 C# Language Specification, ECMA International - European association for standardizing information and communication systems, 2001.
12. Microsoft. Visual Basic Development Center, http://msdn.microsoft.com/vbasic, 2004.
13. Popovici, A., Alonso, G. and Gross, T. Spontaneous Container Services *European Conference on Object-Oriented Programming (ECOOP 2003)*, Springer, Darmstadt, Germany, 2003.
14. Szyperski, C., Gruntz, D. and Murer, S. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 2002.
15. The AspectJ Team. The AspectJ Programming Guide (V1.0.6), http://download.eclipse.org/technology/ajdt/aspectj-docs-1.0.6.tgz, 2002.

Tarantula: Killing Driver Bugs Before They Hatch

Julia L. Lawall
DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Gilles Muller and Richard Urunuela
OBASCO Group, École des Mines de Nantes-INRIA, LINA
44307 Nantes Cedex 3, France
gilles.muller@emn.fr, rurunuel@emn.fr

Abstract

The Linux operating system is undergoing continual evolution. Evolution in the kernel and generic driver modules often triggers the need for corresponding evolutions in specific device drivers. Such collateral evolutions are tedious, because of the large number of device drivers, and error-prone, because of the complexity of the code modifications involved. We propose an automatic tool, Tarantula, to aid in this process. In this paper, we examine some recent evolutions in Linux and the collateral evolutions they trigger, and assess the corresponding requirements on Tarantula.

1 Introduction

The Linux operating system (OS) is undergoing continual evolution to improve performance, meet new hardware requirements, and improve the software architecture. When evolution in one OS kernel module causes the interface of the module to change, the need for evolution percolates out into other OS services. This collateral evolution can become quite tedious when many modules depend on the interface and the modifications required are complex. It is also error-prone, because of the difficulty of understanding both the evolution and its impact on the dependent modules. As a result, some collateral evolutions happen very slowly and bugs are introduced. The problems are compounded for modules outside the kernel source tree, which are maintained by developers different from those performing the original evolution and who may not have access to complete information about evolution requirements.

Device drivers are particularly vulnerable to the need for collateral evolution. As illustrated in Figure 1, drivers depend on services provided by the kernel and by modules generic to various families of devices. Due to the rapid proliferation of new devices, there are many drivers. Indeed, an evolution in a generic function defined by the kernel can require modification of over a hundred driver files. Drivers are also a high priority for users, who, in an open system such as Linux, can submit patches to update the drivers for their machines, despite not having a complete understanding of the implications of the evolution.

A particularly striking example of the difficulty of driver evolution is the case of the function `check_region` used in driver initialization. In Linux 2.4.1, this function was called 322 times in 197 driver files. Starting in Linux 2.4.2 (Feb. 2001), the use of this function began to be eliminated, because changes in the driver initialization process implied that its use could cause race conditions. Eliminating `check_region` requires both replacing it with a call to `request_region` and introducing some cleanup code at any subsequent code point that indicates failure of the driver initialization process. Identifying the latter code points entails a non-trivial control-flow analysis possibly across multiple functions. Accordingly, bugs have appeared in the process of eliminating `check_region` and the evolution is not complete as of Linux 2.6.10 (Dec. 2004), even though the function has been deprecated since Linux 2.5.54 (Jan. 2003).

To reduce the difficulty of performing collateral evolution of device drivers, we propose to develop an automatic tool, Tarantula, to aid in the evolution process. Using Tarantula, a collateral evolution is described as a set of rewrite rules, referred to as a semantic patch, that specify the affected code patterns and associated changes. Given a driver and a semantic patch, Tarantula identifies driver code that matches the code patterns and interactively proposes the associated changes. If the user accepts a change, Tarantula transforms the code automatically. We envision that a developer who modifies the interface of a generic module also writes a corresponding semantic patch. This developer then applies the semantic

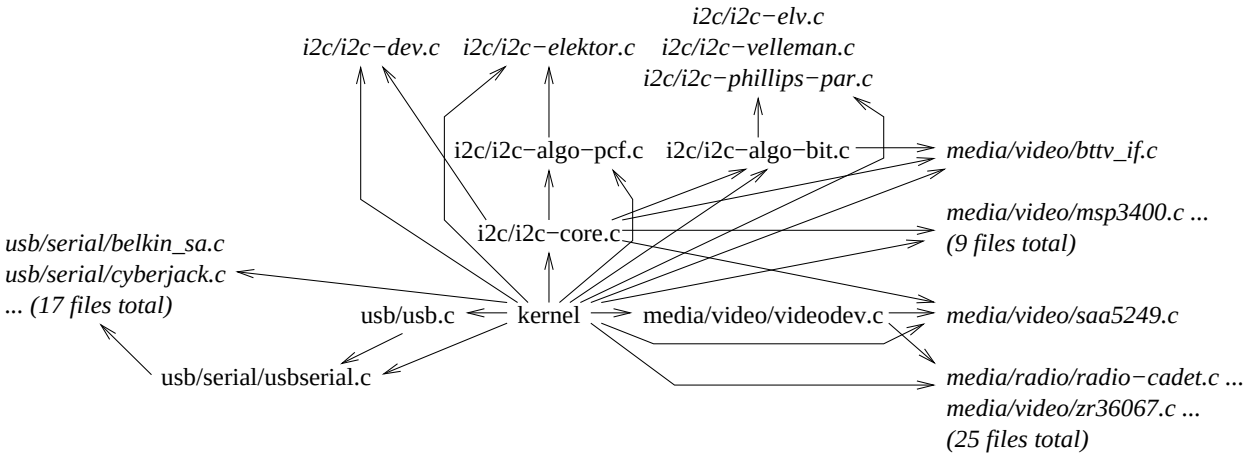

Figure 1: Some kernel dependencies in Linux 2.4.27 (device drivers are shown in italics)

patch to drivers in the kernel source tree, profiting from the interactivity of Tarantula to identify overlooked code patterns and code fragments that are matched inappropriately. When a semantic patch has been validated on the kernel sources, the developer makes it publicly available for use by the maintainers of drivers outside the kernel source tree.

In this paper, we present preliminary work in the development of Tarantula. Based on a study of evolution in driver code across versions of Linux 2.4 through 2.6, we present examples that illustrate the kinds of code modification that collateral evolution entails. In each case, we assess the corresponding requirements on the expressiveness of semantic patches and on the power of the underlying rewriting engine. In terms of expressiveness, we find the need for rewrite rules that describe control-flow paths, for which we propose to use temporal logic (CTL) [10]. To support such rules, we find the need for a rewriting engine that includes inter-procedural control-flow analysis, alias analysis, and constant propagation.

The rest of this paper is organized as follows. Section 2 presents some examples of evolution in Linux. Section 3 assesses these examples in terms of the requirements that they pose on Tarantula. Finally, Section 4 presents related work and Section 5 concludes.

2 Examples

In this section, we present some representative examples of evolution in Linux and the difficulties that have arisen in the collateral evolutions in driver code.

2.1 Elimination of `check_region`

The function `check_region` is used in the initialization of device drivers, in determining whether a given device is installed. In early versions of Linux, the kernel initializes device drivers sequentially [18]. In this case, a driver determines whether its device is attached to a given port as follows: (i) calling `check_region` to find out whether the memory region associated with the port is already allocated to another driver, (ii) if not, then performing some driver-specific tests to identify the device attached to the port, and (iii) if the desired device is found, then calling `request_region` to reserve the memory region for the current driver. In more recent versions of Linux, the kernel initializes device drivers concurrently [5]. In this case, between the call to `check_region` and the call to `request_region` some other driver may claim the same memory region and initialize the device. Starting with Linux 2.4.2, device drivers began to be rewritten to replace the call to `check_region` in step (i) with a call to `request_region`, to actually reserve the memory region. Given this change, if in step (ii) the expected device is not found, then `release_region` is used to release the memory region.

Eliminating a call to `check_region` requires replacing it by the associated call to `request_region` and inserting calls to `release_region` along error paths. In the first step, it is necessary to find the call to `request_region` that is associated with the given call

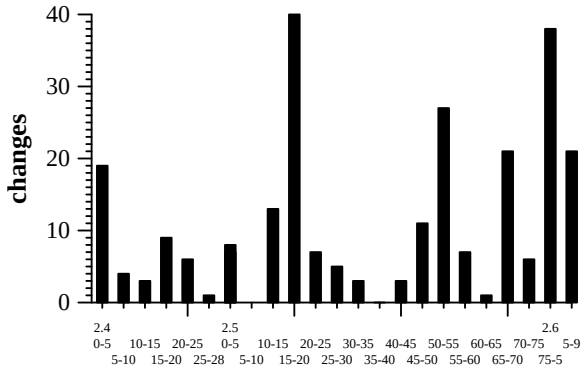

Figure 2: `check_region` elimination in Linux 2.4-2.6

to `check_region`. In Linux 2.4.1, the call to `request_region` is in the same function for only 56% of the calls to `check_region`.[1] In the remaining cases, an interprocedural analysis is needed. In the second step, it is necessary to identify code points at which it is known that the expected device has not been found and thus `release_region` is required. Such points include returning an error value, as found in 75% of the functions calling `check_region`, and going around a loop that checks successive ports until finding one with the desired device, as found in 23% of these functions. At such code points, it may be the case that only a subset of the incoming paths contain a call to `check_region`, as occurs in 31% of the functions calling `check_region`. In this case, the call to `release_region` must be placed under a conditional.

Both steps in eliminating `check_region` are difficult and time-consuming. This difficulty has lead to the slow pace of the evolution, as shown in Figure 2. The evolution is still not complete as of Linux 2.6.10.

2.2 An extra argument for `usb_submit_urb`

The function `usb_submit_urb`, defined until Linux 2.5.7 in the generic module `usb/urb.c`, until Linux 2.5.20 in the generic module `usb/core/usb.c`, and subsequently in the generic module `usb/core/urb.c`, implements the passing of a message, implemented as USB Request Block (urb), by a USB driver. This function uses the kernel memory-allocation function, `kmalloc`, which must be passed a flag indicating the circumstances in which blocking is allowed. Up through Linux 2.5.3, the flag was chosen in the implementation of `usb_submit_urb` as follows:

```
in_interrupt () ? GFP_ATOMIC : GFP_KERNEL
```

Comments in the file `usb/hcd.c`, however, indicate that this solution is unsatisfactory:

```
// FIXME paging/swapping requests over USB should not
// use GFP_KERNEL and might even need to use GFP_NOIO ...
// that flag actually needs to be passed from the higher level.
```

Starting in Linux 2.5.4, `usb_submit_urb` takes one of the following as an extra argument: `GFP_KERNEL` (no constraints), `GFP_ATOMIC` (blocking not allowed), or `GFP_NOIO` (blocking allowed but not I/O). The driver programmer selects one of these constants according to the context of the call to `usb_submit_urb`.

Choosing the extra argument of `usb_submit_urb` requires a careful analysis of the surrounding code as well as an understanding of how this code is used by more generic modules. The only relevant documentation in the Linux code is the comments preceeding the definition of `usb_submit_urb` starting in Linux 2.5.4. These comments state that `GFP_ATOMIC` is required in a completion handler, in code related to handling an interrupt, when a lock is held (including the lock taken when turning off interrupts), when the state of the running process indicates that the process may block, in certain kinds of network driver functions, and in SCSI driver queuecommand functions. Many of these situations, however, are not explicitly indicated by the code surrounding the call to `usb_submit_urb`. Instead, they require an understanding of the contexts in which the function containing the call to `usb_submit_urb` may be applied.

The difficulty in understanding the conditions in which `GFP_ATOMIC` is required and identifying these conditions in driver code is illustrated by the many calls to `usb_submit_urb` that were initially transformed incorrectly. Figure 3 lists the versions in Linux 2.5 in which corrections in the use of `usb_submit_urb` occur and the reason for each correction. In each case, the error was introduced in Linux 2.5.4 or when the driver entered the kernel source tree, whichever came later. A major source of errors is the case where the function containing the call to `usb_submit_urb` is stored in a structure or passed to a function, as these cases require extra knowledge

[1] This analysis and the other analyses reported for the elimination of `check_region` were carried out using CIL [17], which requires parsing each file. Due to problems obtaining appropriate compilation arguments and incompatibilities between the Linux 2.4.1 code and the gcc 3.3.3 compiler, we were only able to parse 78% of the driver files successfully. The percentages reported here are as compared to this set of parsable files.

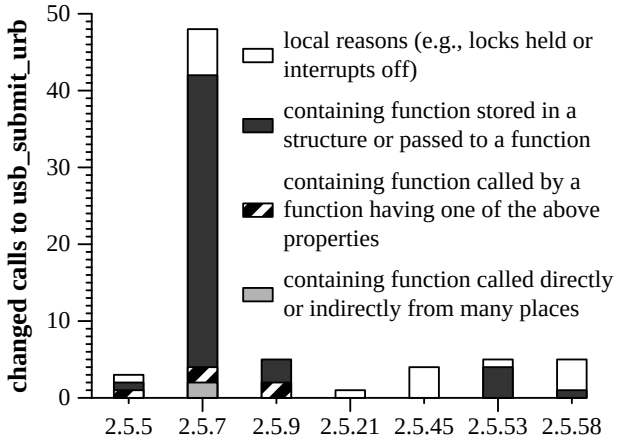

Figure 3: Linux 2.5 versions in which `GFP_ATOMIC` replaces `GFP_KERNEL` in a call to `usb_submit_urb`

about how the structure is used or how the function uses its arguments. Indeed, in the `serial` subdirectory, all of the calls requiring `GFP_ATOMIC` fit this pattern and all were initially modified incorrectly (and corrected in Linux 2.5.7). Surprisingly, in 17 out of the 71 errors, the reason for using `GFP_ATOMIC` is locally apparent, reflecting either carelessness or insufficient understanding of the conditions in which `GFP_ATOMIC` is required. Indeed, in Linux 2.6.10, in the file `usb/class/audio.c`, `GFP_KERNEL` is still used in one function where interrupts are turned off.

2.3 Introduction of `video_usercopy`

A Linux ioctl function allows user-level interaction with a device driver. Copying arguments to and from user space is a tedious but essential part of the implementation of such a function. In Linux 2.5.7, in the generic module `media/video/videodev.c`, a wrapper function was introduced to encapsulate this argument copying. This function was refined in Linux 2.5.8 and named `video_usercopy`. As of Linux 2.6.9, `video_usercopy` was used in 29 `media` files.

Introducing the use of `video_usercopy` requires primarily (i) identifying the ioctl function and (ii) rewriting its code to eliminate copying between user and kernel space. An ioctl function does not have a fixed name, but can be recognized as the value stored in the `ioctl` field of the structure implementing the driver interface. Copying between user and kernel space is typically implemented by using the functions `copy_from_user` and `copy_to_user` to copy information to and from a local structure specific to each ioctl command. `Video_usercopy` provides the ioctl code with a generic-typed kernel pointer to this information. The ioctl code must thus be modified to cast this pointer to the structure type used by each command and to replace references to the local structure by pointer dereferences. The latter transformation can be quite invasive. For example, in the ioctl function of `media/radio/radio-typhoon.c`, 61% of the lines of code changes between Linux 2.5.6 and 2.5.8.

The function `video_usercopy` is not specific to media drivers, and thus there has been interest in making the function more generally available [9]. Some evidence of the difficulties this may cause are provided by the case of `i2c/other/tea575x-tuner.c` in which `video_usercopy` was introduced in Linux 2.6.3. In this file, the calls to `copy_from_user` and `copy_to_user` were not removed. The bug was never fixed. Instead, the use of `video_usercopy` was removed from this file in Linux 2.6.8.

3 Requirements

The semantic patches of Tarantula must (i) identify the code to modify, (ii) describe how to construct the new code, and (iii) describe the impact on the existing context. We review the above examples in terms of these issues, and identify the requirements they place on Tarantula. Required features are shown in italics.

In the `check_region` example, the code to modify is indicated by a use of the function name. The new code that replaces a call to `check_region` is determined by the call to `request_region` that would subsequently be executed at run time. To specify the connection between these calls, the rewrite rules must be able to describe a control-flow path. For this, we propose to use *temporal logic* [10], a logic that describes relationships between successive events, instantiated here as successive program constructs. So that the rewriting engine can identify such paths in the source program, it must include a *control-flow analysis*. Because the calls to `check_region` and `request_region` are not always in the same function, the control-flow analysis must be *inter-procedural*. Finally, replacement of `check_region` by `request_region` implies that calls to `release_region` must be inserted in the context. This again requires rewrite rules that describe paths, and temporal logic and control-flow analysis are useful here. Some of the paths requiring `release_region` are interprocedural error paths. *Constant propagation* of error return val-

ues is thus needed to restrict the analysis to meaningful control-flow paths.

In the `usb_submit_urb` example, the code to transform is again indicated by a use of the function name. The new argument is determined by properties of the enclosing calling context. Again, these properties are interprocedural and depend on control flow, and thus temporal logic and control-flow analysis are useful. In a few cases, functions containing calls to `usb_submit_urb` are stored in structures or variables local to the driver are subsequently invoked through these entities. These cases require *alias analysis*.

In the `video_usercopy` example, identifying the code to transform requires finding the ioctl function, which entails reasoning about *global structure declarations*. The introduction of `video_usercopy` has a significant effect on the context: calls to `copy_from_user` and `copy_to_user` disappear, and the types of the variables manipulated by these functions change. To express these modifications, the rewrite rules must be able to express properties of *local-variable declarations and uses*.

We have previously used rewrite rules including temporal logic to describe the modifications needed to reengineer the source code of a legacy OS to support the Bossa process scheduling framework [1, 16]. Those rules were implemented using the CIL infrastructure for C program analysis and transformation [17]. For Tarantula, we will generalize this work by extending the rewrite rule language to describe a more general set of transformations, and by improving the rewriting engine to include more complex variants of the analyses, such as inter-procedural analyses exploiting constant values. Of the required analyses, CIL already provides intra-procedural control-flow analysis, inter-procedural constant propagation, and inter-procedural alias analysis.

4 Related Work

Our work involves the description of code patterns requiring evolution and the transformation of code matching these patterns using rewrite rules. This work is related to pattern-based approaches to bug finding and to techniques that allow the description of code modifications such as Aspect-Oriented Programming (AOP).

Recent years have seen a surge of interest in automatic approaches to detecting bugs in large pieces of software, including the Linux operating system [6, 7, 8, 14]. These approaches rely on identifying required code patterns and then detect code fragments that are inconsistent with these patterns. In the context of Linux, most of the bugs found using these approaches are in device driver code. We believe that the patterns used by these approaches derive largely from the interface provided by the kernel and generic modules. In the context of evolution of this interface, existing approaches detect bugs after they appear, while our approach prevents bugs by providing assistance in the evolution process. Our work can also be viewed as introducing a new source of code patterns into consideration. While previous work has focused on patterns identified within a single version of Linux, we consider patterns derived from evolution.

AOP is a programming paradigm that isolates the implementation of a modular crosscutting concern in a single unit, known as an aspect [12]. An aspect includes both code implementing the concern and directives indicating how to integrate this code with an existing base program. Coady *et al.* have investigated the use of aspects in OS code to improve modularity, and have considered the impact of OS evolution on these aspects [2, 3, 4]. Semantic patches can be viewed as a form of aspects, as they specify code and a means of determining where this code should be introduced. Nevertheless, the goals of our approach, and hence the mechanisms employed, are different. AOP is directed towards the complete implementation of a functionality that is somewhat orthogonal to the base program. Thus, for example, the widely-used aspect system, AspectJ [11], does not permit fine-grained modification of the base program, such as changing the type of a local variable. Our approach is directed towards specifying modifications to a portion of an integral functionality, specifically the interaction with the interface of a more generic module. Accordingly, our approach allows describing much more invasive, finer-grained transformations and requires more complex supporting analyses.

The Splice aspect system allows an aspect to use program analysis to specify where a base program should be transformed [15]. The specification is described in terms of logic programming rules combined with operators expressing temporal properties. Based on our previous experience in describing temporal properties in the reengineering of Bossa, we plan to use temporal logic directly, rather than via logic programming. The precision of the analyses used by Splice has been restricted to ensure scalability to large programs. Because we have observed that device drivers typically have shallow call graphs,

we plan to favor analysis precision over efficiency. Finally, Splice has only been used to implement lock insertion and a loop transformation, whereas we target a much wider range of transformations.

Our use of temporal logic was originally inspired by that of Lacey *et al.* on using temporal logic to specify program transformations [13].

5 Conclusion

Keeping drivers up to date is known to be difficult, due to the large number of drivers and the varying levels of programmer expertise. In this paper, we have proposed Tarantula to provide automatic assistance in evolving a driver to match changes in the interface of more generic parts of the OS. Tarantula is based on semantic patches, which provide (i) precise description of the contexts in which evolution is required, (ii) encapsulation of relevant information about external functions and data structures, and (iii) help with the tedious process of analyzing the driver file to determine where the evolution applies. So far, besides the examples cited here, we have found around 30 evolutions in driver directories such as `cdrom`, `ide`, `pcmcia`, and `usb` where Tarantula would be useful. We plan to continuing studying driver code to find a more complete set of examples. Our next step will be to refine the language of semantic patches and develop the supporting program analysis infrastructure.

