AOP on the C-side

Bram Adams

Bram.Adams@UGent.be

SEL, INTEC, Ghent University, Belgium

ABSTRACT

Although aspect-oriented programming originally emerged to overcome fundamental modularity problems in object-oriented applications, its ideas have long been backported to legacy languages like Cobol, C, . . . As systems written in these languages are prime targets for re(verse)-engineering efforts, aspects can now be used for these purposes. Before applying dynamic analysis techniques on an industrial case study (453 KLOC of C) using aspects, we devised a list of requirements for possible aspect frameworks. In this paper we explain why no existing framework for C fulfilled all our requirements. We discuss the problems we encountered with Aspicere, our own aspect language for C. We also suggest points of improvement for future reverse-engineering efforts.

Keywords

aspect-oriented programming, legacy software, C, reverse-engineering, comparison study

1. INTRODUCTION

Like any new technology, aspect-oriented programming (AOP) came to life to solve problems inherent to the current state-of-the art, in this case object-orientation (OO), and more in particular Java [15]. Crosscutting concerns were indeed fundamentally ignored in the OO paradigm, so together with their accompanying terminology, aspects revitalized general purpose language research.

As the first waves of enthusiasm set off, people [11, 16] noticed that AOP's ideas were not necessarily tied to OO (and Java). Phenomena like scattering and tangling, the usual indicators for crosscutting concerns, equally (or probably likelier) arise in other OO-languages and less modular paradigms like procedural programming. Soon, every self-respecting language started to get its aspect language, even legacy languages like Cobol [16] and C.

Nearly every organisation is stuck with a battery of mission-critical software written in these old languages. These systems' internal structure and operations are typically no longer known, as the original developers, experienced maintainers or up-to-date documentation are not available anymore. They are inevitably hard to evolve as is, making it nearly impossible to cope with new requirements without prior re-engineering efforts. As aspect technology is getting more widespread in these areas, it can be leveraged to enable reverse-engineering techniques.

In a concrete reverse-engineering case study [23], we used aspects to apply dynamic analysis techniques to a medium-sized (453 KLOC) legacy system written in a mix of K&R- and ANSI-C. Although the role of the aspects there was very light (mere tracing), we envisioned applying AOP for more complex tasks. That is why we did not settle with some ad hoc AOP solution, but thought about relevant requirements for both our purpose back then as well as future efforts.

As it turned out, no aspect language fit the bill completely and we decided to roll our own. Eventually, we succeeded in our experiments, but new requirements came up along the way.

In this paper, we present our requirements gathered thusfar (section 2), as well as their applicability on all currently existing AOP-frameworks for C (section 3). Afterwards (section 4), we discuss possible points of improvement, before concluding (section 5).

2. REQUIREMENTS

To find the most suitable aspect framework for the case study of [23], we devised a set of requirements the preferred aspect framework needed to comply with. Although the functionality expected from the aspects came down to plain tracing, future re(verse)-engineering case studies would be more demanding. Our requirements were designed with this in mind and try to expect the worst.

There are two groups of requirements. The first one deals with specific properties of the whole tool chain, which immediately applies to the aspect framework used. The second one has to do with the specific reverse-engineering techniques used. Some of them are irrelevant here, but others indirectly demand certain functionality from the underlying aspect framework.

These are the tool chain requirements (T1–T5):

T1. Besides "nice" ANSI-C code, the legacy environments we will tackle obviously contain lots of non-ANSI (or K&R[1]) C code, and maybe compiler-dependent extensions too.

T2. The semantics of the original applications should remain intact.

T3. We do not want to delve into the original source code to gain knowledge before deploying our tools, as knowledge mining is exactly what we are after in the first place. I.e. the tools should not require special preparation or exploration of the source code.

[1] Kernighan & Ritchie-style code, after their seminal work [14].

	T1	T2	T3	T4	T5	A2	A3
AspectC	?	+	-	-	+	-	+
AspectC++	-	+	+	-	+	+	+
Aspicere	+	+	+	-	+	+	+
C4	+	+	-	-	+	-	+
WeaveC	?	+	?	-	+	+	+
μDiner	-	+	-	-	-	-	+
TinyC	N/A	+	-	+	-	-	+
Arachne	N/A	+	-	+	-	-	+
TOSKANA	N/A	+	-	+	-	-	+
TOSKANA-VM	+	?	?	-	-	?	?