Acknowledgments We would like to thank Anne-Françoise Le Meur and Laurent Réveillère for their comments on earlier versions of this paper.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.

[2] S. Bray, M. Yuen, Y. Coady, and M. E. Fiuczynski. Managing variability in systems: Oh what a tangled OS we weave. In *Workshop on Managing Variabilities Consistently in Design and Code*, Vancouver, Canada, Oct. 2004.

[3] Y. Coady. *Improving Evolvability of Operating Systems with AspectC.* PhD thesis, The University of British Columbia, July 2003.

[4] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 50–59, Boston, MA, Mar. 2003.

[5] A. C. de Melo, D. Jones, and J. Garzik, 2001. `http://umeet.uninet.edu/umeet2001/talk/15-12-2001/arnaldo-talk.html`.

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

[7] D. R. Engler, D. Y. Chen, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Canada, Oct. 2001.

[8] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 38–51, Berlin, Germany, June 2002.

[9] C. Hellwig, 2003. `http://www.cs.helsinki.fi/linux/linux-kernel/2003-20/1120.html`.

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

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997.

[13] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001*, number 2027 in Lecture Notes in Computer Science, pages 52–68, Genova, Italy, 2001.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Fransisco, CA, Dec. 2004.

[15] S. McDirmid and W. C. Hsieh. Splice: Aspects that analyze programs. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, number 3286 in Lecture Notes in Computer Science, pages 19–38, Vancouver, Canada, Oct. 2004.

[16] G. Muller, J. Lawall, J.-M. Menaud, and M. Südholt. Constructing component-based extension interfaces in legacy systems code. In *ACM SIGOPS European Workshop 2004 (EW2004)*, pages 80–85, Leuven, Belgium, Sept. 2004.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, number 2304 in Lecture Notes in Computer Science, pages 213–228, Grenoble, France, Apr. 2002.

[18] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition.* O'Reilly, June 2001.

On the Configuration of Non-Functional Properties in Operating System Product Lines*

Daniel Lohmann, Olaf Spinczyk, Wolfgang Schröder-Preikschat
{dl,os,wosch}@cs.fau.de
Friedrich-Alexander-University Erlangen-Nuremberg, Germany
Department of Computer Science 4

ABSTRACT

Reaching a good separation, maintainability and configurability of non-functional concerns like performance, timeliness or dependability is a frequently expressed but still unrealisable hope of using AOP technology. Non-functional properties have the tendency to be emergent, that is, they have no concrete representation in the code, but appear through the complex interactions between software components in the whole. This makes it is very hard, if not impossible at all, to express them in a configurable manner by objects or even aspects. The architecture of a software system, however, is known to have a high impact on some non-functional properties. Thus, it may be possible to reach configurability of non-functional software properties by the means of reconfigurable software architectures. This paper discusses the connection between non-functional and architectural properties in the domain of operating system product lines.

1. INTRODUCTION

Reaching better separation of concerns in software systems was and is the driving factor for the development of AOP technologies. One of the big hopes associated with the application of AOP is to get a clearly modularized implementation of even so called *non-functional concerns*. The non-functional properties of a software system are those properties that do not describe or influence the principal task / functionality of the software, but can be observed by end users in its runtime behaviour. *Performance* or *resource utilization* are the most common examples for non-functional properties, but also less observable properties like *robustness* or *dependability* are important members of the class.[1] Even if non-functional properties have no impact on the primary functionality of the software system, they have a big impact on its applicability in the real world. A system that provides perfect functionality, but works terribly slow, is just unusable. Non-functional properties are therefore important concerns, their controllability may be crucial for the success of a software project.

This is especially true in the domain of embedded systems, where hardware cost pressure leads to strictly limited resources in terms of CPU and memory. Under such circumstances, non-functional concerns like memory usage or timeliness may even dominate the functional properties of the final product. As a consequence, there is an ongoing tendency to develop operating systems for embedded devices as tailorable *product lines* that provide a more or less fine-grained selectability of functional features. This facilitates, by leaving out optional functions, some optimization of the system for specific memory constraints. However, existing operating system product lines provide only very limited configurability with respect to other non-functional properties like timeliness, protection or robustness.

1.1 Problem Analysis

The problem with most non-functional properties is that they are *emergent properties*. They are neither visible in the code nor structure of single components, but "suddenly" emerge from the orchestration of many components into a complete system. Properties that manifest in the integrated system only are indeed crosscutting, as they *result* from certain (unknown) characteristics of every single component. Due to their inherent emergence it is, however, not possible to tackle them by decomposition techniques like AOP. They need to be understood holistically, that is, on the global scope of software development. One could say they need to be addressed by *"holistic aspects"*, meaning that the realization of non-functional concerns does not crosscut (just) the code, but the whole *process* of software development.

1.2 The Role of Architecture

The architecture of a software system is known to have a high impact on many non-functional properties. Architecture can well be understood as a *"holistic aspect"*, as it encompasses the set of fundamental design decisions made at the early stages of the software development process. Many architectural decisions are actually driven by non-functional requirements, based on *experience* regarding their effect on such properties. In the operating systems domain, for instance, it is *known* that the isolation of every system components into an own address space (as in μ-kernel OS) has positive effects on safety and fault protection, while a monolithic kernel structure is *known* to lead to lower demands on system resources. From the functional viewpoint (of an application) there is, however, no real difference between a μ-kernel OS and a monolithic kernel OS[7]. Hence, the architecture of the operating system is itself an all-embracing non-functional property[10].

Architecture is usually seen as being something fixed. Most architectural decisions cover large parts of the code, which makes it very expensive to change them later. However, as they crosscut the code they are potentially addressable by aspects. Thus, it should be possible to implement architecture in a *configurable* way and thereby leverage towards an *indirect* configuration of emergent properties.

*This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4

[1]This definition of "non-functional" is intentionally from the perspective of an end user. Other stakeholders (e.g. developers or salesmen) would probably define very different properties like *portability* or *unique selling points* as "non-functional".

In the CiAO project[11] we are currently working on the development of an operating system family that provides configurability of certain architectural properties.

1.3 Structure of the Paper

The rest of the paper is organized as follows: In the next section, we discuss the influence of architectural concerns to non-functional properties in the domain of operating systems. Section 3 describes our approach towards the design of architecture-neutral OS components, which is also explained by an example in section 4. Finally, conclusions are drawn and the paper is briefly summarized.

2. CONCERNS OF OS ARCHITECTURE

To reach configurability of architectural properties in operating system product lines, it is necessary to have *architecture-transparent* OS components, that is, components which are designed and implemented to be independent from the actual architecture to use. It is essential to clearly separate the functional component code from those elements that reflect architectural decisions. The following lists some of the more important properties that "make" the architecture of an operating system kernel[10], together with the emergent properties they are known to imostly unvisible and nfluence. The focus is on embedded systems:

synchronization If the kernel supports concurrent/parallel execution of control flows, concurrent data access must not lead to race conditions. Synchronized access to data may be implemented by waiting-free algorithms, special hardware support (e.g. atomic CPU operations), interrupt locks or higher-order synchronization protocols. Locks may be allocated on a coarse-grained or fine-grained base. The chosen kernel synchronization strategy has a noticeable impact on **latency**, **timeliness** and **performance**.

isolation The different components of an operating system may have access to the whole system state or to well-isolated subsets only. Components may be isolated by design through type-safe programming languages, by hardware support (segmentation or address spaces via memory management units (MMUs) or translation lookaside buffers (TLBs)) or even by distributing them across hardware boundaries. Isolation may cause additional requirements on data alignment, sharing and interaction. The chosen isolation strategy has a noticeable impact on **memory usage**, **safety**, and **performance**.

interaction System services may be invoked and interact with each other by plain procedure calls, local message passing, inter-process calls (IPCs) or remote procedure calls (RPCs). Interaction may imply implicit synchronization, data duplication or (in the case of RPCs) even fail on occasion. The chosen interaction strategy often goes in line with isolation. It has a noticeable impact on **latency**, **memory usage** and **performance**.

The above listed properties are fundamental building blocks of any operating system architecture[9]. In our research activities on applying AOP principles to the PURE operating system family[17, 13], we had the experience that it is not possible to implement an *ex post* configurability of such fundamental properties. The reason is that most architectural properties do not only lead to characteristic code patterns in the component code (which are addressable by aspects), but also to a number of *implicit constraints* that are not visible in the code. The developer of an implementation without isolation, for instance, implicitly relies on the possibility to pass complex data structures by simple untyped references. An implementation that uses message-based interaction implicitly relies at some places on the serialization of inter-component invocations. However, it is nearly impossible to detect which parts of the code implicitly rely on which constraints. An automatic transformation of such component code to another architecture, e.g. by aspects, is not feasible.

The integration of symetric multiprocessing (SMP) support into Linux is an impressive example for the enormous impact of an architectural property (kernel synchronization) to a non-functional property (performance). It is also a good example for the high costs of architectural transformations in legacy code: The first kernel release that supported SMP hardware was version 2.0. As most components still relied on the coarse-grained kernel synchronization scheme of earlier versions, it performed badly in SMP environments. To improve the performance property, a switch towards a fine-grained synchronization strategy was unavoidable. Hundreds of device drivers, file systems, and other components of the system had to be adapted[2]. Now the 2.6 kernel has fine-grained locking in almost all parts of the system and performs quite well, but the process took several years to complete.

3. THE CIAO APPROACH

Our conclusion from the experiences with PURE is that an operating system has to be designed *specifically* for architectural transparency. In the CiAO project we are now developing a new family of operating systems that fulfills the requirements for architectural configurability. This is a challenging task, as one needs to become aware of all the explicit and implicit elements that are induced by an architectural property. As it is not possible to build software without *any* architecture, a set of abstractions is needed that generalizes over the concrete property implementation. These abstractions are then used by components and later transformed, by aspects, into their architecture-specific representation.

The possible different implementations of architectural properties highly influence each other. Method calls are, for example, a suitable abstraction for the interaction concern. However, to be able to transform them by aspects into a message-based communication scheme, it is necessary to have a clear distinction between inter- and intra-component invocations. This can be realized by naming conventions. Moreover, if isolation enforces message-passing to other address spaces, untyped references must not be used as arguments, as they are not resolvable for transportation into another address space. Message-based interaction is, however, synchronized by design, while interaction by method calls is not. As a consequence, critical sections in the component code always have to be marked explicitly.

This implementation interdependence advises a bottom up design process. *Domain analysis* is the first step in finding suitable abstractions for a specific architectural property. Domain analysis encompasses a detailed analysis of the property implementations in all architectures to support, e.g. by taking a close look at existing operating systems. The result of domain analysis is a set of commonalities and differences between the architecture-specific implementations, represented as *feature diagrams*[4]. The commonalities are the anchor for developing the abstractions of the architecture-neutral model during the design phase. Differences

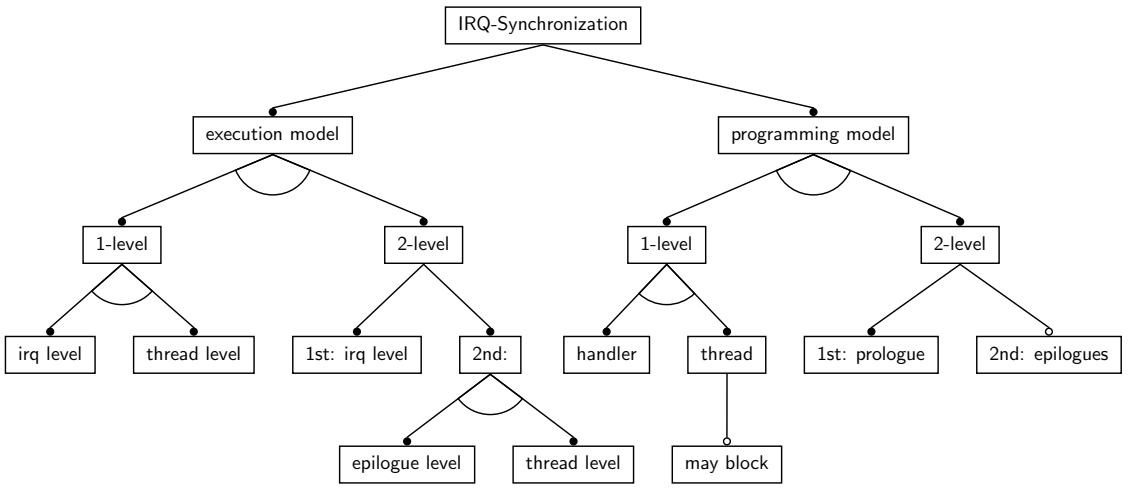

Figure 1: Feature diagram of the *IRQ Synchronization* domain

are integrated into the model step-by-step by generalization, if possible, or are separated out into architecture-specific models. The architecture-neutral model finally can be used as a reference architecture for the architecture-independent implementation of functional components.

4. EXAMPLE: INTERRUPT SYNCHRONIZATION

Most operationg system kernels support two different notions of *control flows*. Continuing, long-running control flows are typically supported by a *thread abstraction*. Control flows to perform short-term reactions on (non-deterministic) external hardware events are implemented by *interrupt handlers*. From the perspective of the operating system, a thread can be understood as walking top down through the kernel, while the control flow of interrupt handlers goes bottom up through the kernel. If, by any chance, interrupts and threads can meet on their execution paths (e.g. by accessing some common state), the kernel needs to ensure synchronized access to this state. The strategy provided for this purpose is usually referred to as *interrupt synchronization.* Interrupt synchronization is an important part of the kernel synchronization concern discussed above.

4.1 Domain Analysis

Analyzed Systems

The following systems were analyzed: Linux, Windows (NT/2000/XP), Solaris, PURE[1] and L4Ka[8]. For systems with SMP support the single-CPU case was analyzed.

General Observations

The execution of an interrupt control flow is initiated by hardware. In case of an IRQ signal, the CPU interrupts the current executing control flow and branches into an IRQ handler function. The IRQ handler function must not block, as this might freeze the system. If an IRQ handler needs to access some resource which is currently in use by some thread (or some other IRQ handler), it cannot wait for the resource to be released. Therefore, every OS needs some mechanism to *delay* the execution of the interrupt code, or at least of those parts accessing the ressource, until the resource is available.

Delay Mechanisms

The most simple way to enforce delayed execution is using *hard synchronization*, which, however may result in high latency and lost interrupts. For this reason, most operating systems follow a more sophisticated approach and implement the delayed execution by some *software mechanism*. The following describes the approaches used by the analyzed systems:

hard synchronization This approach is, because of its simplicity, often used on small μ-controller OS that execute only very few tasks. The idea is to delay the propagation of the interrupt signal by disabling the IRQ line. Most interrupt controllers are able to hold a signaled but disabled interrupt until the IRQ line is reenabled again. However, if interrupts are disabled too long or too often, latency goes up and IRQ signals might be lost.

prologue/epilogues This approach is used by Linux [2, 14], Windows[15], PURE[13] and many other operating systems. The general idea is to explicitly divide the code to be executed in case of an interrupt into a critical and an uncritical part. The critical part, called *prologue*, is executed with low latency at interrupt level. It should perform only the most time-critical tasks and may only access resources that are protected at interrupt level. Before termination, the prologue may request the delayed execution of the part which is not time-critical by registering one or more *epilogues*[2]. Epilogues are queued until the kernel propagates them for execution, which is (typically) the case after all nested interrupt handlers have terminated and before the scheduler is activated. Epilogues thereby have priority over threads, but are interruptable by prologues if new IRQ signals come in. Threads inside the kernel can temporary disable the propagation of epilogues to access shared resources. In this case, epilogue propagation is delayed until the thread finishes its access.

driver threads This approach is common for μ-kernel OS like L4Ka[8]. The general idea is to lift all code to be executed in case of an interrupt up to the thread level. The kernel itself contains only a generic interrupt handler, which sends

[2]Epilogues correspond to *bottom halves* or *tasklets* on Linux and to *deferred procedure calls (DPCs)* on Windows.

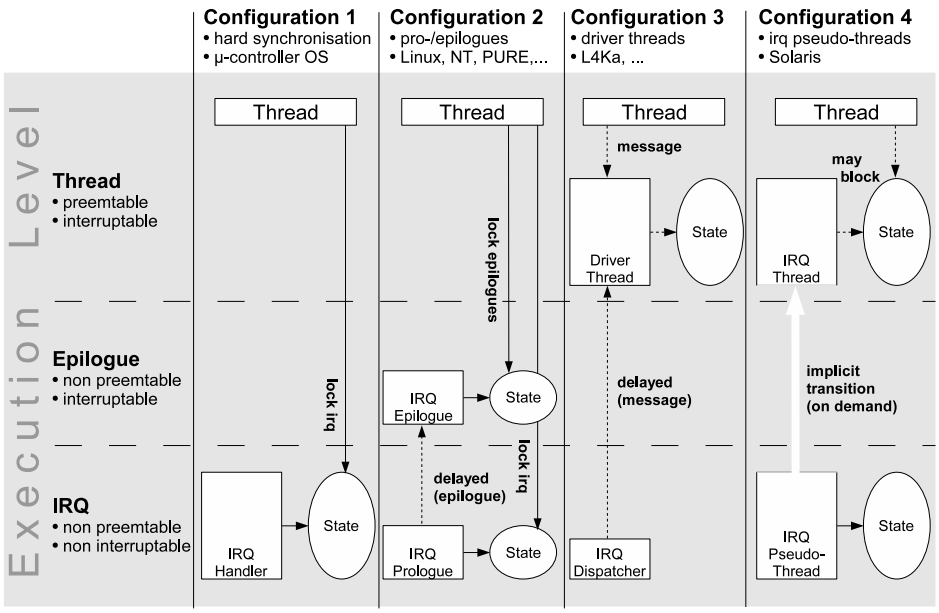

Figure 2: Different configurations of interrupt synchronization in a device driver

a message to the thread registered for an interrupt signal and activates the scheduler[3]. If the thereby activated driver thread has the highest priority, it is then selected for execution and starts the real processing of the interrupt request. Because the code is executed inside a thread, it may even block on other threads (e.g. use synchronous IPCs). Interrupt synchronization is thereby mapped to ordinary thread synchronization, no special mechanism is required.

IRQ pseudo-threads This very sophisticated approach is used by Solaris[6]. The general idea is, again, to map interrupt synchronization to thread synchronization and thereby avoid the need for an extra interrupt synchronization mechanism. However, instead of sending a message to a waiting thread and activating the scheduler in case of an interrupt, Solaris directly switches to a special pre-allocated pseudo-thread. The pseudo-thread owns a complete thread context (instruction pointer, stack), but is not a deschedulable entity as it is still running on interrupt level while executing the handler code. Only if the control flow is required to block (e.g. because of waiting for a locked mutex), the kernel *transparently* lifts up the pseudo-thread to become a real schedulable (but blocked) thread entity, ends the interrupt, and activates the scheduler. Henceforth, the interrupt handler code is executed on thread level until it terminates.

Commonalities and Differences

The various approaches used for interrupt synchronization result in different *execution models* for interrupt code. They also lead to different *programming models* for the developer of device drivers, in which interrupt handling typically takes place. The feature diagram in Figure 1 depicts these two models as main dimensions of commonality and difference in the domain of IRQ synchronization.

[3]The message is actually an IPC sent to a usermode thread in another address space (e.g. of a device driver process). However, this is not relevant here, as it is part of the isolation and interaction properties.

In the *execution model dimension*, one can distinguish approaches that execute the complete IRQ handler on one synchronization level from those, where the interrupt code is spread over two different synchronization levels. The simple *hard synchronization*, as well as the L4Ka *Driver threads* belong to the first category, whereas Solaris' *IRQ pseudo-threads* and *prolog/epilogues* belong to the second. In the 2-level approaches, an interrupt control flow always begins execution on interrupt level. It continues delayed execution on either thread level (Solaris) or epilogue level (Windows, Linux, PURE).

In the *programming model dimension*, one can again distinguish between 1-level and 2-level approaches. However, only for *prologue/epilogues* the developer has to split the code explicitly between boths levels, by optionally requesting the execution of epilogues. *Driver threads* and *IRQ pseudo-threads* offer an identical programming model, as the latter performs an automatic transition from interrupt level to thread level on demand. Optionally, the thread-based approaches permit the interrupt control flow to block on other threads.

4.2 Generalization

The variability in the execution model is desired, as it corresponds to the different implementation of the architecural property, which in turn lead to the variability regarding non-functional properties. The variability in the programming model, however, has to be generalized for the development of architecture-neutral components. The goal is to be able to configure the execution model *without* having to buy another programming model. Figure 2 shows the possible configurations, as well as the resulting structure, of a device driver which is accessed from thread and interrupt level.[4]

Finding a common set of abstractions for the architecure-neutral progamming model requires some reduction to the common de-

[4]The possible configurations also depend on the configuration of other system components, like multi-threading support, which is required by the thread-based configurations.

nominator. If the specific architectures depends on certain assumptions, the most restrictive ones have to make it into the architecture-neutral model. For instance, only the *pro-/epilogues* model enforces an explicit splitting of the driver into two different levels of execution (Figure 2). Nevertheless, it has to become a part of the architecture-neutral model. This is possible, as the enforced explicit splitting is just an additional requirement which does not conflict with other requirements. In other cases, however, it might be necessary to seperate out an abstraction into the architecture-specific model. The optional *may block* feature (Figure 2) is an example for an architecture-specific abstraction that is only available in the thread-based models. A device driver implementation which depends on this feature can not be transformed into the *hard synchronization* or *pro-/epilogues* model.

The aim of the architecture-neutral driver model is to provide enough context information to enable a transformation of the driver code into the architecture-specific model by aspects. This basically means to insert the "right" synchronization primitives at the "right" places. Logically, it is access to state which has to be synchronized. However, this can be mapped to method synchronization, as all state information is considered to be accessible by a restricted set of methods only. In our model, each method of a device driver is placed in one of three different synchronization classes:

synchronized Methods of this class are, depending on the actual configuration, synchronized by some higher-order protocol. If invoked by an interrupt, the actual execution is typically delayed. If invoked by a thread, parallel invocation from interrupt has to be prevented. This is the default class for driver methods. It corresponds to the following execution levels: *IRQ* (Configuration 1), *Epilogue* (Configuration 2), *Thread* (Configuration 3), *IRQ + Thread* (Configuration 4)

blocked For Configuration 2 (*pro-/epilogues*), methods of this class correspond to execution level *IRQ*. For all other configurations they are simply merged into the class *synchronized*.

transparent Methods of this class do not need any synchronization at all, as they perform atomic operations only or use interruption-transparent algorithms. Hence, they can be invoked from any control flow at any time.

If methods from the same class invoke each other, no synchronization is necessary. Synchronization primitives have to be inserted for transitions from thread or interrupt level to *synchronized* or *blocked*. Configuration 2 additionaly requires synchronization of transitions between the classes *synchronized* and *blocked*. The following section describes with a brief example how this can be implemented in AspectC++[16].

4.3 Implemention Sketch

Consider a simple device driver for the system timer, as in the following listing.

```
class Timer {
  ... // state
public:
  void init( long time );
  long get() const;
  void add_event(const EventCallback* cb);

private:
  void tick();
  void process_events();
  friend class irq_dispatcher;

  void handler() {
    tick();
    process_events();
  }

  // what belongs to which synchronization class
  pointcut int_handler()  = "% Timer::handler()";
  pointcut blocking()     = "% Timer::init(...)"
                            || "% Timer::tick()";
  pointcut transparent()  = "% Timer::get(...)const";
  pointcut synchronized() = "% Timer::%(...)"
   && !int_handler() && !blocking() && !transparent();
};
```

The driver offers a public interface for threads to set and get the system time (`init()`, `get()`) and to be notified at a certain time (`add_event()`). The private `handler()` method is invoked by the low-level interrupt dispatcher in case of an interrupt signal. It advances the system time and notifies all registered events that have expired. The `get()` method performs an atomic read operation and is therefore considered to be *transparent*, while `init()` is assigned to *blocked*, as it performs a non-atomic write operation on the internal timer value, which is also modified by `tick()`. All other methods (except `handler()`) are assigned to *synchronized*.

The `Timer` driver code is architecture-neutral regarding the interrupt synchronization property. The following aspect is used to transform it to use *hard synchronization*:

```
aspect Configuration1 {
  pointcut block() = Timer::synchronized()
           || Timer::blocking();
  advice call( block() && !within( block()
         || Timer::int_handler() ) : around() {
    disable_int();
    tjp->proceed();
    enable_int();
  }
};
```

The aspect for the *prologue/epilogues* model has to give some extra advice for the delayed execution of epilogues and for the potential transitions between the synchronization classes *blocked* and *synchronized*:

```
aspect Configuration2 {
  pointcut block() = Timer::blocking();
  pointcut delay() = Timer::synchronized();
  advice call( delay() )
         && !within( "% Timer::%(...)" ) : around() {
    lock_epilogues();
    tjp->proceed();
    leave_epilogues();
  }
  advice call( block() ) && !within( block()
         || Timer::int_handler() ) : around() {
    disable_int();
    tjp->proceed();
    enable_int();
  }
  advice call( Timer::synchronized() ) && !within(
         Timer::synchronized() ) && cflow(
         execution( Timer::int_handler())): around(){
     add_epilog (tjp->action ());
  }
};
```

For the *driver threads* model no advice has to be given, as all necessary synchronization is implicitly done by the message-based interaction used to invoke methods. As discussed in section 2, *inter-*

action is another architectural property which is not in the scope of this paper.

```
aspect Configuration3 {
  // nothing to do!
};
```

Finally, the necessary synchronization primitives for the *IRQ pseudo-threads* model are applied by this aspect:

```
aspect Configuration4 {
  pointcut exclude() = Timer::synchronized()
           || Timer::blocking();
  advice call( exclude() )
         && !within( exclude() ) : around() {
    lock_mutex();
    tjp->proceed ();
    unlock_mutex();
  }
};
```

5. SUMMARY AND CONCLUSIONS

Many non-functional properties of software systems are emergent and, thus, need to be addressed on a global scope by some sort of "holistic aspects". Architecture can be understood as such a "holistic aspect", as it has a noticeable impact on many non-functional properties. Architectural decisions do crosscut significant parts of the actual implementation of every component. This gives the opportunity to address them by aspects and thereby configure non-functional properties *indirectly* by the means of configurable architectures. On the other hand, software components have to be specifically designed with architectural configurability in mind, which can be a quite complicated task. The work on CiAO is clearly at an too early stage to evaluate the benefits of using aspects for this purpose on the large scale. From what we did so far we can, however, draw some preliminary conclusions:

The devil is in the details While it is broadly accepted that aspects are feasible for encapsulating crosscutting concerns, their applicability for the non-trivial case always seems to be a question on its own. For the (relatively complex) interaction patterns found in operating systems this is specifically true, as subtle implementation details can have an enormous impact on correctness or performance. Hence, an in-depth analysis of the technical details is unavoidable for a reliable evaluation if and how AOP is beneficial for the encapsulation of certain architectural properties.

Applicability to other OS concerns Interrupt synchronization is just one of the properties that "make" the architecture of an operating system. It is a natural starting point for a bottom-up process, which is required to tackle the inherent interdependencies between architectural properties. Architecture-neutral models for other fundamental properties, including isolation and interaction, have to be developed as well. This will probably be again a matter of very specific details. However, we are optimistic that the general approach as described in section 3 works for these other properties as well.

Additional Requirements to AspectC++ The interrupt synchronization example demonstrates that AspectC++ already provides the ability to perform quite complex context-dependent transformations. Nevertheless it is rather likely that we have to carefully extend AspectC++ to address other architectural properties. To implement, for instance, component interaction via IPCs as an aspect, one has to be able to give advice that forwards the whole calling context to a thread running in another address space.[5]

6. RELATED WORK

There is some related work in the domain of applying AOP techniques to operating systems. Coady et all demonstrated the encapsulation of an architectural OS property (prefetching) by an aspect in the FreeBSD kernel[3]. However, the focus of this work was not on configuration of architectural properties. The THINK framework demonstrates, how operating systems with different interaction schemes can be constructed from architecture-neutral components by using special "binding components"[5]. THINK does not use AOP, it is based on COM interfaces and does not support the configuration of other architectural properties. Related work that suggest to exploit aspects for specifying synchronization constraints is to numerous to list, however, the work of Lopes[12] probably had the most noticeable impact to this topic.

7. REFERENCES

[1] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

[3] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE '01*, 2001.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. AW, May 2000.

[5] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *2002 USENIX TC*, pages 73–86. USENIX, June 2002.

[6] S. Kleiman and J. Eykholt. Interrupts as threads. *ACM OSR*, 29(2):21–26, Apr. 1995.

[7] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *ACM OSR*, 13(2):3–19, Apr. 1979.

[8] J. Liedtke. On μ-kernel construction. In *15th ACM Symp. on OS Principles (SOSP '95)*. ACM, Dec. 1995.

[9] A. Lister and R. Eager. *Fundamentals of Operating Systems*. Macmillian, 4 edition, 1988.

[10] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. On the design and development of a customizable embedded operating system. In *SRDS Dependable Embedded Systems (SRDS-DES '04)*, Oct. 2004.

[11] D. Lohmann and O. Spinczyk. Architecture-Neutral Operating System Components. *19th ACM Symp. on OS Principles (SOSP'03)*, Oct. 2003. WiP session.

[12] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[13] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the PURE operating system family. In *5th ECOOP W'shop on Object Orientation and Operating Systems*, pages 49–54, Malaga, Spain, June 2002.

[14] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2001.

[15] D. A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. MS Press, 3 edition, 2000.

[16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *40th Int. Conf. on Technology of OO Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, Feb. 2002.

[17] O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS Eur. W'shop*, pages 188–192, Leuven, Belgium, Sept. 2004. ACM.

[5] Within the same address space this is already possible in AspectC++ by using so-called *action objects*[17, 16].

Software security patches

Audit, deployment and hot update

Nicolas Loriant, Marc Ségura-Devillechaise, Jean-Marc Menaud

Obasco Group
École des Mines de Nantes, INRIA
4 rue Alfred Kastler
44307 Nantes, France
nloriant,msegura,jmenaud@emn.fr

ABSTRACT

Due to its ever growing complexity, software is and will probably never be 100% bug-free and secure. Therefore in most cases, software companies publish updates regularly. For the lack of time or care, or maybe because stopping an application is annoying, such updates are rarely, if ever, deployed on users' machines.

We propose an integrated tool allowing system administrators to deploy critical security updates on the fly on applications running remotely and without the intervention of the end-user. Our approach is based on Arachne, an aspect weaving system that dynamically rewrites binary code. Hence applications are still running while they are updated. Our second tool Minerve integrates Arachne within the standard updating process: Minerve takes a patch produced by `diff`, a tool that lists textual differences between two versions of a file, and eventually builds a dynamic patch that can later be woven to update the application on the fly. In addition, by translating patches into aspects and thus generating a more abstract presentation of the changes, Minerve eases auditing tasks.

1. INTRODUCTION

Despite the availability of correcting patches, in 2003, 80% of computer attacks exploited already published security vulnerabilities [3]. Sasser for example is not an exception - the patch preventing its propagation was available two weeks before it spread all over the world. Thus, most threats could be avoided by strict tracking of security bulletins and quick updating of security vulnerabilities. System administrators can not achieve these tasks without adequate tools. Indeed, reading the 5500 security alerts annually published by the CERT/CC (assuming 5 minutes per bulletin) would require about 13 weeks of work. If only one percent of the reported vulnerabilities were relevant, if the computer network is composed of one hundred machines and if updating one machine takes about an hour, deploying patches would require 157 weeks per year [2]. And this evaluation neglects the time spend in negotiations with end users to stop their applications during updates.

In this paper, we propose a semi automatic approach to deploy security updates. Its goal is to reduce the required time while still allowing system administrators to protect their network efficiently. Our framework is based on two tools, Minerve and Arachne [13]. The first reduces the time spent to audit and to adapt the patch by translating regular patches into aspect source code. The second is a dynamic weaver that deploys the translated patches on the fly freeing administrators from the hassle of negotiating with users.

This paper is organized as follows: section 2 describes a global view of our framework and shows how it integrates itself in the usual patch deployment process. Sections 3 and 4 present Minerve and Arachne respectively. Section 5 summarizes our experimental results and presents a complete example. Sections 6 and 7 discuss benefits of AOP for dynamic patching and the future work. Section 8 concludes.

2. THE FRAMEWORK

Within the open source community, security holes are corrected through the distribution of patches. A patch is produced with the `diff` tool [6], it traces the differences between the source of the old vulnerable version of the application and the source of the patched one. Hence upon a patch publication, administrators are left with no option but recompiling and redeploying the application.

Redeploying a software is very expensive with respect to time and resource consumption. First, the system administrator has to review the patch to check whether it can be trusted or not. This review is difficult as patches are not meant to be read and solely composed of the lines of source code that are different between the vulnerable version and the version of the patch. While patches stress the differences between two versions, they do not help administrators to understand the impact of the changes on the application. Secondly, patches are effective only once the application has been recompiled, redeployed and relaunched. But stopping or even suspending an application is often uncomfortable or simply impossible. Small companies running their own e-commerce site can not afford the additional costs a proper fault tolerant system forgiving temporarily unavailability of a single machine. In a roaming or mobile context, it is hard to believe that even a fault tolerant approach would ever be a solution.

The framework we propose in this paper aims at both reducing the time spent in administrative tasks and decreasing

the resources required to update an application on the fly. It is worth noticing that fault tolerant approaches meet the second objective but not the first. Our approach is based on two tools: Minerve and Arachne. Minerve is a patch transformer. Its input is a `diff`-like patch. Minerve outputs a series of aspects written in the Arachne aspect language [4]. The use of an aspect language clearly presents the modifications made by the patch. Such a clear presentation decreases the time required to audit the patch. Moreover, prior to the generation of the aspect source code, Minerve checks that the new patch can be deployed dynamically without leading to an incoherent execution .

Once the system administrator has validated the new patch, our second tool, Arachne comes into action. Arachne is a run-time aspect weaver for C applications. Pre-installed on every computer of the network, Arachne dynamically weaves the patch provided by Minerve into the running program. Modifications are injected atomically ensuring the consistency of the running program.

The modifications carried out by the patch are taken into account immediately without stopping the services provided by the program or losing current work for the end-user. Nevertheless, modifications are only made on the running process, thus our framework should be complemented with a usual patching like it can be done with static patch deploying tools [8], in order for the modifications to be permanent.

3. MINERVE, PATCH ANALYSIS AND TRANSFORMATION

Source patches provided by developers are usually generated and deployed using tools like `diff` and `patch`. `diff` simply lists line by line textual differences between the two versions of every source file of a program. Thus this tool does not provide much information about the semantics of the modifications. `patch` does the opposite work by injecting differences listed into the source of the application.

In a static update process, the contents of the patch is partially validated by the compilation process. But this off-line verification does not apply to a dynamic update (on-line). From the original source code, Minerve is in charge of retrieving information about the modifications contained in the patch. This additional information permits to verify the dynamic applicability of the patch, and to produce an expressive dynamic patch that can be validated by the system administrator.

In the rest of this section, we will present how Minerve extracts, transforms and validates a patch according to the original source code of a program. In order to demonstrate the feasibility of our approach, we reviewed security holes affecting ANSI C applications running under the GNU Linux operating system on an IA32 platform. It is also important to note that Minerve does not verify the static correctness of a patch but only validates its dynamic applicability.

3.1 Modification analysis

Minerve's first task is to classify modifications contained in a patch. As we focus on applications written in C, we enumerated all possible modifications that could be applied at run-time. C is a typed, procedural language with side effects. We distinguish two kinds of types: simple types corresponding to entities that can be manipulated efficiently by the processor, *e.g. int*, and complex types made up of other types. A source patch can modify a program behavior in two ways. First, by modifying the mechanisms it contains (functions). Second, by changing type definitions of the data it manipulates (variables). From a static point of view, this distinction is unnecessary but it is essential to update application on the fly since compiled code of function bodies is usually kept in read only memory while data is not.

3.1.1 Possible modifications

Three kinds of modifications can occur: a patch can add, remove, or replace a function. Minerve treats the addition and the removal of a function as if they were function replacements. Indeed, adding a function in a running application is useless if the patch does not add another function that use it.

Replacing a function f by another function f' can possibly modify the prototype of f. This case can be seen as the addition of a new function f' while modifying all calls to f. Furthermore, when a patch replaces an existing function without changing its signature, the updating process has to guarantee that the original function is not executing at the time it is updated [7, 12]. In order to ensure this condition, we rely on Arachne' mechanisms presented in Section 4.

In order to ensure the coherency of the program, the replacement of a function f by f' must be done atomically. For this we rely on Arachne. Nevertheless this is not sufficient. Indeed to ensure coherency of the application when replacing a function, the new version f' should not read data written directly or indirectly by the execution of f because in certain cases this could lead to an incoherency. In order to ensure this, we chose to examine statically the new function f' to determine if it might use data produced by f. We use an ad hoc source code parser to check that property.

3.1.2 Modifying data's type definition

In this section we distinguish modifications made on basic types from the ones made on complex types.

Two operations have to be executed on a simple type redefinition of a variable. First, updating the value hold by the variable. Second, modifying the code that manipulates the variable. When increasing the capacity of a variable without changing its numerical type (eg: *short* → *long int*), no conversion problem can occur as the new type can always hold the current value. However, diminishing the capacity of a variable (eg: *long int* → *short*), or modifying the numerical type of the variable (eg: *int* → *float*), can only be done if the current value can be contained in the new type definition or if a conversion formula is provided. If it is possible to transfer the current value of the variable, the code manipulating it must be updated too. Indeed assembly opcodes, registers, processor flags and exceptions triggered may vary according to the size and type of operands to be manipulated. Consequently, this modification may affect surrounding instructions. In order to handle this situation generally, our tool recompiles the entire function being modified. In certain cases this might not be sufficient. Indeed, as

specified in the System V Application Binary Interface [14], the responsibility of saving floating point registers belongs to the calling function, and thus modifying a variable from type *int* to *float* requires modifying the code of the calling function. Nevertheless, this case is handled by Arachne and thus modifications are always limited to the function that accesses the modified variable.

A program's behavior can also be altered by modifying a variable of a complex type definition. In this paper, we only present the addition of a new field in a structured type as it is relevant to modifications that can be made (addition, deletion, replacement). At the processor level, alteration of a structured type can modify alignment constraints on variables of that type. Some assembly instructions can have a different behavior and even not work at all when the operand they manipulate does not respect these constraints [9]. Thus, our updating process does not modify the base program code which continues to manipulate the original definition. Only the code added is aware of the new field and thus is translated to access it via a hash table indexed with the original variable's address. This solution allows us to ensure coherency of the base program without stopping it, nor needing to update all the variable at once.

3.2 Patch auditing

There are two reasons for auditing patches: to ensure that the vulnerability is really corrected, and to check that the code added by the patch does not include a new vulnerability. It can also be necessary to adapt the patch to a specific security policy. As an example, many specialists advise inserting an alarm associated with an Intrusion Detection System (IDS) in addition to the patch, in order to detect exploitation attempts [11]. The use of Arachne's aspect language make it easy for the system administrator to the add code triggering the IDS inside the patch.

Contrary to `diff` that gives very little information on the modifications contained in a patch and that presents them in a very low-level line-by-line manner, Minerve translates the patch into Arachne's aspect language. This more abstract representation of the modifications lists all functions, variables and type definitions that have been altered and their respective new version and thus eases the comprehension.

Arachne's aspect language offers an efficient join point model and high level constructs that allow to easily benefit from aspect-oriented programming [4]. Nevertheless, the dynamic patching of security violations does not make full usage of the higher level constructs.

4. ARACHNE, DYNAMIC PATCH INJECTION

In this Section, we present tools provided by Arachne that allow compiling and injecting patches into a running application.

4.1 Compilation and deployment

Arachne provides an aspect compiler and a run-time weaver. The aspect compiler, `acc`, transforms aspect source code into a native shared library. The run-time weaver, `weave`, injects this library inside the application. In addition to the verifications that are made by Minerve, `acc` ensures the dynamic patch is syntactically correct. At injection time, weave checks that references made to the application by the patch exist, partially ensuring that the patch corresponds to the right application version. Even if the patch comprises multiples aspects or rewriting points, the rewriting strategies of Arachne ensure the coherency of the application during the injection. Moreover, Arachne guarantes that on failure of the weaving process, the application remains unchanged.

4.2 Arachne inside

Arachne's weaver is used via the `weave` command, it rewrites application binary code at run-time in order to inject the aspects. This section focus on the mechanisms provided by Arachne, used by Minerve. A complete description of all of Arachne's mechanisms is available in previous publication [13]. On a Pentium processor a function call is translated into binary code as a single instruction, `call`, with an address as operand. Arachne disassembles binary code in order to find `call`*s*. To associate a function name with an address, Arachne parses the application symbol table that has been produced by the C compiler. At weaving time, Arachne loads the aspect library in to the memory of the application and rewrites previously found `call`*s* to redirect the control path to the appropriate functions in the library. A similar technique is used to rewrite accesses to variables in the heap.