Table 1: Overview of existing aspect frameworks for C and their relation to our requirements. A1 and A4 do not apply here, while T1 does not for the dynamic weaving mechanisms (except for μDiner), as these operate on binary code. A question mark means we could not decide due to a lack of tools, documentation or both.

T4. The existing build system should remain in place, with only minimal alterations. To refactor it, considerable knowledge of its current internals is needed, which again is lacking.

T5. The tools should be deployable in other environments (operating systems, platforms, compilers, . . .), so that it can be validated against other case studies.

Requirement T2 only depends on the particular advice code used in the experiments, as obliviousness is one of AOP's hallmarks. This remark also partially affects T3, but we will see there is more involved here.

Each of the analyses of [23] had the following number of requirements (A1–A4):

A1. We need a well-covering execution scenario in order to obtain a representative result set.

A2. The data fed into the analyses needs to be sufficiently fine-grained. In the context of C this means individual procedure calls. We also need context information, e.g. on the relevant source modules.

A3. We need information about the call- and return-sequences of procedure calls, in order to get an accurate picture of the dynamic behaviour of the applications.

A4. The analyses should be able to deal with initially unknown amounts of trace data in a reasonable amount of time (i.e. they need to be scalable).

When rephrasing these requirements for reverse-engineering techniques in general, similar demands would arise. We can immediately exclude A1 and A4 from further discussion, as these apply to the specific analysis techniques used. This leaves us with seven requirements.

Based on these two groups of requirements, we will discuss the state-of-the-art aspect languages and mechanisms for C known to us at the time of writing.

3. CURRENT AOP-FRAMEWORKS FOR C

There are various ways to classify aspect frameworks, but we will do this per type of weaving mechanism in a chronological way. Table 1 gives a general overview of all the aspect frameworks mentioned and their relation to the seven requirements.

3.1 Compile-time weaving

All compile-time weaving aspect frameworks operate on the level of source code and transform in some way the advised source code and relevant aspects into regular C code before handing the woven code off to a normal C compiler. Their weaver acts as a source code preprocessor, fullfilling T5 because source code is one of the most portable things in the context of C. All compile-time weavers except for AspectC++ (section 3.1.2) expect a working Java run-time environment, while AspectC++'s PUMA-framework is written in (Aspect)C++.

This scenario has one important drawback, one we unfortunately experienced and had to deal with ourselves [23]: the aspect framework itself crosscuts the existing build system. In order to use a weaver in combination with existing compilers and other processors (for embedded SQL, . . .), the existing makefile hierarchy has to be severely altered, violating T4. Directories with include-files, linking dependencies, . . . can make things really hard. Unless there is only one compiler or tool used or the makefiles are automatically generated and have not been modified manually afterwards (very unlikely), this issue can not be solved easily with some clever trick or one single pass through the build process.

Without special tools like makefile refactorers, rewriters, . . . manual adaptation of the build system is inevitable. Or maybe an aspect framework for makefiles could prove useful?

3.1.1 AspectC

The original AspectC [6] was targeted at tackling crosscutting concerns in operating system code. To do so, the designers started from the original AspectJ [15] and stripped it down by removing all unnecessary (OO-related) features. The resulting language has [5] a special aspect construct, function call/execution join points and pointcuts (A2), before/after/around-advice (A3) consisting of normal C code and well-known pointcuts like "cflow" and "within". Arguments of advised procedures can be assigned to variables and used in advice (not in the remainder of the pointcut expression though, cf. AspectJ).

Although variable argument lists can be abbreviated using "..", return types or procedure names can not (even not using regular expressions). Other means of selecting the right join points, like pointcuts based on structural, semantical, dynamic, . . . information [19] are not provided. Hence, all procedures one wishes to advise must be declared by their exact name in the pointcut, as illustrated in [4]. Also, for every possible return and argument type a separate pointcut *and* advice have to be written down, which is unpractical (T3). Due to performance reasons, there is no thisJoinPoint-struct, so advices cannot access any join point context uncaught in the pointcut (A2).