Some considerations are problematic during the process we just described. Indeed, the process must guarantee the coherency of the application during the weaving. Basically, no added code should be executed before every aspect is fully woven into the program. Moreover Arachne must overcome memory isolation mechanisms and consider performance issues. Arachne solves the coherency issue by the use of locks and dynamically generated hooks that save and restore the program state. To circumvent the memory isolation, Arachne uses debugging support to insert itself inside the process's memory space.

5. EVALUATION

In this section, we have evaluate of our framework. We have applied our framework to all security advisories concerning open source C softwares published by the CERT since 2002. After a brief presentation of the CERT, we present our results over the whole test suite, and one complete example.

5.1 Test suite

CERT stands for "Computer Emergency Response Team / Coordination Center". It was created in November 1988 after the appearance of the Morris worm. It aims at training and warning about internet computing security. Its age and its independence from software editors make the CERT an international reference in security. Since 1988, it has collected an accurate database of vulnerabilities reported in softwares. We made our evaluation over all major vulnerabilities (CERT Advisories) reported in open source software since 2002. This period counts a total of 67 advisories. 30% of these concern Microsoft products, 20% other proprietary products, 10% concern embedded softwares and finally 40% open source softwares. In these last 14 advisories, we neglected 2 because they were affecting unavailable versions of the software.

In the considered vulnerabilities, about half of them are buffer overflows, 20% are format string bugs, 10% are double free bugs, 5% are integer overflows, and finally the remaining ones are combinations of the 4 previously cited. All these bugs are mainly based on assertions made by the developers on the input that are not verified at execution. Thus these vulnerabilities can be easily corrected by adding tests on input data. We verified this when auditing the patches provided by the developers for these security advisories. Indeed 90% of the patches contain modifications of function code without changing prototypes and only 10% modify type definitions.

Our experiments show that our approach can be applied successfully to all the security advisories considered.

5.2 Example

In this section, we present a full example from our test suite. The vulnerability it concerns was published in June 2002 under the reference `CA-2002-18` by the CERT. The software affected is the communication server openSSH. An integer overflow might be exploited in authentication functions of the SSH2 protocol in versions from 2.3.1p1 to 3.3. It might allow the execution of arbitrary code on the targeted host.

5.2.1 The source patch

The source patch provided by the openSSH development team modifies two functions of `sshd`: `input_userauth_info_response` and `input_userauth_info_response_pam`. The modifications only add tests on the parameter `nresp`. When the parameter is invalid, the patch calls the function `fatal` to terminate the program. As shown in listing 1, the patch does not offer much information about the semantic of the modification and useful information can only be obtained by looking at the program source code.

5.2.2 The dynamic patch

Minerve transforms the source patch into a dynamic patch that essentially contains a collection of aspects that are meant to replace vulnerable functions by their safe version. Minerve names the new and safe functions by adding the suffix "`_new`" to their original name. As a function can be called in `sshd` (as for any application) via a direct call or via the use of a function pointer, it is necessary to produce two aspects in order to replace any kind of call to the replaced function. The listing 2 shows this two aspects for the original function `input_userauth_info_response`.

The pointcut of the aspect `ReplaceFunctionCall` traps every call made with a constant address to the old function (line 2). The advice call the new version with the same parameters (line 3). Thus this aspect replaces every direct call to the function with the call to its safe version. In a similar way the second aspect, `Replacepointer` (line 5) traps every read access to the address of `input_userauth_info_response` (line 6) and returns in place of it the address of the safe version (line 7). Thus any future indirect call to `input_userauth_info_response` will be replaced by its new version.

As shown in listing 2, patches produced by Minerve ease the audit by describing modifications of the application in a language close to C. Our experiments show that the framework offers a significant reduction of the time spend to deploy patches. Indeed, excluding network transfer time, Arachne updates an application in less than $250\mu s$. And because updates are made in parallel on the entire network, the time for applying the update is independent of the network size.

6. DISCUSSION AND RELATED WORK

We intended this work in order to evaluate whether Aspect Oriented Programming in suitable for dynamic patching. It is legitimate to wonder what is the benefit of AOP in this field. We already pointed that during our experiments, Minerve did not make full use of Arachne's aspect language constructs. There are two reasons for this. First, because it is a complex task to analyze an application source code in order to infer high level rules about the modifications made by patches. Second, for most part, security patches are written in emergency. Then modifications are often limited to a single test where the vulnerability might appear, thus making patches less crosscutting. Nevertheless, our experiment on larger patches show an interesting potential for AOP.

To our knowledge, no other work provides both coherency analysis and dynamic updating. Previous work has focused on determining when in the execution flow an update may be applied safely [7], without the ability to guarantee that such a moment is reachable. In contrast to this, our approach tries to determine the applicability of a patch independently of the execution.

Dynamic patching also benefits from AOP because aspects are far more comprehensive than patches, indeed, reasoning about the program execution is easier than on its code. Also, AOP is more appropriate for dynamic patching than binary rewriting APIs like Dyninst [1] or Vulcan [5]. First, aspect code is far more intuitive to read than a program. Second, when using binary rewriting APIs, the developer is responsible to ensure the program won't behave abnormally whereas our dynamic weaver, Arachne, ensures that modifications are made atomically.

7. FUTURE WORK

Our analysis of dynamic applicability of patches are for now limited to simple cases, mainly due to source code parsing. It is necessary to base our analysis on higher level representation of source code in order to avoid this limitation. Thus, we plan to use the type propagation tool Lackwit [10].

For technical reasons our framework is limited to open source C applications running on Linux, IA-32 architecture. Nevertheless, as compiled aspects are independent of the woven program, there is no restrictions for software editors to diffuse Arachne compiled patches to be applied on binary distributed programs. This would permit dynamic patching without needing the application's source code. We also plan to adapt our framework to integrate other languages and platforms and to apply it to kernel code in the near future.

8. CONCLUSION

In this paper we have presented a novel approach for security updates based on a framework for dynamic software updates. Our first tool Minerve determines whether a patch can be deployed on the fly. The use of an aspect language

```
    authctxt->postponed = 0;    /* reset */
    nresp = packet_get_int();
+   if (nresp > 100)
+       fatal("input_userauth_info_response: nresp too big %u", nresp);
    if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)
```

Listing 1: The source patch correcting the vulnerability CA-2002-18

```
ReplaceFunctionCall :
    call(void input_userauth_info_response(int, u_int32_t, void*)) && args(type, seq, ctxt)
         then input_userauth_info_response_new(type, seq, ctxt);

ReplacePointer :
    readGlobal(void* (input_userauth_info_response)(int, u_int32_t, void*))
        then return &input_userauth_info_response_new;
```

Listing 2: The aspect patch correcting the vulnerability CA-2002-18

allows administrators to validate more rapidly patches. Our second tool, Arachne applies patches dynamically without data loses and makes security updates effective immediately. Moreover our framework can easily be integrated in the static update process as it accepts standard patches published by software developers.

Despite the potential existence of patches that might not be translated in dynamically injectable aspects, our framework is efficient enough to be applied successfully on all the security advisories published by the CERT since 2002.

9. ACKNOWLEDGEMENTS

This work is supported by a regional grant from the Pays de la Loire, France. The authors would like to thank Thomas Fritz for his valuable comments.

10. REFERENCES

[1] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[2] CERT/CC. Incident and vulnerability trends, May 2003. http://www.cert.org/present/cert-overview-trends/.

[3] Devoteam. European study on computer network security. Technical report, XP Conseil, 2004. http://www.devoteam.com.

[4] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system-level applications with Arachne. In *Proceedings of the 4th international conference on Aspect-oriented software development*, Mar. 2005. to appear.

[5] A. Edwards, A. Srivastava, and H. Vo. Vulcan. Technical Report MSR-TR-99-76, Microsoft Research (MSR), Jan. 2001.

[6] GNU Project. Diffutils. http://www.gnu.org/software/diffutils/diffutils.html.

[7] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb. 1996.

[8] HP. NOVADIGM: Software management, automated change management, 2004. http://www2.novadigm.com/products/patchmanager.asp.

[9] Intel. *IA-32 Intel Architecture Software Developer Manual*, 2001. Instruction Set Reference Manual. ftp://download.intel.com/design/Pentium4/manuals/25366613.pdf.

[10] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 338–348. ACM Press, 1997.

[11] B. Schneier. *Secrets and Lies : Digital Security in a Networked World*. Wiley, Aug. 2000.

[12] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2), Mar. 1993.

[13] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, Boston, Massachusetts, USA, Mar. 2003. ACM Press.

[14] U. S. L. System Unix. *System Application Binary Interface and Intel 386 Architecture Processor Supplement*. Prentice Hall Trade, 1994.

WEAVING ASPECTS TO SUPPORT HIGH RELIABLE SYSTEMS: DEVELOPPING A BLOOD PLASMA ANALYSIS AUTOMATON

Valérie MONFORT, Muhammad Usman BHATTI, Assia AIT ALI SLIMANE

Université Paris 1 Panthéon Sorbonne
Centre de Recherche en Informatique
90 rue de Tolbiac
75013 Paris, France
Valerie.monfort@univ-paris1.fr; muhammad.bhatti@malix.univ-paris1.fr; assia.ait-ali-slimane@malix.univ-paris1.fr;

Abstract

Among current architectures, Service Oriented Architectures aim to easily develop more adaptable Information Systems. Most often, Web Service is the fitted technical solution which provides the required loose coupling to achieve such architectures. However there is still much to be done in order to obtain a genuinely flawless Web Service, and current market implementations still do not provide adaptable Web Service behavior depending on the service contract. Therefore, our approach considers Aspect Oriented Programming (AOP) as a new design solution for Web Services. Based on both Web Service Description Language (WSDL) and Policy contracts, this solution aims to allow better flexibility on both the client and server side. In this paper, we aim to develop an automaton to analyze blood plasma; Web Services are used for software part of the automaton. Faced by the lacks of Web Services, we propose a concrete solution based on aspects.

Key Words

Web Services, Interoperability, Security, Aspects

1. INTRODUCTION

Companies have to communicate with distant IS, such as, suppliers, partners … they use to exchange data through workflows in heterogeneous contexts. The company for which we are working aims to develop automatons to analyze blood plasma, which means patient data information has to be highly reliable and correct. We are involved in the architecture definition and implementation of one of its automaton. In order to support consequent evolution and successive reutilization of the machines, this company decided to define and promote flexible and adaptable architecture according to the new emerging requirements. In this context, Web Service technology is asked to handle the same features as components from the DCOM, J2EE or CORBA worlds already handle. These features, such as security, reliability, or transactional mechanisms, can be considered as non-functional aspects. Obviously these aspects are crucial for business purposes and one cannot build any genuine Information System (IS) without consideration for them. However, managing these aspects is likely to involve a great loss in interoperability and flexibility. This effect has already been experienced with various middleware technologies. Mostly, middleware delegates these tasks to the underlying platform, hiding these advanced mechanisms from the developer, and then establishing a solid bond between the application and the platform. Thus, a great deal of work is required to make Web Service fully adapted for the industry. Especially, mechanisms in charge of handling non-functional tasks must preserve seamless interoperability.

In this article, we introduce the industrial context and technical choices for applications integration with Web Services. From the limitations of this solution, we propose a solution based on aspects and we explain how to apply this solution with a concrete implementation.

2. INDUSTRIAL CONTEXT

2.1. Description of the Automaton

Figure 1 shows some high level functional domains supported by the automaton, including software and firmware: arrows represent the communication flow. Application displays specific Human Machine Interface (HMI) according to profile and maturity level of the user. Access is allowed or denied according to user profile and protected from unauthenticated usage. However, it is possible to ask for analysis and to receive result with different media as mobile phone, PDA, and Web with specific passwords reserved for laboratory managers and doctors. Non functional services, such as security, reliability, persistency, archiving, multi tasking, and supervision, have to be

defined and implemented. Automaton supports some business functions as patient data management and used consumables for plasma blood analysis. Using automaton involves data generation that is analyzed for preventive maintenance. Communication between software and firmware with specific protocol is implemented by using CAN bus [14]. Automaton allows handling of tubes containing the blood plasma. Automaton arms take the blood plasma and use reagents to test coagulation. With this system, blood disorders, such as hemophilia, can be detected.

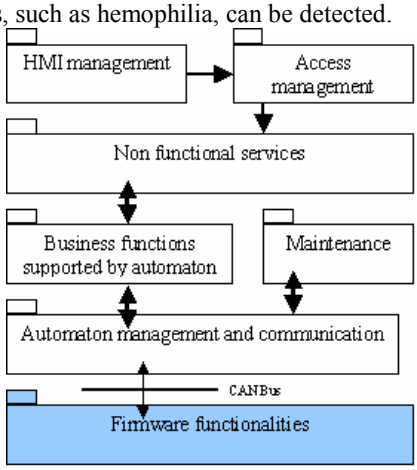

Figure 1: Functional Architecture

Communications between domains can be supported by Web services. Moreover, it might be necessary to exchange patient data and results between different hospitals or other Information Systems (IS). Infrastructures might be based on heterogeneous technologies. For instance, a laboratory uses IBM J2EE technologies and hospital uses Microsoft technologies. Thus, we invoke Web Services developed by different platforms supporting different technologies.

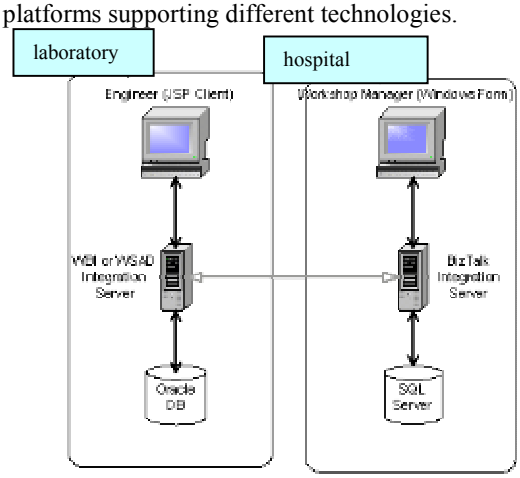

Figure 2: Technical Architecture

2.2. Using Web Services

DCOM, J2EE or CORBA don't scale to the Internet: their reliance on tightly coupling the consumer of the service to the service itself implies a homogeneous infrastructure. Web services use industry standard protocols to guaranty interoperability between IS. In order to provide the missing business features required to leverage Web Service technology, a first set of tools has emerged. Built on top of .NET and J2EE platforms, Microsoft and IBM have implemented their own toolkits based on the Web Service specifications. Web Services Enhancements for Microsoft .NET (WSE) [7] is a supported add-on to the Microsoft .NET framework providing developers the latest advanced Web Services capabilities such as security, security policy, addressing, routing, and attachments.

The Emerging Technologies Toolkit (ETTK) [8] is a software development kit for designing, developing, and executing emerging autonomic and Web Service technologies. It provides an environment in which to run emerging technology examples that showcase recently announced specifications and prototypes from IBM's emerging technology development and research teams. Based on Axis [9], ETTK processes messages through handlers in chain. One particular chain enables developers to insert their own message managers, such as security handlers. A MessageContext object is included in outgoing messages and is extracted from incoming messages. The handlers in charge of the transformations are specified in a Web Service Deployment Descriptor (WSDD) file. These toolkits look quite similar in the sense that they operate and compute messages. SOAP Engines are composed of filters (SOAP handlers) whose main role is to perform transformations on the SOAP message [6], depending on parameters included in the header. The SOAP headers are in charge of delivering the context of the message (authentication tokens, reliable messaging properties, etc.).

Our technical approach to current Web Service solutions enabled us to notice two major facts which are at the root of Web Service's lack of flexibility. First, there is no dynamic mechanism to bind policies and Web Service handlers. Secondly, there is no clean separation of concerns [5] between the functional and the non-functional code as well as between SOAP logic and non-functional logic within handlers, as figure 3 shows. Once the client or service is coded and the handlers are deployed, the Web Service cannot handle new features and, because the different logics are tangled up, it is not easy for another developer to reuse the application in a different context.

3. USING ASPECTS

3.1. Discovering aspects

Consequently, an appropriate way to deal with these crosscutting concerns [2] would be to use different units of modularization to encapsulate these logics [4]. Moreover, if these units of modularization could be managed by a dynamic mechanism, then the whole system would be able to dynamically reconfigure itself depending on the policies [1].

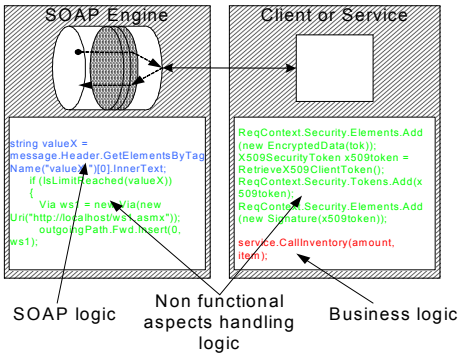

Figure 3: Tangled Logic within SOAP Services

These requirements lead us to consider Aspects Oriented Programming (AOP), in the first step, as an answer to Web Services reusability issues [3]. AOP is one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality. It allows the separation of crosscutting concerns into single units called aspects, which are modular units of crosscutting implementation. With AOP, each aspect is expressed in a separate and natural form, and can be dynamically combined together by a weaver. As a result, AOP widely contributes to increased reusability of the code and provides mechanisms to dynamically weave aspects [4].

Considering Web Services, non-functional aspects handling logic should be encapsulated within multiple aspects. Each aspect would be in charge of certain features, such as security, and would deal directly with well-defined objects like Kerberos tokens (security) or Shipping forms (reliable messaging). Pushing the non functional handling logic inside aspects means that handler's role has to be redefined, as they will only contain SOAP logic then. The idea is to replace the multiple specific handlers, which used to process SOAP messages depending on their own implementations, by a global handler whose role will be restrained to extracting non-functional data contained in incoming messages, and pushing it inside outgoing messages.

3.2. Weaving Process

At this point, we need to define where, when and how the aspects should be weaved. Let us answer these questions by considering the different opportunities for each of them. First, aspects could be weaved to the global handler, to the stub or to the service implementation itself. In fact, considering the global message path and process, choosing any of these entities does not really influence the mechanism. However, we found it more convenient to weave aspects to the stub since it provides a natural meta object to focus on the service itself [15]. Secondly, there are multiple choices for when to weave aspects. It could occur during compile time, deployment time, load time or run time. If the weaving were to happen at compile time or deployment time, it would not be possible to handle policy changes dynamically. Conversely, there is no need to weave aspects at runtime since the policy document will most likely not be changed after the service starts running. Thus, the ideal solution is to weave aspects when the service is loaded to enable one single yet sufficient analysis of the policies document for each new instance [11]. Thirdly, the weaver should be an application capable of reading the policy document, interpreting the policies, selecting the relevant aspects and finally mixing them with the plain stub, as can be seen on figure 4.

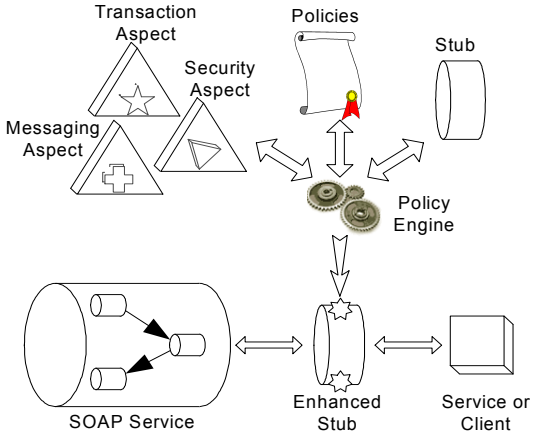

Figure 4. Aspects weaving at load time.

Transmitting non-functional data to aspects weaved to the stub at load time is one possible solution to achieve genuinely flexible Web Services. This mechanism allows Web Services to be reused more easily since each non-functional aspect is detached from both the service implementation and the handler. The Policy Engine inserts these aspects depending on the service

contract requirements [16][7], which means that interoperability is preserved if, for instance, requirements from different clients vary.

We have seen how AOP can help to gain flexibility through a cleaner separation of logics and which mechanism can help to provide policy awareness among Web Services. We shall now present our concrete implementation of these concepts.

4. A CONCRETE SOLUTION

4.1. Structure of Axis

In our solution, we take advantage of multiple open source solutions already available for Java therefore we modify and assemble them easily. This way, we can start with a ready-to-use platform that we need to complete in order to obtain flexible Web Services. Thus, the Web Server and the SOAP Engine are constituted by the famous open source duo Tomcat-Axis. Basically, Axis plugs into the Tomcat Servlet Engine, meaning that it can be considered the same as any other Web Application. Web Services are hosted and managed by Axis in a transparent way for Tomcat as shown in figure 5. Axis is based on the concept of a chained message. The MessageContext object is a wrapper object for the request and the responses message and for contextual information about process, request, response, etc. In figure 5, Request and Response are handlers that manipulate the MessageContext.