Equally important, AspectC seems unmaintained since 2003 without any official releases, ruling it out as a viable aspect framework. That is why we do not know much regarding T1.

3.1.2 AspectC++

As (nearly) every compilable C program is also a valid C++ application, AOP languages for C++ could be applied to C base code. AspectC++ [21] is the most mature and general-purpose aspect language for C++ to date, with join point, advice and pointcut types comparable to AspectJ (A3), although more structural pointcut types like "callsto" and "reachable" are available. Advising variable accesses is not supported, because of C/C++'s pointer mechanism.

The aspect construct is in fact the C++ class construct with added pointcut and advice abilities. InterType Declaration (ITD) of new members in classes, structs and unions is also possible. Both advice and ITD are declared in the same way.

As templates have been a part of C++ for years, AspectC++ offered generic (and generative) advice much earlier than AspectJ did [18] (T3). Join point context (A2) like types and values of advised function calls, is easily accessible by a join point API (both static and dynamic parts) and applicable in the advice body.

AspectC++'s weaver[2] is based on the PUMA-framework, a C++ source code transformation system [21] (T5). It processes the whole program at once, demanding drastic changes to the existing build system (T4). Theoretically, ANSI-C code can be advised and subsequently compiled using a (sufficiently template-capable) C++-compiler (with some glitches), but tests with K&R-code failed however (T1). Full C support will only be provided starting from a release near the end of january 2006, feature by feature.

3.1.3 Aspicere

At the time of our case study, only tools for AspectC (alpha), AspectC++ and Arachne (section 3.2.3) were available. As both Arachne's (section 3.2) and AspectC's pointcut and advice model were too restrictive and AspectC++ only partially supported C, we were forced to design and implement our own framework. Since we did not realize the extent of the problems related to T4, we opted for a straightforward preprocessor approach, as C code itself guarantees the best portability of our weaver to other architectures.

Aspicere [1, 23] originally started out as WICCA [22], an AspectC-clone without an explicit aspect construct: aspects are simple compilation units with the extra power of containing advice. Experiments pointed out the T3 shortcomings of AspectC's aspect language, i.e. the inability to write down sufficiently, "generic" advice. Also, our (slow) parser did not cope with T1 and the fixed set of low-level pointcut primitives like execution and args prohibited easy addition of new pointcut types for other research efforts.

We decided to build a new aspect framework based on Logic Meta-Programming (LMP) [13], a template mechanism and a mature ANTLR C parser capable of parsing both K&R- and ANSI-code (T1). Basically, all pointcuts are made up of Prolog predicates and can be mapped one-to-one onto a Prolog rule. The predicates' logic variables can be bound to context information and reused within the pointcut expression to express certain constraints (unification). These bindings can then be applied within the advice body and the advice signature as some sort of template parameter (like C++ has) to denote types, caught arguments, weaving metadata, ... This results in simple, generic advice for C (T3). The around advice type and thisJoinPoint-like struct support A3 and A2. Lexical order of aspects and advice determines their precedence.

Besides the makefile anomaly, the case study of [23] showed that Aspicere's weaver currently works too slow and that its type inferencing capabilities are not perfect yet. Also, the ability of extending the collection of join point types was not really pursued in the current prototype[3], which currently only supports call join points. For the moment, we are experimenting with LLVM in a less naive preprocessor-setup, without the heavy demands of TOSKANA-VM (see also sections 3.3.1 and 4).

3.1.4 C4

AspectC's ideas of aspectizing UNIX-like operating systems using a static (preprocessing) weaver, live on in the aspect language called C4 (CrossCutting C Code) [10]. It aims at replacing the traditional patch system by a simplified AOP-driven, semantic approach. Aspects describe "modifications" to a base system on a higher level than the pure lexical patch(1)-tool does. However, chances of adoption are reversely proportional to C4's complexity.

Basically, the idea is that a programmer writes down advice (so-called woven C4) in situ (A2) in the base program, without any quantification. The C4 unweaver extracts the changes into a separate unwoven C4 file (a semantic patch) which can be freely distributed to everyone or (if needed) converted to a plain patch first. At compile-time, the unwoven C4 is physically woven with the base code.