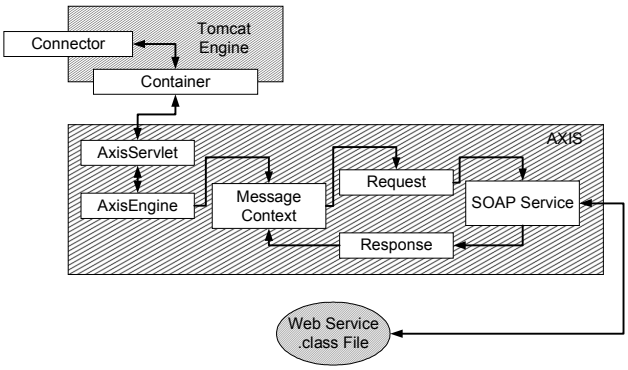

Figure 5. Axis Server-side Architecture.

Since these handlers can easily manipulate this object, it is quite natural to select these handlers to act like basic SOAP logic handler. For instance, if an incoming SOAP header contains data that says the body message is encrypted, then the Request handler needs to decrypt the body. But the genuine non-functional logic is hosted by the aspects, and non-functional data used by these aspects is transmitted by the provider. The provider is another handler that, when invoked, calls the stub corresponding to the service invoked. Once processed and transformed into appropriate objects, these data will be passed to the stub weaved with aspects.

4.2. Stub Bytecode Modifications

Let us now see how aspects are weaved to the stub. First, we need to understand how class loading works in Tomcat. Indeed, if we can modify the bytecode of the stub object when it is loaded into the Java Virtual Machine (JVM), then it will be possible to weave the aspects at load time. Tomcat uses multiple class loaders, which are java objects aiming to load resources (class or jar files). With Java 2, class loaders follow a delegation model, which means that if a class is asked to be loaded by a class loader, then this class loader will first ask its parent class loader to do so. If it cannot load the class, the initial class loader will search inside its own resources. All Tomcat class loaders follow this rule except Web Application class loaders, which are responsible for the loading of each class of the Web Application they are in charge of. Consequently, the idea is to modify the class loader in charge of Axis Web Application so we can reach any Web Service stub anytime it is loaded. To obtain such a class loader, we just need to reuse the code of the Axis regular WebAppClassLoader and specify that Tomcat has to use the ModifiedClassLoader when it loads Axis Web application, via the server.xml configuration file.

```
<Context docBase="C:\axis-1_1\webapps\axis"
path="/axis">

   <Loader loaderClass =
"org.apache.catalina.loader.ModifiedClassLoader"/>

</Context>
```

The next step is to use a tool which allows both introspection and reflection - the former to inspect the stub code when it is loaded and the latter to achieve the weaving of aspects. One particularly convenient answer to these requests is brought by Javassist [1]. Javassist is a class library for enabling structural reflection in Java, which is performed by bytecode transformation at compile time or load time. In order to modify bytecode at load time, Javassist performs structural reflection by translating alterations of structural reflection into equivalent bytecode transformation of the initial class file. After the transformation, the modified class file is loaded into the JVM by a special class loader. To bring this mechanism into our solution, the ModifiedClassLoader must adhere to three rules. First, it must encapsulate a Javassist.ClassPool object, which will act as a

container for objects containing class files to be loaded. These objects derive from the CtClass class which is a convenient handle for dealing with class files (methods or fields adds or renames, etc.). Next, when the ModifiedClassLoader constructor is called, this ClassPool object must be instantiated with the Web Application class path so it can get the scope of the classes it can handle. Finally, whenever a class is to be loaded, the findClassInternal (String name) method is called and must contain the transformation logic which will affect the stub object anytime it is loaded. The code below shows these modifications inside of what used to be the regular WebAppClassLoader class.

```
public class ModifiedClassLoader extends URLClassLoader {
  protected ClassPool pool = null;
  public WebappClassLoader() {
    pool = ClassPool.getDefault();
    pool.insertClassPath(new LoaderClassPath(this));
  ...}
  /* Method called whenever a class is to be loaded */
  protected Class findClassInternal(String name) {
    ResourceEntry entry = findResourceInternal(name, classPath);
    Class clazz = entry.loadedClass;
    /* Javassist loader is invoked to get an easily modifiable CtClass
*/
    CtClass cc = pool.get(name);
    /* Class modifications according to the PolicyEngine */
    if(isStubClass("name"))
      PolicyEngine.Process(cc);
    byte[] b = cc.toBytecode();
    clazz = defineClass(name, b, 0, b.length);
    ...
    return clazz;
  }…
```

4.3. Policy Engine as a Weaver

Eventually, we shall define how the Policy Engine works. As explained before, Policies constitute the Service Contract and, thus, describe the requirements to establish communication. For instance, the <wsse:SecurityToken> element, as shown below, is used to describe which security tokens are required and accepted by a Web service. It can also be used to express which security tokens are included when the service replies.

```
<SecurityToken wsp:Preference="..." wsp:Usage="..." >
  <TokenType>...</TokenType>
  <TokenIssuer>...</TokenIssuer>
  <Claims>...Token type-specific claims...</Claims>
  ... (TokenType-specific details)
</SecurityToken>
```

Once the PolicyEngine.Process(…) method is called, the engine gets a CtClass object containing the code of the stub. Because the name of this class is related to the name of the service itself, it becomes easy for the Policy Engine to locate the Policy contract and thus it can access the policy's requests. The next step for the engine is to fulfill each of these requests by inserting the appropriate aspects within the methods of the stub. This mechanism is almost equivalent for both client and service side. Eventually, the Policy Engine adds fields to the stub so it can obtain and set the non-functional data that the provider manages. At this point, the new "SOAP messages process" is effective and can be used to dynamically handle each of the functional aspects declared in the Policy document. Figure 7 below illustrates the global mechanism at runtime.

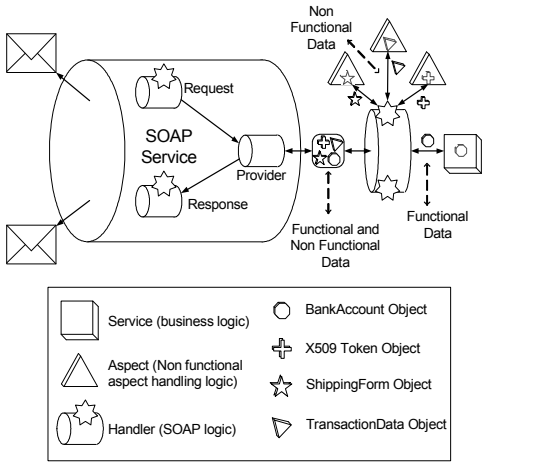

Figure 7. Functional, non-functional and SOAP logics.

5. RELATED WORKS

The Web Service Management Layer (WSML) [10] is an aspect based platform for Web Services allowing a more loose coupling between the client and server sides. The idea of this technology is to transfer the Web Service related code from the client code to this new management layer. The advantages are the dynamic adaptation of the client to find the most fitted Web Service, and it also deals with the non functional properties like Traffic Optimization, Billing Management, Accounting, Security, and Transaction. This work looks very similar to the solution we provide

in the sense that it aims to gather the scattered code in aspects. However, our solution especially aims to target the norms of the Web Service Architecture, which are described in the policies. The Web Services Mediator (WSM) [11] is a middleware layer that sits above standard Web Services technologies such as Simple Object Access Protocol (SOAP) Servers. It aims to decouple an application from its consumed Web Service, and to isolate the application's characteristics (e.g., reliability, scalability, latency etc). Aspect-Oriented Component Engineering (AOCE) [12] has been developed to capture the cross-cutting concerns, such as transaction, co-ordination and security. To achieve this solution, the WSDL grammar has been extended by enriching it with aspect-oriented features so that it becomes better characterized and categorized. However, there are no universally accepted standards of the terminologies and the notations used in AOCE. On the whole, AOCE and our work seem to offer very similar approaches but, although just using the policies to select aspects might be restrictive, our strategy does not require developers to understand any vendor specific standard. The Web Service Description Framework (WSDF) [13] consists of a suite of tools for the semantic annotation and invocation of Web Services, by mixing both Web Service and Semantic Web communities. Instead of establishing a hard wired connection between the client and the service, by specifying the Web Services through addresses, WSDF enables the developer to formally specify a service using rules and ontological terms.

6. CONCLUSION

Service Oriented Architectures require loose coupling to access the services which will most likely be implemented with emerging Web Service technology. Using current SOAP toolkits, we noticed that interoperability between client and Web Service is damaged by non-functional aspects required by businesses (such as security, transaction, reliable messaging, etc). In fact, they require establishing a strong coupling between the service logic, the non-functional handling logic, and the SOAP logic. On top of this, there is no dynamic adaptation mechanism to bind the service contract requirements to the Web Service and client abilities. These facts significantly reduce Web Service flexibility and affect the loose coupling ability offered by Services. The solution that we are providing aims to offer a dynamic mechanism to compute the service contract on the fly, enabling Web Services to become fully aware of the business requirements. The main principle consists of using computational reflection [15] as a means to achieve separation of concerns and dynamic adaptability. Our new SOAP Service design provides a cleaner separation between the multiple logics weaved at load time. After analyzing the policies requirements, a Policy Engine is in charge of selecting the appropriate aspects to handle business mechanism like security, transactions, etc. This mechanism allows Services to gain in loose coupling.

Future works will consist of widening the application scope of this solution and validating the Web Services behavior in concrete Service Oriented Architectures. The main tasks will be to implement a library to handle the multiple WS-* norms and then develop a policies fully compliant Policy Engine.

7. REFERENCES

[1] F. Baligand, V. Monfort "A Pragmatic Use of Contracts and Aspects to gain in Adaptability and Reusability" The 2004 2nd European Workshop on Web Services and Object Orientation, EOOWS'04, ECOOP, June 14-18, 2004, Oslo, Norway

[2] M. N. Bouraqadi-Saâdani, R. Douence, T. Ledoux, O. Motelet, M. Südholt "Status of work on AOP at the OCM group, April 2001" , École des Mines de Nantes, technical report, no. 01/4/INFO, 2001 KW: AOP, execution monitoring, program transformation, interpreter

[3] Kiczales G. et al. "Aspect-Oriented Programming", in Proc of ECOOP'97. LNCS 1241, Spinger-Verlag, 1997

[4] Eric Tanter, Jacque Noyé, Denis Caromel, Pierre Cointe "Partial Behavioral Reflection: Spatial and Temporal Selection of Reification", 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003

[5] O. Barais, L. Duchien, R. Pawlak, "Separation of Concerns in Software Modeling: A Framework for Software Architecture" Transformation, IASTED International Conference on Software Engineering Applications (SEA), IASTED, USA, November 2003.

[6] visit web site http://www.w3.org/TR/SOAP

[7] visit web site http://msdn.microsoft.com/webservices/building/wse/

[8] Visit web site http://www.alphaworks.ibm.com/tech/ettk

[9] Visit web site http://www.axis.com/

[10] Verheecke B., Cibrán M.A., "Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection," to be published in the proceedings of the European Conference on Web Services 2004 (ECOWS'04), Erfurt, Germany, September 2004.

[11] Visit web site http://javaboutique.internet.com/articles/WSApplications/

[12] Singh, S., Grundy, J.C., Hosking, J.G. Developing .NET Web Service-based Applications with Aspect-Oriented Component Engineering , In Proceedings of the Fifth Australasian Workshop on Software and Systems Architectures, Melbourne, Australia, 13-14 April 2004.

[13] A. Eberhart. Towards universal Web Service clients. In B. Hopgood, B. Matthews, and M. Wilson, editors, Proceedings of the Euroweb 2002.

[14] Visit web site http://www.ixxat.de/

[15] Chiba, S., "Load-time Structural Reflection in Java" in Proc. of ECOOP'2000, 2000, SpringerVerlag LNCS 1850

[16] D. Mandrioli, B. Meyer « Applying Design by contract » Interactive Software Engineering Inc editions Prentice Hall

Evaluating an Aspect-Oriented Approach for Production-Testing Software

Jani Pesonen
Nokia Corporation
Tieteenkatu 1
Tampere, Finland
jani.p.pesonen@nokia.com

Mika Katara
Tampere University of Technology
Korkeakoulunkatu 1
Tampere, Finland
mika.katara@tut.fi

Tommi Mikkonen
Tampere University of Technology
Korkeakoulunkatu 1
Tampere, Finland
tommi.mikkonen@tut.fi

ABSTRACT

Aspect-orientation enables an approach where tangled code can be addressed in a modular fashion. However, the design of interworking between object-oriented baseline architecture and aspects attached on top of it is an issue, which has not been solved conclusively. For industrial-scale use, guidelines on what to implement with objects and what with aspects should be derived. In this paper, we introduce a way to reflect the use of aspect-orientation to production testing software of mobile systems. Such piece of infrastructure software is used to smoke test the proper functionality of a manufactured device. The selection of suitable implementation technique is based on variance of devices to be tested, with aspects used as means for increased flexibility.

Keywords

Production testing, variability, aspects

1. INTRODUCTION

Aspect-oriented approaches provide facilities for sophisticated dealing with tangled and cross-cutting issues in programs [2]. With aspects, it is possible to weave new operations into already existing systems, thus creating new behaviors. Moreover, it is possible to override methods, thus manipulating the behaviors that already existed.

With great power comes great responsibility, however. The use of aspect-oriented features should therefore be carefully designed to fit the overall system, and ad-hoc manipulation of behaviors should be avoided especially in industrial-scale systems. This calls for an option to foresee functionalities that will benefit the most from aspect-oriented techniques, and focus the use of aspects to those areas. Unfortunately case studies on the identification of properties that potentially result in tangled or scattered code in a certain problem domain have not been widely available. However, understanding the mapping between the problem domain and its solution domain, which includes both conventional objects as well as aspects, forms a key challenge for industrial-scale use.

In this paper, we address domain-specific identification of types of properties that lend themselves to aspect-oriented methodology. The domain we will use as an example is that of production testing of a family of mobile devices, where common and device specific features form different categories of requirements that can be used as the basis for partitioning between object-oriented and aspect-oriented techniques. The way we approach the problem is that the common parts are included in the object-oriented base implementation, and the more device-specific ones are then woven into that implementation as aspects.

The rest of this paper is structured as follows. Section 2 gives an overview of production testing of mobile devices. The section also introduces a production-testing framework for Symbian OS based mobile devices. Section 3 discusses how we relate the problem domain and its aspect-oriented solution domain in this particular case. Section 4 provides an evaluation of aspect-orientation in this setting, and Section 5 concludes the paper with some final remarks.

2. PRODUCTION TESTING

Production testing is a verification process utilized in the product assembly to measure production line correctness and efficiency. The purpose is to evaluate devices' assembly correctness by gathering information on the sources of faults and statistics on how many errors are generated with certain volumes. In other words, production testing is the process of validating that a piece of manufactured hardware functions correctly. It is not intended to be a test for the full functionality of the device or product line, but a test for correct composition of device's components. With volumes typical to modern mobile terminals, the production testing involves software support that must be increasingly sophisticated, versatile, cost-effective, and adapt to great variety of different products. In software the most successful way of managing such variance is to use product families [1].

2.1 Overview

Individual design of all software for all mobile device configurations results in an overkill for software development.

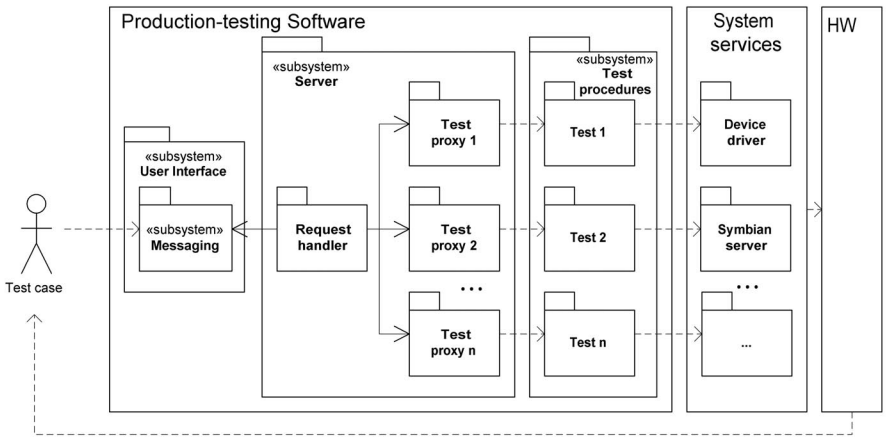

Figure 1: Production-testing framework.

Therefore, product families have been established to ease the development of mobile devices. In such families, implementations are derived by reusing already implemented components, and only product specific variance is handled with product specific additions or modifications. In this paper, we will focus on a product family where Symbian OS [3] is used as the common implementation framework. Symbian OS is an operating system for mobile devices, which includes context-switching kernel, servers that manage devices' resources, and rich middleware for developing applications on top of the operating system.

The structure of the production-testing framework in Symbian environment follows the lines of Figure 1 and consists of three subsystems: user interface, server and test procedures. Test procedure components (Test 1, Test 2, etc.) implement the actual test functionalities and together form the test procedures subsystem. These components form the system's core assets by producing functionality for basic test cases. Furthermore, adding specializations to these components produces different product variants hence dedicating them for certain specific hardware, functionality, or system needs. In other words, the lowest level of abstraction created for production testing purposes is composed of test procedure components that only depend on the operating system and used hardware. As a convenience mechanism for executing the test cases, we have implemented a testing server, which is responsible for invoking and managing the tests. This server subsystem implements the request-handling core and generic parts of the test cases, which are abstract test procedure manifestations as test proxies. Finally, a user interface is provided that can be used for executing the test cases. The user can be a human user or a robot that is able to recognize patterns on the user interface, for instance. The user interface subsystem implements the communication between the user and the production-testing system.

2.2 Variability management

From the viewpoint of production testing, the most important pieces of hardware are Camera, Bluetooth, Display and Keyboard. In addition, also more sophisticated pieces of equipment can be considered, like WLAN for instance. The test software on the target is then composed of components for testing the different hardware and device driver versions, which are specific to actual hardware. When composing the components, one must ensure that concerns related to a certain piece of hardware are taken into account in relevant software components as well. For instance, more advanced versions of the camera hardware and the associated driver, allow higher resolution than the basic ones, which needs to be taken into consideration while testing the particular configuration. Since the different versions can provide different functional and non-functional properties, the testing software must be adapted to the different configurations. For example, the execution of a test case can involve components for testing display version 1.2, Bluetooth version 2.1 and keyboard version 5.5. The particular display version may suggest using a higher resolution pictures as test data than previous versions, for instance. To further complicate matters, the composition of hardware is not fixed. All the hardware configurations consist of a keyboard and a color display. However, some configurations also include a camera or Bluetooth, or both. Then, when testing a Symbian OS device with a camera but without Bluetooth, for instance, Bluetooth test procedure components should be left out from the tester software.

To manage the variability inherent in the product line, the production testing software is assembled from components pertaining to different layers as follows. Ideally, the basic functionality associated with testing is implemented in the general testing components that only depend on the Symbian OS or certain simple, common test functionality of generic hardware. However, to test the compatibility of different hardware variants, more specialized test procedure components must be used. Moreover, to cover the particular hardware and driver versions, suitable components must be selected for accessing their special features. Thus, the test software is assembled from components, some of which provide more general and others more specific functionality and data for executing the tests.

3. APPLYING ASPECT-ORIENTED TECHNIQUES TO PRODUCTION TESTING

In the following we assess the possibilities of applying aspect-oriented techniques to production-testing by identifying the

most important advantages of the technique in this problem domain.

3.1 Identifying tangling

Strive for high adaptability and support for greater variability implies more complex implementations and a large amount of different product configurations. Attempts to group such varying issues and their implementations into optimized components or objects using conventional techniques make the code hard to understand and to maintain. This leads to heavily loaded configuration and large amounts of redundant or extra code, and complicates the build system. Thus, time and effort are lost in performing re-engineering tasks required to solve emerging problems. Hence, for industrial-scale systems, such as production-testing software, this kind of tangled code should be avoided in order to keep the implementation cost-effective, easily adaptable, maintainable, scalable, and traceable.

Code tangling is evident in test features with long historical background. The need for maintaining backwards compatibility causes the implementation to be unable to get rid of old features, whereas the system cannot be fully optimized for future needs due to the lack of foresight. After few generations the test procedure support has cluttered and complicated the original simple implementation with new sub-procedures and specializations. As an example consider testing a simple low-resolution camera with fairly small photo size versus a mega-pixel camera with an accessory flashlight. In this case the first generation of production-testing software had fairly simple testing tasks to perform, perhaps nothing else but a simple interface self-test. However, when the camera is changed the whole testing functionality is extended, not only the interface to the camera. In addition to new requirements regarding the testing functionality, also some tracing, monitoring or other extra tasks may have been added. While the test cases still remain the same, the test procedure becomes heavily tangled piece of code.