This unwoven C4 file is in fact a (tweakable) classic aspect written in an AspectC-based dialect capable of ITD in structs/unions and advising global variables, but lacking the "call" and "cflow" pointcuts. The rationale here, is that one can always extract code blocks and encapsulate them into their own methods, which can be advised directly. It is clear that the woven C4 severely violates T3, while the unwoven version suffers from the same disadvantages as AspectC (T3, T4 and A2). No special thisJoinPoint-construct exists.

C4 is based on the XTC-framework [12], an advanced macro facility for C, that takes care of the physical weaving. Domain-specific language extensions like the C4-language (AOP domain) are declared as macro's and mapped onto specific plain C structures and extra type information. C4's unweaver and (logical) weaver are still under construction[4].

3.1.5 WeaveC

WeaveC[5] is a very recent aspect language, in which both pointcuts and advice are written in XML-files. As it aims at becoming a general-purpose language, it has the same join point types as AspectC. Pointcuts are name-based (and wildcarded) and the dynamic pointcut types like "cflow" are only provided in the advanced version of WeaveC. Advice is prioritized to handle conflicts at joint join points using a priority level mechanism. Advice bodies or ITD of types or functions are written down in CDATA-elements of the XML-file. Currently, there is no around-advice yet.

Some predefined context variables (function name, argument types, ...) are available, and variables appearing near the join point shadow can be used freely in the advice body (A2). It is unclear whether generic advice is possible using these context variables (T3).

WeaveC's weaver is implemented in Java, and transforms the AST of the base program. In the advanced version, CodeSurfer[6] is used to perform the necessary analyses.

3.2 Run-time weaving

Then, there are a number of aspect languages with dynamic weavers, based on instrumentation libraries or binary rewriting techniques. As they act on binary code, they fulfill T4 (except for μDiner) and T1, but not necessarily T5, as platform-independence is questionable. Different operating systems use other binary formats and each processor's instruction set potentially needs a modified weaver.

All dynamic approaches provide some sort of around advice or a combination of before and after (A3). They support procedure calls and variable access join point types, but typically there is not enough context information available at the binary level (A2). Generic advice is impossible as all these approaches' languages require duplication of advice and a priori knowledge of return types (T3).

[2]`http://www.aspectc.org/`

[3]`http://users.ugent.be/~badams/aspicere/`

[4]`http://c4.cs.princeton.edu/`

[5]`http://weavec.sourceforge.net/`

[6]`http://www.grammatech.com/products/codesurfer/`

3.2.1 μDiner

μDiner [20] requires that declarations of advisable funtions and global variables are annotated as “hookable”, and that a support library is linked with the created (base) executable, so T4 potentially shows problematic.

Aspects are compiled into shared libraries which are loaded into the advised base application at run-time. Hooks are then constructed to connect advised join point shadows with the right advices, and measures are taken to avoid freezing the base application. The weaving process is processor architecture-dependent (T5).

μDiner’s aspect language features around advice (written in C) which can access the arguments of an advised procedure call, the current value of a global variable and (for variable assignment) the assigned variable. Types are hard-coded in the pointcut (T3). The familiar cflow-pointcut is replaced by an explicit nested call hierarchy, always ended either by a function call or by a variable access.

As Arachne supersedes μDiner (see section 3.2.3), no tools are available for μDiner.

3.2.2 TinyC^2

TinyC^2 [24] was developed independently from μDiner, and relies on the DynInst-instrumentation library instead, which uses the UNIX debugging API (ptrace). Aspects are transformed into self-contained C++ programs driving DynInst. Once compiled, one can dynamically advise a running C application.

Disadvantages of this approach, are the relatively high cost due to DynInst’s use of trampolines and ptrace, and the impossibility of modifying arguments and return values.

The pointcut language of TinyC^2 only supports function call join points. There is both onentry- and onexit-advice (comparable to before and after), containing regular C code (A3). Pointcuts use prefix- or regular expression-based matching of procedure names. Available context includes explicitly bound function arguments and global variables, but no thisJoinPoint-construct (A2). When specified, return and argument types need to be hardcoded (T3).