Another typical source of tangling code is any additional code that implements features not directly related to testing but still required for all or almost all common or specialized implementations. These are features such as debugging, monitoring or other statistical instrumentation, and specialized initializations. Although the original test procedure did not require any of these features, apart from specialized products and certainly should be excluded in software in use in mass production, they provide useful tools for software and hardware development, research, and manufacturing. Hence, they are typically instrumented into code using precompiler macros, templates, or other relatively primitive techniques.

In object-oriented variation techniques, such as inheritance and aggregation, the amount of required extra code for proper adaptability could be large. Although small inheritance trees and simple features require only a small amount of additional code, the amount expands rapidly when introducing test features targeted for not only one target but for a wide variety of different, specialized hardware variants. Redundant code required for maintaining such inheritance trees and objects is exhaustive after few generations and hardware variants. Hence, the conserved derived code segments should provide actual additional value to the implementation instead of gratuitous repetition. Furthermore, these overloaded implementations easily degrade performance. Hence, the variation mechanism should also promote light-weighted implementations, which require as little as possible extra instrumentation.

Intuitively, weaving the aspects into code only after preprocessing, or pre-compiling, does not add complexity to the original implementation. However, assigning the variation task to aspects does only move the problem into another place. While the inheritance trees are traceable, the aspects and their relationships, evolution and dependencies require special tools for this. Hence, the amount of variation implemented with certain aspects and grouping the implementations into manageable segments is the key asset in avoiding tangling with at least tolerable performance lost.

3.2 Partitioning to conventional and aspect-oriented implementation

The Symbian OS provides more abstract interfaces on upper and more specialized on lower layers. Hence, Symbian OS components and application layers provide generic services while the hardware dependent implementations focus on the variation and specializations. In order to manage this layered stucture in implementation a distinction between conventional and aspect-oriented implementation is required. Separating features and deciding which to implement as aspects and which using conventional techniques is, however, a difficult task. On the one hand, the amount of required extra implementation should be minimized. On the other hand, the benefits from introducing aspects to the system should be carefully analyzed while there are no guidelines or history data to support the decisions.

We propose a solution where aspects instrument product level specializations into the common assets and hence, provide linking time binding into the system. Furthermore, the common product specific and architecture and system level test functionalities comply with conventional object-oriented component hierarchy. However, certain commonalities, such as tracing and debugging support, should be instrumented as common core aspects and hence, optional for all implementations. Thus, we identify two groups of aspects: test specialization aspects and general-purpose core aspects. The specialization aspects embody product-level functionalities and are instrumented into the lowest, hardware related abstraction level. Secondly, the common general-purpose aspects provide product-level instrumentation of optional system level features.

In this solution we divided the implementation on the basis of generality: Special features were to be implemented using aspect-oriented techniques. These are all special, strictly product-specific features for different hardware variants clearly adding special dedicated implementations relevant to only certain targets and products. On the contrary, however, the more common the feature is to all products, it does not really matter whether it is implemented as part of the conventional implementation or as a common aspect. The latter case would benefit from smaller implementation effort but suffer from lack of maintainability. Hence, com-

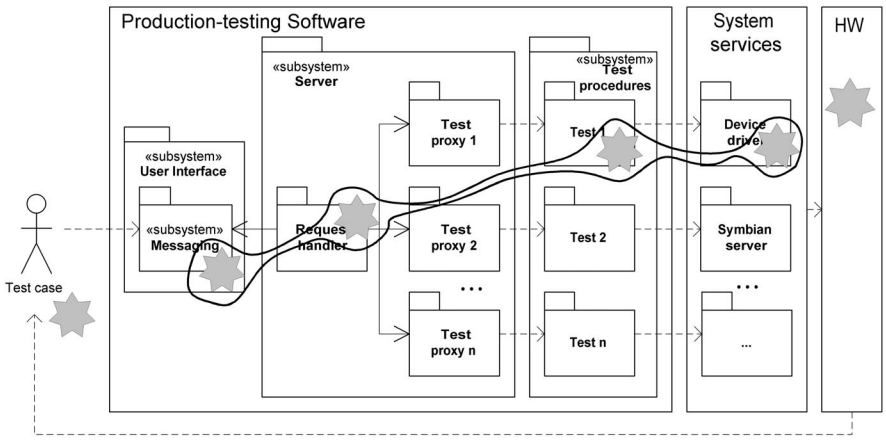

Figure 2: An aspect capturing specialization concern in production-testing framework.

mon aspects are proposed to include only auxiliary concerns and dismiss changes to core implementation structures and test procedures.

3.3 Camera example

We demonstrate the applicability of aspect-orientation in production-testing domain with a simple example of an imaginary camera specialization. In this example, an extraordinary and imaginary advanced mega-pixel camera with accessory flashlight replaces a basic VGA-resolution camera in a certain product in our product family. Since this unique hardware setup is certainly a one-shot solution, it is not appropriate to extend the framework of the product family. Evidently, changes in the camera hardware directly affect the camera device driver and in addition to that, require also changes to the production-line's test cases. New test procedure is needed for accessory flashlight and camera features and the old camera tests should be varied to take into account the increased resolution capabilities. Hence, enhanced camera hardware has an indirect effect on the production-testing software, which has to support these new test cases and algorithms by providing required testing procedures. Hence, camera related specialization concerns affect four different software components, which are all located on different levels of abstraction: the user interface, request handler, related test procedure component, and the device driver. Components requiring changes compared to the initial system illustrated in Figure 1 are illustrated in Figure 2 as grey stars.

From the figure it is apparent that the required specialization cuts across the whole infrastructure and is likely to be difficult to maintain using conventional techniques. In this case, that is how to comply with the extraordinary setup. In practice this could involve new initialization values, adaptation to new driver interface, and, for example, introduce new algorithms. With conventional techniques, such as object-orientation, this would entail inherited specialization class with certain functionalities enhanced, removed or added. Furthermore, a system level parameter for variation must have been created in order to cause related changes also in the server and the user interface level, which is likely to bind the implementation of each abstraction level together. Hence, a dependency is created not only between the hardware variants but also between the subsystem variants on each abstraction level. These modifications would be tolerable and manageable if parameterization is commonly used to select between variants. However, since this enhancement is certainly unique in nature, a conventional approach would stress the system adaptability in an unnecessary heavy manner.

However, the crosscutting nature of this specialization concern makes it an attractive choice for aspects that group the required implementation into a nice little feature to be included only in the specialized products. These aspects, which are illustrated in Figure 2 as a bold black outline, would then implement required changes to user interface, request handler, testing component, and device driver without intruding the implementation of the original system. Hence, the actual impact of the special hardware is negligible to the framework and the example thus demonstrates aspect-orientation as a sophisticated approach of incorporating excessive temporary fixes.

4. EVALUATION

In order to gather insight into the applicability of aspects to production-testing system, we assessed the technique against the most important qualities for such system. These include system's adaptability, variability, reliability and robustness, and performance. In addition, major concerns are the traceability and maintainability of the implementation.

Since the production-testing system is highly target-oriented and should adapt easily to a wide variety of different hardware environments, the system's adaptability and variability are the most important qualities. We consider that by carefully selecting the assets to implement as aspects could extend the system's adaptability with still moderate effort. A convincing distinction between the utilization of this technique and conventional ones is fairly dependent on the scope of covered concerns. While the technique is very attractive for low-level extensions, it seems to lack potential to provide foundation for multiform, large-scale implementations.

Including aspects in systems with lots of conventional imple-

mentations has drawbacks in maintenance and traceability. Designers can find it difficult to follow whether the implementation is in aspects or in the conventional part. As the objects and aspects have no clear common binding to features, following the implementation and execution flow becomes more complex and difficult to manage. Aspects can be considered as a good solution when the instrumented aspect code is small in nature. In other words, aspects are used to produce only common functionalities, for example tracing, and do not affect the internal operation of the functions. That is, aspects do not disturb the conventional development. However, these deficiencies may be caused by the immaturity of the technique and hence reflect designers' resistance for changes. Also the lack of good understanding of the aspect-oriented technology and proper instrumentation and development tools tend to create skeptic atmosphere. However, the noninvasive nature of aspect-oriented techniques makes it superior technique in incorporating tracing and debugging features.

Production-testing software should be as compact and effective as possible in order to guarantee highest possible production throughput. Hence the performance of the system is a critical issue also when considering aspects. Although the conventional implementation can be very performance effective, the aspects provide interesting means to ease the variation effort without major performance drawbacks.

5. DISCUSSION

In this paper, we have described an approach for assembling production-testing software from components that provide test functionality and data at various levels of generality. To implement this product line architecture, we have described a solution based on aspects. The solution depends on the capability of aspects to weave in new operations into already existing components, possibly overriding previous ones. Thus, the solution provides functionality that is specialized for the testing of the particular hardware configuration.

One practical consideration in mobile setting is the selection between static and dynamic weaving. While dynamic weaving adds flexibility, and would be in line with the solution of [4], static weaving has its advantages. The prime motivation for advocating static weaving is memory footprint, which forms an issue in mobile devices. Therefore, available tool support [5] is technically adequate for our purposes.

Unfortunately, tool support for weaving is not the only source of problems in our case. The tool chain of the Symbian development is built around GCC version 2.98, with some manufacturer specific extensions needed in mobile setting [6]. Our first attempts indicate that using tools enabling aspects in this setting is not straightforward but requires more work. While in principle we could circumvent the problem by using mobile Java and AspectJ [7] to study the approach, hiding the complexities of the implementation environment would not be in accordance to the spirit of the problem, where specialized hardware and tools are the important elements.

So far we have not tried out our approach in actual production testing, mainly due to the aforementioned problems. Thus, it remains as future work. We would also like to investigate more on the possibilities aspects could have in conjunction with product family architectures. Especially, the compositionality of aspects in the setting where platform-specific tools are needed is an open issue.

6. REFERENCES

[1] J. Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. Addison–Wesley, 2000.

[2] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison–Wesley, 2004.

[3] R. Harrison. *Symbian OS C++ for mobile phones*. John Wiley & Sons., 2003.

[4] J. Pesonen. Assessing production testing software adaptability to a product-line. In *Proceedings of the 11th Nordic Workshop on programming and software development tools and techniques (NWPER'2004)*, pages 237–250, Turku, Finland, August 2004. Turku Centre for Computer Science.

[5] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

[6] C. Thorpe. Symbian OS version 8.1 Product description. At http://www.symbian.com/ on the World Wide Web.

[7] AspectJ WWW site. At http://www.eclipse.org/aspectj/ on the World Wide Web.

Two Party Aspect Agreement using a COTS Solver

Eric Wohlstadter
University of British Columbia
wohlstad@cs.ubc.ca

Stefan Tai
Thomas Mikalsen
Isabelle Rouvellou
IBM Watson Research Center
stai,tommi,rouvellou@us.ibm.com

Prem Devanbu
University of California, Davis
devanbu@cs.ucdavis.edu

ABSTRACT

A number of researchers have proposed an aspect-oriented approach for integrating concerns with component based applications. With this approach, components only implement a functional interface; aspects such as security are left unresolved until deployment time. In this paper we present the latest version of our declarative language, GlueQoS, used to specify aspect deployment policies. Our work is focused on automating the process of configuring cooperating remote aspects using a client-server handshake. During the handshake the two parties agree on aspect configuration by using mixed integer programming. A security example is presented as well as initial performance observations.

1. INTRODUCTION

Extending component interfaces directly with information about non-functional concerns limits the reusability of an interface. Each component implementing the interface must be prepared to handle these concerns appropriately. Furthermore, it also limits customizability, for example, the ability of local security officers to tailor policy enforcement code to suit their settings.

To address this shortcoming, a number of researchers have proposed an aspect-oriented approach for integrating concerns with component based applications [18, 6, 8, 11, 14, 19]. With this approach, components only implement a functional interface; aspects such as security are left unresolved until deployment time. A pointcut specification, written by a deployment expert, can be used to weave aspects and the original components. As presented, the approach does not consider the issue of matching client-side aspects to the deployment on the server. This is important when client and server-side aspects must cooperate [19, 13], security or fault-tolerance aspects being prime examples.

This inflexibility limits the use of the approach in new, emerging application areas such as service-oriented architectures (SOA). In a SOA, client applications and server applications from the same product family are not always consistently deployed across a wide-area. This may yield variation in the features[1] of client and server software. We propose to provide *dynamic and symmetric reconciliation* between the (potentially different) features (implemented as aspects) of two communicating processes. However, different aspects can interact in various ways, and this complicates reconciliation.

We use the term *interaction* [21] to reflect how aspect combinations affect each aspect's ability to function as it would separately. Interactions can be complex, subtle, and very difficult to identify. Finding such interactions is outside the scope of this paper. In addition, aspect configuration is a matter of deployment policy and can vary.

In this paper we present the latest version of our declarative language, GlueQoS, used to specify aspect deployment *policies*. A middleware-based resolution mechanism uses these specifications to dynamically find a satisfying set of aspects that allow a client and server to inter-operate. We have presented a previous version of this work in [20]. The motivation and example we present are an update of our previous work. However, this paper additionally describes a completely revised language design and implementation.

The remainder of this paper is organized as follows: Section 2 presents the GlueQoS language, Section 3 presents an example, Section 4 presents the COTS solver we used, Section 5 presents implementation, Section 6 reviews related work and finally we conclude in Section 7.

2. POLICY LANGUAGE

Policies are specified in the GlueQoS policy language. The language provides a set of built in operators to specify acceptable aspect *configurations*, as well as the ability to extend the system with functions to measure operating conditions (such as load, available energy, and bandwidth). A configuration includes which aspects should be used as well as what their operating parameters should be.

Each client to server session is associated with a set of *adaptlets* [19]. An adaptlet conceptually encapsulates a pair of client and server aspects. This includes the provided and required interfaces of client and server aspects. This scoping of aspects can be viewed in analogy to the "`aspect per`" scoping of AspectJ but based on a client/server negotiated session. The adaptlet abstraction serves to properly type the "connection" between cooperating aspect instances. Our work is focused on automating the process of configuring these instances using a client-server handshake. During the handshake the two parties agree on a set of adaptlets to use, as well as the values of any parameters they expose. In this section we detail the constructs in GlueQoS used

[1]We use the term *feature* to denote an artifact of software requirements and *aspect* to denote an artifact of software implementation.

to automate this handshake including support for boolean constraints, linear constraints, and run-time monitoring.

2.1 Boolean Constraints

Aspect agreement can range from a very simple problem (e.g., when all aspects are orthogonal (non-interacting)), to a very hard problem (e.g., when aspect interactions are arbitrary). Since aspect-oriented middleware systems are not widely deployed, we draw on work in other areas [5, 21, 1] in order to hypothesize what an ideal framework requires. Therefore, our description is highly expository in nature rather than purely prescriptive.

Each host (client or server) will need to include in their policy a set of statements to impose some requirements on the adaptlets used in a session. If an adaptlet is used in a session, we say the adaptlet's *status* is *on* (true) for that session; otherwise, the adaptlet's status is *off* (false).

Due to the nature of interactions, sometimes one adaptlet may be dependent on another adaptlet. Also, adaptlets might conflict: viz, they cannot be used together. Finally, since hosts do not have *a priori* knowledge of which adaptlets are supported by a peer, it is useful to provide choices amongst a number of adaptlets, in order to meet some requirement. Here we present the encoding of these constraints in GlueQoS, whose syntax follows from boolean logic:

- $\mathcal{D}ependency$: The deployment of an aspect, A, depends on the deployment of another aspect, B. This can be encoded as an implication, `(A implies B)`.
- $\mathcal{C}onflict$: Two aspects conflict if their combination has a negative effect on the behavior of the entire application. The deployment of one aspect should exclude the deployment of the other. The decision that an effect is negative is application dependent but may include effects such as introducing deadlock, putting data in inconsistent states, or degrading performance. $\mathcal{C}onflict$ is encoded by, `not(A and B)`.
- $\mathcal{C}hoice$: Either aspect or both can be chosen to meet some requirement. This is encoded as `(A or B)`.

Based on these examples, it is straightforward to encode other requirements such as those stemming from three-way interactions. Now, given that in a SOA agreement may need to be performed at run-time, one can see it is desirable to provide for efficient computation of aspect status based on host policies. However, even deciding status for policies of $\mathcal{C}hoice$ and $\mathcal{C}onflict$ is not easily computable (i.e., not tractable) in general. The important question that we address in the remainder of this section is, "Is there a reasonable restriction of arbitrary boolean constraints that is tractable?". We believe the answer is no, so we have opted not to impose any restrictions on boolean constraints.

One way to answer this question would be list all known tractable restrictions [7] and argue why each one is not reasonable. We believe this is possible, however the list in lengthy, and contains relatively few restrictions actually used in any practical setting. Noteworthy examples from the list are 2-SAT and Horn-SAT. Instead, we describe a minimal restriction of boolean constraints which is still intractable, yet we argue is reasonable. By minimal we mean that it is difficult to imagine how one could usefully restrict it further.

We start with 2-SAT. It requires that dependencies be of the form `(A implies B)`. For instance, `((A and B) implies C)` is not allowed. This seems reasonable since software dependencies are usually cast in terms of binary relationships. Now, conflicts are required to be of the form, `not(A and B)`. For instance, `not(A and B and C)`, is not allowed. This seems reasonable since it is difficult to imagine a case where two software packages don't conflict *until a third* is present. Finally, choices are required to be of the form `(A or B)`. For instance, `(A or B or C)` is not allowed. Our argument rests on the fact that we believe this restriction seems unreasonable. So, relaxing 2-SAT slightly to allow multiple choice gives us our restriction which is reasonable yet intractable. An equivalence reduction between 3SAT (a classical intractable problem) and this relaxation of 2-SAT is straightforward. This is by no means a formal proof, but we hope it gives the reader insight into our language design.

Since no shortcuts seem likely, we appeal to brute force provided by a COTS constraint solver. In terms of language design, we have traded off scalability for expressiveness. Arbitrary interactions are supported but can only be reasoned over efficiently if the total number of aspects on a given system is limited. From our initial experiments we believe support for up to 50 aspects should be easily manageable.

2.2 Linear Constraints

Now that we have addressed the motivation and interpretation for policies regarding acceptable adaptlet status, we turn to the matter of adaptlet parameter constraints.

Every adaptlet may need to be configured according to a set of parameters. This is analogous to Component Oriented Programming [16]. For example, in the Java Beans component model every bean may expose a set of attributes for deployment time configuration. However, in our scenario we must allow for joint agreement, between client and server, of the session-time parameters.

For this purpose, we allow the representation of *linear constraints* [15, 4] over adaptlet parameters. For example, a linear constraint could be used with two adaptlets implementing a service-level agreement,

$$-2.0 * PayFeature.price + 1.0 * QoS.guarantee = -100.0$$

This constraint sets the price of a connection at fifty dollars plus half the amount of bandwidth reservation. Graphically, this allows clients and servers to negotiate a choice of the two aspect parameters (referenced as fields of the aspects) anywhere along the line defined by the equation.

In contrast to systems of non-linear constraints, linear systems are decidable and tractable. Thus far we have not explored support for non-linear constraints. Modern solvers are usually based on the *Simplex* [4] algorithm due to Dantzig.

2.3 Run-time Policy Adaptation

Recall that hosts execute in an environment that is continuously changing; they might need to be configured according to a dynamic deployment context. Rather than force deployment experts to constantly update policies manually, our policy language includes constructs to reflect these environmental changes. The constructs are of two types: user-defined value functions and user-defined predicate functions.

The values of coefficients or constants in linear constraints can be input through user-defined value functions. Evaluation of these functions occurs periodically throughout the execution of client and server applications. Before policy resolution occurs, a "snapshot" of the client and server policies is taken to reflect their current states. A similar approach is used in QuO [11], however, not in the context of aspect agreement. For example, we can update the example given as,

$$-2.0 * PayFeature.price + 1.0 * QoS.guarantee = \{cpuLoad() * 100.0 - 100.0\}$$

Graphically, this allows the expression of a line which is shifted vertically based on the current value of the user-defined function `cpuLoad`.

Likewise, requirement of a particular adaptlet in an acceptable configuration may also depend on the state of the execution environment. A security feature, for example, may only be required for certain types of network connections e.g.,
Password and (Encryption when { linkType("mobile") }).

Here, the required configuration varies between using the Password feature alone and using both the Password and Encryption feature. This variation is based on evaluation of the `linkType` user-defined predicate using our `when` keyword.

We have shown that the acceptable feature configurations may vary dynamically. The actual policies expressed depend on the moment when resolution occurs. We have provided two constructs in our language to express this variation.

2.4 Special Functions

In addition to the language elements we have laid out so far, two special types functions are supported. These are the `Supports` and `Preference` functions.

Our client/server scenario must account for the fact that client and server policies are written in isolation. Therefore, the sets of adaptlets mentioned in each policy might not be the same. We chose the semantics that any adaptlet not mentioned in both policies would be assumed to have its status forced to off. This default assumption can be suppressed by adding a `Supports` clause. For example, the clause `Supports(A)` can be interpreted as adding the tautology `(A or not(A))` to the set of constraints.

Assuming the client is given some *Choice* (as in Section 2.1) between adaptlets to meet a particular requirement, the `Preference` function provides a way to instruct the COTS solver which adaptlet to choose. Optimization methods based on linear programming allow for computation of a solution which maximizes some utility function over the constraint variables. Leveraging this utility function we can support preferences over the configuration of aspects from any possible configurations. Currently the `Preference` function is only available to clients because the asymmetry in our handshake protocol (see Section 5.1) cannot provide proof that any server `Preference` functions would be respected. We plan to revisit this restriction in future work.

In the following section we demonstrate a possible usage of the language elements laid out in this section.

3. SECURITY EXAMPLE

Consider deployment of a client/server application in an environment where two security adaptlets are required. The first is *authentication*. The server must protect certain services from unauthorized access; so client requests must be preceded or accompanied by an authentication step involving the presentation of credentials in order to gain group membership for those services. Credentials can be based on a password, or on public-key signatures. In this case, an aspect on the server side is responsible for checking credentials, and the corresponding aspect on the client-side is required to present the appropriate credentials.

The second adaptlet, the *client-puzzle protocol* (CPP) [5], defends against denial-of-service (DoS) attacks. A DoS attack occurs when a malicious client (or set of malicious clients) overloads a service with requests, hindering timely response to legitimate clients. Certain components of the server may be prone to DoS attack because of the amount of computation required by the components. CPP protects a component by intercepting client requests and refusing service until the client provides a solution to a small mathematical problem.

CPP and Authentication interact in interesting ways. For example, suppose the server's only requirement is to prevent DoS attacks. If we trust authenticated clients not to mount DoS attacks, then the authentication and client-puzzle adaptlets are equivalent and one can be substituted for the other; it would be redundant to use both. However, sometimes authentication may not imply a decreased risk of DoS attacks, so these adaptlets would be viewed as orthogonal. In other situations, we may require both authentication and DoS defense.

Client-side preferences must also be considered when selecting the adaptlets that govern a client-server interaction. A client may consider CPP and Authentication to be equivalent, and express a policy that it can use either. A client with a performance requirement, however, would naturally prefer to employ authentication to avoid computing puzzle solutions. A client who values its privacy would prefer to expend CPU cycles in order to not have to reveal their identity; this client may prefer to use CPP rather than provide identity-revealing credentials.

Figure 1 is a realization of the security policy as expressed in GlueQoS. The first policy is shown for the server.

Each line (1, 2, and 3) represents a different configuration constraint. The first is an implication between the status

Server:
```
(1) (Authentication implies (CPP.size = { cpuLoad()*8 }));
(2) (CPP when { cpuLoad() > .5 });
(3) (Authentication or (CPP.size = { cpuLoad()*16 }));
```

Client1:
```
(4) Authentication;
```

Client2:
```
(5) Supports(CPP,Authentication);
(6) Preference(not(Authentication),Authentication);
(7) (CPP.size <= 4);
```

Figure 1: Security Example

of the Authentication adaptlet and a constraint on the size parameter of the CPP adaptlet. It states that with Authentication, the size of puzzles varies linearly from 0 to 8 depending on CPU load. Another constraint (line 2) uses a predicate `(cpuLoad() > .5)` to determine whether CPP is required. When CPU load is less than .5, the server allows Authentication to be used without the CPP; otherwise just CPP, with the largest puzzle size, can be used. This shows how run-time conditions can dynamically adapt the acceptable adaptlet combinations expressed by hosts.

The first client policy is shown on line 4. This client will only use the Authentication adaptlet (perhaps because of software availability, or because it is too performance-limited for CPP). Therefore, this client can only create a session with the server when the server's load is less than 0.5.

The second client policy (lines 5-7) uses parameter constraints to choose between two adaptlet combinations. Note that the `Preference` semantics in our language denotes a preference for the first alternative. Consider a situation where this client wishes to maintain its anonymity by not using the Authentication feature. However, it also has a performance requirement that takes precedence. Perhaps the client is on a mobile device with low computing power. Line 6 expresses the client's preference to maintain anonymity. However, in order to keep performance at a certain threshold the client will also use Authentication if it will keep the puzzle size low. By comparing to the sample server's policy (lines 1 and 3 in particular): if this client contacts the server when the server's CPU load is 25 percent or lower the client can maintain its anonymity by using CPP only (from line 3 and 7, $16 * .25 <= 4$). However, if it contacts the server and the server's CPU load is between 25 percent and 50 percent it will agree to reveal its identity to maintain higher performance (from line 1 and 7, $8 * .5 <= 4$). When the server's load passes 50 percent the client will be not be able to find a solution to the constraints imposed by the policy.

4. MIXED INTEGER PROGRAMMING POLICY MATCHING

In Section 2, we described our policy language for expressing adaptlet configurations. We have made some informed design decisions and arrived at an implementation based on mixed integer programming. Pragmatically, the best choice for these decisions would be based on best practices observed over a number of years. Certainly this is difficult as aspect-oriented middleware is not widely deployed in practice.

Mixed Integer Programming has been used widely in the area of Operations Research [4] for decades. Here we apply this technique for automating the configuration of aspect-oriented software in a client/server setting.

Mixed Integer Programming extends the theory of linear programming. In a mixed integer program (MIP) a subset of variables can be constrained to integer values. Hence, the "mixed" denotation refers to a mix of real and integer variables. A popular strategy for solving a MIP is based on the Branch-and-Bound [15] algorithm. In this paper we view the MIP algorithm as a COTS component that is utilized for the purpose of resolving policies. This is achieved by modeling adaptlet status as 0/1 integers and adaptlet parameters as integer or real variables.

5. IMPLEMENTATION

Our prototype implementation builds on the existing DADO dynamic AOP middleware [19] and the Lindo API [10] for mixed integer programming. This involves attaching policies to applications, maintaining a run-time representation of policies, and finally deploying the properly parameterized resolved aspects.

A deployment expert considers local requirements and aspect interactions to design a policy. The policies are associated with CORBA interface types, before an application is executed. Our implementation currently does not support policies on a per-method basis; a single policy can be assigned to each interface type. At application load-time the GlueQoS middleware builds a data-structure representing these policies. Now we describe the overall set-up as in Figure 2.

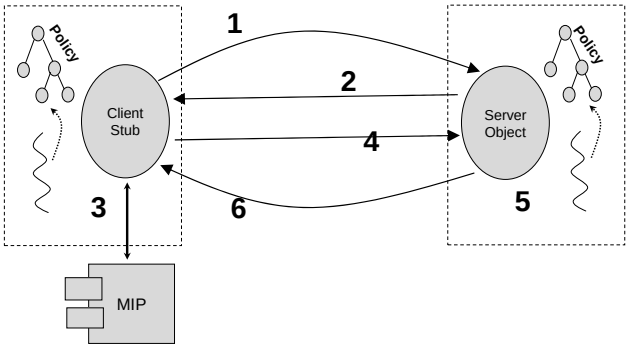

Figure 2: The overall flow of the GlueQoS runtime, including client stub, server stub, and the Mixed Integer Programming (MIP) runtime component

The figure represents the client and server runtime using our GlueQoS middleware, separated on the left and right sides respectively. The dotted-line boxes represent the boundary between middleware related functionality and the black-box MIP component.

Inside the dotted lines are three pieces. First, the large circles represent the client stub and server object to which the session based aspect agreement applies. Second, the tree of nodes represents the policy data-structure. Third, a separate thread, shown as the curved line, is responsible for updating this data-structure based on the values retrieved from user-defined functions. Now we focus on the interaction defined by the numbered flow of the diagram.

5.1 Client/Server Interaction

The GlueQoS middleware at each end of an interaction determines adaptlet configuration for each application session. These aspects and their operating parameters remain fixed for the lifetime of the session. In the future, we plan to investigate support for continuous adaptation of operating parameters.

When a client locates a server, it sends a *policy request* (1) to the server object to initiate a session. Policy requests are implemented as a CORBA operation that is transparently introduced to all IDL interfaces. This is performed by a compiler that is part of our DADO toolset.

The server creates a session for the client in the form of a *cookie*. Now, the server serializes the policy data-structure, associates it with the newly created session and returns the serialization to the client. Note that at this point, all run-time adaptation functions have been evaluated out of the policy, creating a static policy based on the current environment. This means that in the example, the server does not have to reveal the fact that policy is based on current cpu load.

Now, the client must match its own policy with the server and choose an adaptlet configuration acceptable to both. First, client and server data-structures are merged. Now, a client matches policies by carrying out the mixed integer program resolution. The merged data-structure and a vector representing the client's preferences are passed to the Lindo API. It will return a satisfying assignment for all variables or signal unsatisfiability (3). These results are used to control the execution of aspects. In the case of unsatisfiability, an exception is thrown to the application to signal incompatible policies.

The configuration chosen by Lindo is used in the creation of aspects which implement the adaptlet collaboration. These aspects are instantiated using the Java Reflection API. The parameter values chosen are passed to the aspect's constructor. The signature of the constructor and parameters for each aspect are part of the adaptlet type. The values can then be used by the advice to configure aspect execution. In this way the resolved aspects are activated and configured according to the policies of both client and server.

The setup chosen by the client is then serialized and sent to the server (4). This message is piggybacked on a subsequent application request to the server. The server must verify that the setup chosen by the client actually satisfies its own policy. This requires only a simple linear time check of constraint satisfiability (5). The values for the variables are plugged into the policy which was associated with the client's session. If verification is successful, the server can discard the associated policy and create aspects in the manner described for the client side. On subsequent requests, the cookie from the client is used to execute aspects and advice on a per-client basis. If verification is unsuccessful an exception is thrown back to the client (6).

5.2 GlueQoS Prototype

Our GlueQoS implementation has been tested on the example presented in this paper and a previous version on an example in a related paper [17].

To understand some of the performance impact induced by the GlueQoS software we measured the overhead of the GlueQoS handshake phase (Figure 2, steps 1 - 5) on the example of Figure 1 with the second client policy. Our experiments showed that the overhead is dominated by the communication costs of steps 1 and 2 in Figure 2.

An important detail missing from this experiment is the fact that only a single example policy was used. Since the policy solver of step 3 grows exponentially with the number of integer variables required in the policy encoding, it will be important to repeat the experiments for a range of policy sizes. We could draw from the approach described in [12]. This work shows how to generate random 3-SAT instances of a desired size and difficulty (i.e., time required to solve the instance). In the future it may be possible to extend that work for generating random policies of varying difficulty that can be used for further experiments.

6. RELATED WORK

Aspect-Oriented middleware is motivated by the need to provide flexible customization with a simplified deployment process, combining the benefits of reflective middleware with container based deployment.

Recently, the open-source JBoss [9] application server announced aspect-oriented deployment of container services using the Javassist [2] byte code editing toolkit. A similar approach is used in the Java Aspect Components (JAC) framework [14] that also utilizes load-time byte code weaving (using BCEL [3]) in Java. New services can be constructed by implementing aspect-specific interceptors. Deployment takes place using the notion of pointcuts from the AspectJ language. In JAC, aspects can be un-deployed/re-deployed dynamically using a standardized API.

The Quality of Objects (QuO) [11] project aims to provide consistent availability and performance guarantees for distributed objects in the face of limited or unreliable computation and network resources. QuO defines an abstraction known as the operating region for processes (client or servers) cooperating in a distributed object environment. Changes in perceived run-time conditions move a process into different operating regions. Advice that is bound to a particular operating region or region transition is the main vehicle by which adaptation is achieved.

The aspect-oriented middleware presented in this section achieve both flexible customization and simplified deployment. This is made possible by a clear separation between adaptation programming and deployment. Deployment is facilitated by pointcut based descriptions which map adap-

tation behavior to application events. Our work on GlueQoS could be used to simplify run-time deployment of cooperating aspects in client-server applications. Previously, we have presented the notion of an adaptlet collaboration which serves to properly type client and server aspect roles.

7. CONCLUSION

GlueQoS is middleware software to support dynamic adjustment of aspects between clients and servers. Configuration preferences are specified in the GlueQoS policy language. These policies are exchanged at binding time between systems interacting in an ad-hoc setting. The polices are then matched up, and resolved by the middleware. The resolved aspects are then deployed and executed. GlueQoS has been implemented in the context of adaptlets.

8. REFERENCES

[1] BEA, IBM, Microsoft, and SAP AG. Web services policy framework (WS-Policy), May 2003.

[2] S. Chiba. Load-time structural reflection in Java. In *Proc. of the European Conference on Object-Oriented Programming*, pages 313–336, 2000.

[3] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universit at Berlin, Institut fur Informatik, 31 pages, 2001.

[4] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, N.J., 1962.

[5] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proc. of the USENIX Security Symposium,* 9 pages, 2001.

[6] F. Duclos, J. Estublier, and P. Morat. Describing and using non-functional aspects in component based applications. In *Proc. of the International Conference on Aspect-Oriented Software Development*, pages 65–75, 2002.

[7] J. Franco and A. van Gelder. A Perspective on Certain Polynomial Time Solvable Classes of Satisfiability. In *Abstracts of the International Symposium on Mathematical Programming*, 1997.

[8] F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. Aspectix: a quality-aware, object-based middleware architecture. In *Proc. of the 3rd IFIP Int. Conf. on Distrib. Appl. and Interoperable Sys.*, 2001.

[9] JBoss. <http://www.jboss.org>. 4.0 edition.

[10] Lindo API. <http://www.lindo.com/>. 2.0 edition.

[11] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. QuO Aspect languages and their runtime integration. In *Proc. of the Workshop on Languages, Compilers and Runtime Systems for Scalable Components,* 16 pages, 1998.

[12] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Prof. of the Conference on Artificial Intelligence*, pages 459–465, 1992.

[13] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed aop. In *Proc. of the International Conference on Aspect Oriented Software Development*, 2004.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible framework for AOP in Java. In *Proc. of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection),* 24 pages, 2001.

[15] M. Simonnard. *Linear Programming.* Prentice Hall, 1966.

[16] C. Szyperski. *Component Software – Beyond Object Oriented Programming.* Addison Wesley, 1997.

[17] Stefan Tai, Thomas Mikalsen, Eric Wohlstadter, Nirmit Desai, and Isabelle Rouvellou. Transaction policies for service-oriented computing. *Data and Knowledge Engineering Journal: Special Issue on Contract-based Coordination and Collaboration*, 51:59–79, 2004.

[18] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B.N. Jorgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the International Conference on Software Engineering*, pages 233–242, 2001.

[19] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Dado: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *Proc. of the International Conference on Software Engineering*, pages 174–186, 2003.

[20] Eric Wohlstadter, Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, and Premkumar Devanbu. Glueqos: Middleware to sweeten quality of service policy conflicts. In *Proc. of the International Conference on Software Engineering*, 2004.

[21] P. Zave. An experiment in feature engineering. *Programming Methodology*, pages 353–377, 2003.

Development environment for configuration and analysis of embedded and real-time systems *

Aleksandra Tesanovic, Peng Mu, and Jörgen Hansson
Department of Computer Science
Linköping University, Linköping, Sweden
{alete,jorha}@ida.liu.se

ABSTRACT

In this paper we present a tool framework that provides developers of real-time systems with support for building and configuring a system out of components and aspects, as well as real-time analysis of the configured system. This way a real-time system can efficiently be configured to meet functional requirements and analyzed to ensure that non-functional requirements are also fulfilled, e.g., available resources such as memory and CPU. We illustrate the usability of the tool by a case study of an embedded real-time database system.

1. INTRODUCTION

Modern real-time computing systems are faced with increasingly complex requirements in their development and operation. Successful usage of these systems in different applications strongly depends on the ability to tailor a real-time system to meet specific needs of the underlying application. Hence, there is a need for an efficient way of developing families of real-time systems, i.e., a product line architecture, suitable for a plethora of applications.

One way to ensure meeting these requirements is to adopt the component-based software development paradigm for real-time systems. This way systems are developed out of pre-defined software components, and can be tailored for a specific real-time application by adding or removing components to/from the architecture. However, approaches to developing families of real-time systems out of pre-defined components existing in a component library, e.g., [9, 15], do not provide efficient support for features that cannot cleanly be encapsulated into components, e.g., temporal constraints, scheduling policies, and synchronization. Aspectual component-based real-time system development, ACCORD [14], is an example of an effort to integrate the two software engineering techniques, aspect-oriented and component-based software development, into real-time system development. ACCORD enables building flexible product line architectures for real-time systems as it enables assembling the system both from components and aspects.

For a real-time system, it is essential that results produced by the system are both produced correctly and in a timely manner. To ensure timeliness, tasks[1] in a real-time system are associated with deadlines, and a number of real-time scheduling techniques has been developed ensuring that tasks meet their respective deadlines. These typically require the knowledge of the worst-case execution time (WCET) of a task, making the issue of determining the WCET essential for real-time systems. Furthermore, majority of real-time systems is also embedded, implying that they have sparse resources in terms of CPU and memory consumption or footprint. Thus, to be able to use a real-time system in a particular run-time environment and an application, analysis techniques need to be available to ensure that non-functional requirements, i.e., WCET and memory consumption needs, are met.

The contribution of this paper is a tool framework that provides real-time system developers with support for configuration and analysis of a real-time and embedded system composed of aspects and components, i.e., a system developed using ACCORD approach. The tool framework includes tools that support a designer in the composition process, as well as in the process of performing WCET and memory footprint analysis, and formal verification of the composed real-time system. We show the way the tool can be used for developing various configurations of a real-time database system, based on the specified functional and temporal requirements of the underlying run-time environment.

The paper is organized as follows. In section 2 we give background information on ACCORD and its main constituents. Then, in section 3, we describe the tool framework we developed to support the process of assembling and analyzing systems developed based on ACCORD. We illustrate the usability and applicability of the tool on the example of the real-time database system in section 4. Related work is discussed in section 5. Finally, the paper is concluded in section 6 with conclusions and directions for future work.

2. ASPECTUAL COMPONENT-BASED REAL-TIME SYSTEMS DEVELOPMENT

Within ACCORD aspects in real-time systems are classified in different categories [14]: (i) application aspects, (ii) run-time aspects, and (iii) composition aspects. The classification eases reasoning about different embedded and real-time related requirements, as well as the composition of the system and its integration into a run-time environment.

Application aspects can change the internal behavior of components to suit a particular application as they crosscut the code of components in the system, e.g., memory

*This work is supported by the Swedish Foundation for Strategic Research (SSF) via the SAVE project, the Center for Industrial Information Technology (CENIIT) under contract 01.07, and the Swedish National School in Computer Science (CUGS).

[1]Real-time systems are typically constructed out of concurrent programs, called tasks.

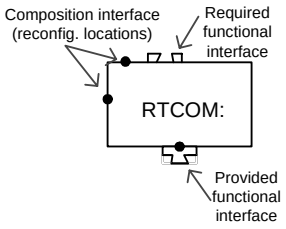

Figure 1: Reconfigurable Real-Time Component Model (RTCOM)

optimization aspect and real-time policy aspect. Run-time aspects give information needed by the run-time system to ensure that integrating a composed real-time system would not compromise timeliness or available memory consumption. Composition aspects describe the version of the component, possibilities of extending the component with additional aspects, and a set of other components and application aspects with which this component can be combined.

ACCORD provides a real-time component model, denoted RTCOM, to support reconfigurability [14]. RTCOM components are "grey" as they are encapsulated in interfaces, while changes to their behavior can be performed in a predictable way using aspects.

Each RTCOM component has two types of functional interfaces: provided and required (see figure 1). Provided interfaces reflect a set of operations that a component provides to other components, while required interfaces reflect a set of operations that the component requires (uses) from other components. Composition interfaces define reconfiguration locations in the component code. Reconfiguration locations define the points in the component code where additional modification of components can be done by aspect weaving. These points can be used by the component user (or component developer) to reconfigure a component for a specific application or reuse context. Reconfiguration locations are also used analysis purposes [13, 12]. Note also that the operations declared in the provided functional interface can be used for aspect weaving and, thus, they represent also implicit reconfiguration locations.

ACCORD enables design of a system both when (1) the components and aspects are not available in the library, and also when (2) there is a pre-existing library of aspects and components developed for a family of real-time systems. In the first case, the design of a real-time system using ACCORD method is performed as follows. First, a real-time system is decomposed into a set of components. Decomposition is guided by the need to have functionally exchangeable units that are loosely coupled, but with strong cohesion. Then, a real-time system is decomposed into a set of aspects. After the design, components and aspects are implemented based on RTCOM. Further, components and aspect that provide specific functionality to the system are grouped into aspect packages. This is to ease the reuse of already developed software artifacts. The second case is illustrated in figure 2 where the composition of the system is done using software artifacts available in a library. Now, in the design phase of the system, the developer chooses appropriate aspects and components and forms a system configuration based on application requirements. Moreover, if the system needs to be modified during its lifetime to support new

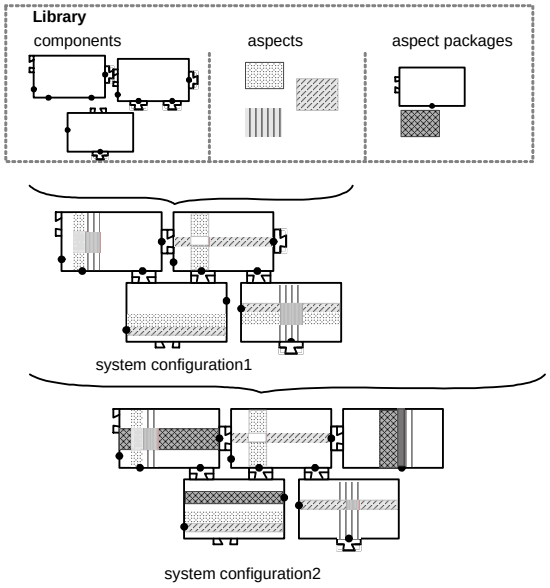

Figure 2: ACCORD-based design

functionality this can be achieved by adding aspect packages that provide exactly the functionality needed.

3. DEVELOPMENT ENVIRONMENT

The ACCORD development environment is a tool framework developed to provide tool support for the system designer when assembling and analyzing a real-time system for a particular application. We assume that, for a particular family of real-time systems, components and aspects are already developed and placed in the library, hence, providing support for case (2) of the ACCORD development process described in section 2.

The ACCORD development environment consists of (see figure 3): the ACCORD library of pre-developed software artifacts, the ACCORD Modeling Environment (ACCORD-ME), and a configuration compiler. In this section we describe each of the constituents of the ACCORD development environment and then discuss limitations and benefits of the environment.

3.1 ACCORD Library

The ACCORD library contains two types of artifacts, namely design-level artifacts and implementation artifacts (see figure 3). Design-level artifacts are: (i) models of components and aspects, (ii) run-time aspects of components and application aspects (as prescribed by ACCORD in section 2), and (iii) formal representations of the components and aspects. Design-level artifacts are used when designing, modeling, and analyzing the system. Implementation artifacts represent the implementation, i.e., source code, of the components and aspects. The implementation artifacts are used when producing the final product, i.e., a compiled code of the system that can be deployed in a specific run-time environment.

3.2 ACCORD-ME

The ACCORD-ME part of the development environment is implemented using the generic modeling environment (GME),

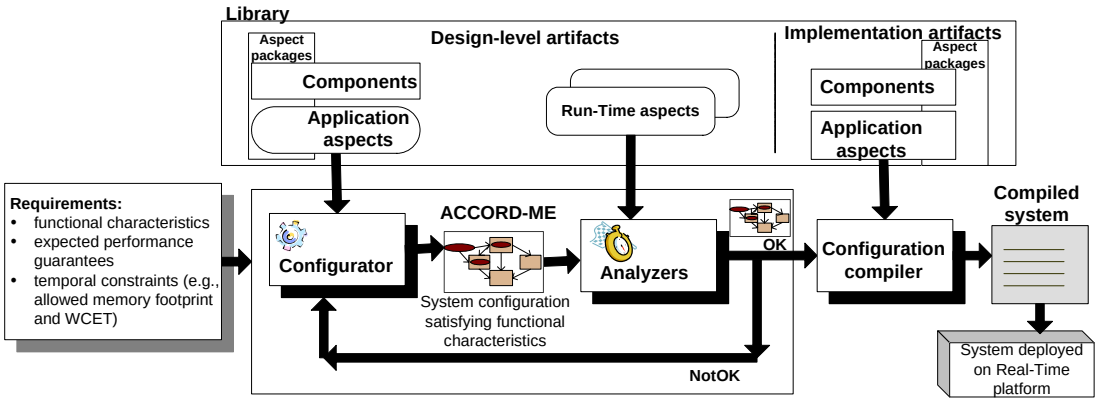

Figure 3: ACCORD Development Environment

a toolkit for creating domain-specific modeling environments [3]. The creation of a GME-based tool is accomplished by defining metamodels that specify the modeling paradigm (modeling language) of the application domain. The GME environment also enables specifying different tool plug-ins, which can be developed independently of the environment and then integrated with the GME to be used for different modeling and/or analysis purposes.

The input to the ACCORD-ME are requirements that are placed on the system. This includes functional and non-functional, e.g., performance guarantees, and temporal and memory constraints, requirements a system should fulfill when used in a specific run-time environment. This tool also uses the design-level artifacts for configuration and analysis of the system. The output of the tool is a configuration file that contains information about the system configuration that fulfills the specified functional and non-functional requirements.

ACCORD-ME is developed with a number of sub-tools that are plugged into ACCORD-ME, namely the configurator, memory and WCET (M&W) analyzer, and formal verifier.

The *configurator* helps the designer to assemble the system configuration by suggesting a subset of suitable aspects and components. This tool provides three levels of support to the system configuration based on the expertise and preferences of the developer.

Expert option is used by developers familiar with the library of components and aspects. The expert option enables the developers to create a number of custom made configurations of the system. This is useful in cases when a comparison of the performance of different configurations is of interest.

Configuration-based option gives a list of possible configurations of the system. This option is intended for developers that can directly express the requirements of the system in terms of available configurations.

Requirement-based option provides the system developer with the list of requirements from which the developer can choose a relevant subset of requirements for a particular application. Thus, developers do not need to have knowledge of what components and aspects exist in the library.

The *M&W analyzer* analyzes a configuration with respect to WCET and memory requirements (the algorithm employed for WCET calculations is presented in [13]). This tool takes as input (i) a configuration file that resulted from the configuration process aided by the configurator; and (ii) the run-time aspects. These run-time aspects contain the run-time information the tool needs to calculate the impact of aspect weaving and system composition with respect to WCET or memory consumption.

We employ symbolic techniques [4] for calculating WCETs and memory requirements of operations. This implies that WCETs in the run-time aspects are expressed in terms of symbolic expressions. Hence, they are a function of one or several parameters that in turn abstract the properties of the underlying run-time environment. The symbolic expressions need to be re-evaluated for each run-time environment, i.e., parameters in symbolic expressions should be instantiated in the analysis. Hence, the M&W analyzer provides a list of symbolic expressions and the list of necessary parameters that need to be configured. Since it is possible to assemble a number of configurations in ACCORD-ME, e.g., when using the expert option in the configurator, the M&W analyzer detects the configurations and enables developers to choose which configuration she/he would like to analyze. Moreover, it is possible to choose whether WCET or memory analysis of the system should be performed.

The *formal verifier* is a tool that performs the formal verification of the composed real-time system. Formal analysis is done based on the formal framework for modular verification of reconfigurable components presented in [12]. The behavior of the system configuration is checked using formal models of aspects and components represented as augmented timed automata with reconfiguration and verification interfaces (stored in the ACCORD library).

3.3 Configuration Compiler

This tool is part of the development environment that aids the designer in compiling the final product. The configuration compiler takes as input: (i) the information obtained from ACCORD-ME about the created configuration that satisfied functional and non-functional requirements, and (ii) the implementation level artifacts. Based on this input it generates a compilation file. This file is then used for compiling source code of needed aspects and components into the final system. Hence, the output of this tool is the com-

piled system configuration, which is ready to be deployed on a specific run-time environment. The configuration compiler also provides necessary documentation about the generated configuration which can later be used for maintaining the system.

3.4 Benefits and Limitations

The ACCORD development environment offers benefits for system developers in terms of automated support during the composition process (via the configurator tool), analysis of the system (via various analysis tools), and the compilation and deployment (via the configuration compiler tool). With this automation the overall development time for a real-time system decreases dramatically.

The tool environment in general, and ACCORD-ME in particular, treats both components and aspects as first-class constituents of a real-time system. Moreover, the uniqueness of the environment is the set of tools encompassed by the ACCORD-ME for analysis of a real-time system woven with aspects (via components' reconfiguration points).

For each family of real-time systems developed using ACCORD, i.e., each new application domain, specific implementation of components and aspects that constitute the domain should be made, and these should be placed in the ACCORD library. Moreover, parts of the ACCORD development environment should be extended to embrace a particular domain (section 4 shows the tool applied to the domain of real-time databases). Namely, the configurator should be extended with a set of requirements and composition rules for this particular domain, and the compilation rules in the configuration compiler tool need also to be updated to contain the rules for compilation of the newly developed components and aspects.

4. EXAMPLE APPLICATION

We illustrate the applicability of the tool on an example of the COMET real-time database system. We first briefly introduce COMET components and aspects, and then show how to develop a COMET database configuration using the ACCORD development environment.

4.1 COMET Aspects and Components

The ACCORD library contains, in this case, a set of COMET components and aspects developed for the domain of embedded real-time databases. COMET components are [8, 11]: a user interface component, a transaction management component, an index management component, a memory management component, a locking manager component, and a scheduling manager component. COMET aspects include concurrency control policies, scheduling policies, transaction model, and QoS policies. Depending on the application with which the database is to be integrated, aspects, components or specific aspect packages e.g., the concurrency control aspect package and the quality of service aspect package [11], can be used in the system composition.

4.2 Developing COMET configuration(s)

The following example illustrates how the COMET configuration can be tailored for a particular real-time system. We focus here on using ACCORD-ME for configuring a product line architecture for a specific electronic control unit (ECU) in vehicles. The ECU at hand is used for controlling the

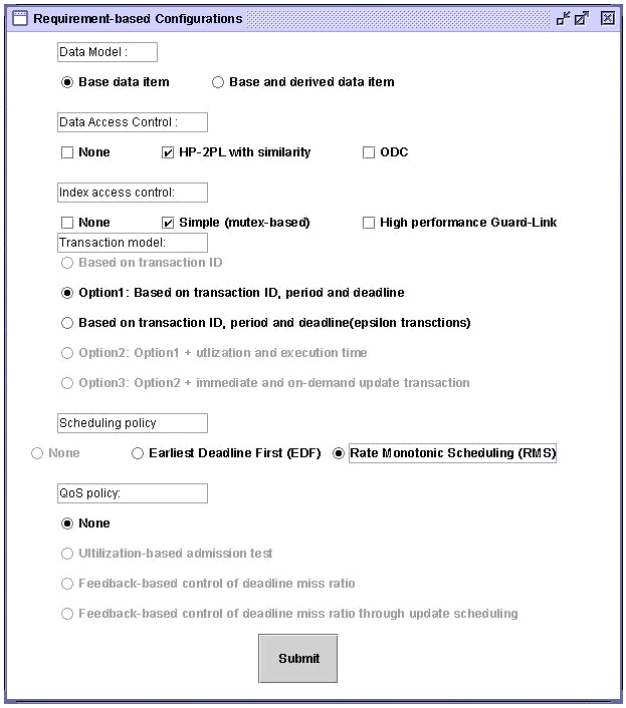

Figure 4: Requirement-based configuration of the real-time database system

engine in a vehicle, and has the following set of data management requirements [7].

R1: The application performs computations using data obtained from sensors; these data items are typically referred to as base data items.

R2: Sensor data should reflect the state of the controlled environment implying that transactions used for updating data items should be associated with real-time properties, such as periods and deadlines.

R3: Values of data should be updated only if they are stale[2] to save computation time of the CPU.

R4: The tasks in the ECU should be scheduled according to priorities.

R5: Multiple tasks can execute concurrently in an ECU. This in turn implies that the same data items can be read and written by different tasks (which could result in inconsistent data values in the system).

R6: The memory footprint of the system should be within the *memBound*, which typically is obtained from the ECU specifications.

When developing a configuration for such a system we start by specifying the requirements using the configurator in ACCORD-ME. Given that we know the system requirements, then the requirement-based option can be chosen in configurator to guide the system composition. Now, based on the requirements R1-R5 we can choose options in the

[2]A data item is stale if its value does not reflect the current state of the environment.

requirement-based form (shown in figure 4) provided by the configurator as follows. The configuration of the COMET database system that is suitable for the ECU at hand should contain only base data items (R1). Furthermore, it should provide mechanisms for dealing with concurrency such that conflicts on data items are resolved (R5) and data items that are stale are updated (R3). This can be achieved using one of the similarity techniques, e.g., HP-2PL with similarity [5]. The transaction model should be chosen such that transactions are associated with periods and deadlines (R2). We choose the rate monotonic scheduling policy [6] to enforce priority-based scheduling of tasks (R4). Performance guarantees in terms of levels of quality of service are not required.

When these requirements are submitted, the configurator loads those components and aspects that could satisfy them into ACCORD-ME. For more efficient composition process one can use help provided in the descriptions of components and aspects in terms of composition rules. Moreover, the relevant components and aspects are grouped in aspect packages for easier system composition.

Observe that if one wants to make several different configurations satisfying a broad spectrum of functionalities, then the expert option in the configurator can be chosen which loads all possible components, aspects, and aspect packages so that the developer can configure the system. A system configuration satisfying functional requirements is shown in the upper part of figure 5. In ACCORD-ME, ovals are used as the graphical representation of aspects, while squares represent components. When the composition of the system is made, it should be analyzed to get the memory and WCET needs of the configuration and contrast these to the prescribed available values of memory and WCETs. Hence, when the configuration part of the system development is finished then the obtained configuration can be analyzed using the M&W analyzer tool (see figure 5 for the snapshot of the analysis process). When the M&W analyzer is invoked, it detects the configuration(s) one might have made in ACCORD-ME and prompts for the choice of configuration. In our example, we created only one configuration and this one is detected by the M&W analyzer (see figure 5). After the configuration is chosen, the appropriate files describing run-time aspects of components and aspects are loaded for analysis. Since run-time properties of aspects and components are described in terms of symbolic expressions with parameters, the values of these parameters should be provided in the analysis. If a parameter is needed for a symbolic expression of a WCET or a memory of a component or an aspect, we say that this component/aspect should be configured for a run-time environment. The list of components that require configuration is shown during analysis, so one can make an inspection of the symbolic expressions, and input the values of parameters in the expressions. Note that advice that modify the components are included in the component run-time description as shown in the lower right corner of figure 5. Once the values of parameters are set for this particular ECU, the tool outputs the resulting WCET and/or memory consumption values which can be compared with the values given in requirement (R6).

If the obtained configuration satisfied the requirements of the engine ECU, the next step is to compile the system and deploy it in the run-time environment, which is done using the configuration compiler. As mentioned previously, the configuration compiler also provides documentation of the composed configuration of COMET.

5. RELATED WORK

A number of GME-based tools exist that address modeling and development of software systems. Here we review only those tools that provide support for development of component or aspect-oriented systems; the full list of GME-based tools can be found in [3]. C-SAW [2] is an example of a GME-based tool that enables weaving of aspect models providing a constraint weaver for nor-real-time systems. VEST [10] is a GME-based tool that supports building component-based real-time and embedded systems with support for run-time aspects. In contrast to our approach and the ACCORD development environment, VEST does not provide support for system configuration (out of components and application aspects), nor does it support the designers in suggesting a relevant subset of components during system composition.

The Koala tool suite [15], which is a tool environment for composition of product line architectures in industry, provides similar support for the developer when assembling an embedded system out of components only.

6. SUMMARY

In this paper we have presented the ACCORD development environment for building embedded and real-time product-line architectures using components and aspects available in the library. In this environment, we provide the support for real-time system developers in all phases of the system development, from requirement specification to configuration, system analysis, and compilation and deployment of the system. The ACCORD development environment is suited both for developers with extensive knowledge of available artifacts as well as ones with little or no knowledge of available components and aspects in the library by having a tool called configurator as the part of the environment. The configurator suggests the subset of aspects and components suitable for the underlying system. Moreover, the analysis tools enable developers to efficiently analyze the system with respect to CPU and memory needs. This analysis is essential in the real-time domain. The tool environment outputs the compiled system configuration as a final product with documentation to ease maintenance of the system.

Our ongoing work focuses on the implementation of the formal analyzer, which provides mechanisms for formal analysis of the system, and its integration into the development environment.

7. REFERENCES

[1] Uppaal tool. http://www.uppaal.com.

[2] Constraint-specification aspect weaver (C-SAW). http://www.gray-area.org/Research/C-SAW/, December 2004.

[3] The generic modeling environment (GME). http://www.isis.vanderbilt.edu/Projects/gme/, December 2004.

[4] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th IFAC Workshop on Real-Time Programming*, May 2000.

[5] K.Y. Lam and W.C. Yau. On using similarity for concurrency control in real-time database systems.

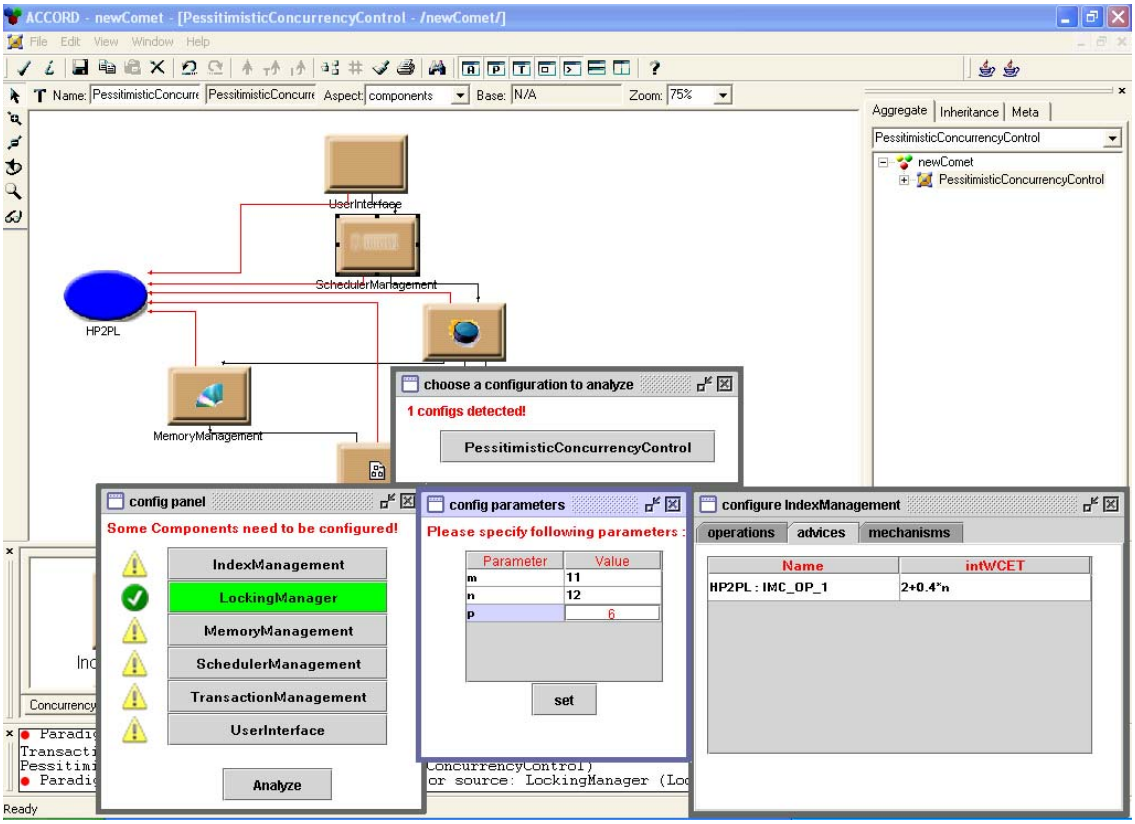

Figure 5: Snapshot of ACCORD-ME when doing WCET analysis on a real-time system configuration

The Journal of Systems and Software, 43(3):223–232, 1998.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time traffic environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[7] D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N-E. Bankestad. Data management issues in vehicle control systems: a case study. In *Proceedings of the 14th IEEE Euromicro International Conference on Real-Time Systems*, June 2002.

[8] Dag Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A component-based real-time database for automotive systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04)*, May 2004. IEEE Computer Society Press.

[9] H. Schmidt. Trustworthy components-compositionality and prediction. *The Journal of Systems and Software*, pages 215–225, 2003.

[10] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: an aspect-based composition tool for real-time systems. In *Proceedings of the 9th Real-Time Applications Symposium 2003*, May 2003. IEEE Computer Society Press.

[11] A. Tešanović, M. Amirijoo, M. Björk, and J. Hansson. Empowering configurable QoS management in real-time systems. In *Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, March 2005.

[12] A. Tešanović, S. Nadjm-Tehrani, and J. Hansson. chapter Modular Verification of Reconfigurable Components. *Component-Based Software Development for Embedded Systems-An Overview on Current Research Trends* Springer-Verlag, 2005.

[13] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components. In *Proceedings of the 27th IFAC/IEEE Workshop on Real-Time Programming (WRTP'03)*, May 2003. Elsevier.

[14] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), October 2004.

[15] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, May 2002. ACM Press.