No implementation of TinyC^2 is freely available on the Internet.

3.2.3 Arachne

Arachne [7] improves on the μDiner framework, as annotating the base code is now obsolete. Dependence on a particular architecture is now localised in so-called “rewriting strategies” which guide insertion of hooks for a particular join point type on a specific architecture. For some reason, Arachne did not function on our machines, but the prototype weaver is still under heavy development[7].

The pointcut language has been reworked, inspired by Prolog (unification). There is also a new join point type: sequences of function call and/or (in)direct variable access join points. This is a natural means for advising protocol-like behaviour, as each element of the sequence can be advised individually (A2).

Unfortunately, advice is just a normal C procedure, so bound variables cannot be used in the advice body like Aspicere allows, nor is there an explicit proceed()-statement. Worse, a procedure’s return type cannot be hidden behind a predicate, nor is there any context data available, apart from captured arguments which are passed to the advice (A2). This means that advice needs to be repeated for all possible return and argument types (T3).

On the other hand, Arachne’s pointcut and join point model should be easily extensible and some issues could be ironed out eventually.

[7]`http://www.emn.fr/x-info/arachne/`

3.2.4 TOSKANA

TOSKANA (Toolkit for Operating System Kernel Aspects with Nice Applications) [8] is another aspect language for C with a dynamic weaver (appearing more or less at the same time as Arachne), but targeted solely at NetBSD’s kernel mode. The weaving mechanism is similar to Arachne’s (i.e. “code splicing”), but aspects are compiled into kernel modules. No prototype is available.

Due to the low-level nature of kernel code, TOSKANA’s aspect language is very limited. Basically, advice is a C procedure with a special return type (“ASPECT”) and it can call special macro’s to proceed with the advised code, access stack state, ... In general, this is too low-level for reverse-engineering purposes (A2). In an initialising function, the advice is then instantiated as e.g. around advice and applied to a specific procedure execution. There are no name patterns or other means to advise many procedures at once.

3.3 VM-weaving

The dynamic weaving approaches of the previous section are restricted by the available information in the advised binaries. As such, richer join point models are hard to provide. What is more, the dynamic weaving approaches result in unoptimized woven applications, as the weaving process happens way after the last compiler optimization passes.

3.3.1 TOSKANA-VM

The folks of TOSKANA decided to take a look at virtual machines and came up with the TOSKANA-VM approach [9]: on top of an L4 microkernel, a bunch of LLVM (Low-Level Virtual Machine) [17] instances and a weaver are running. LLVM is a compiler framework with a universal IR (Intermediate Representation) and life-long analysis facilities. It can be extended with optional components to emulate a real virtual machine. So, the LLVM instances are virtual machine instances, in which applications (or operating systems) are running which have been compiled in the LLVM bytecode format. This way, the optimization problem is fixed and more advanced join point types are possible, as LLVM’s IR stands on a higher level than mere binary code.

A downside of this approach, especially in older environments, are the relatively high infrastructural requirements like the microkernel and suitable operating system personalities (T5).

No information on the aspect language is available, except for the available join point types: call, execution, variable assignment and access. Likewise, no prototype is available on the Internet.

4. POINTS OF IMPROVEMENT

It is important to stress again the fact that our requirements considered worst case scenarios. E.g. if the case at hand is situated on a modern Linux environment on top of a standard Intel processor, T5 is not an issue. If one can fall back on an expert to refactor the build system, then T4 is not important either.

In the general case, work is needed to address T4. Many Java projects are using Ant or other XML-based build systems which can more easily be transformed. Even genuine makefiles are much more recent there than is the case for C systems. Although T4 is not a unique issue for C, it will show up more often.

Besides T4, T3 remains the principal problem when using AOP on crosscutting concerns affecting thoroughly scattered, unrelated join points. Reverse-engineering is one example, especially during the initial phases where one tries to narrow down the focus to the places of interest. Programming conventions [4, 2] are another one.

Only AspectC++ (section 3.1.2), once these features are supported in C, and Aspicere (section 3.1.3) provide both generic pointcuts (i.e. beyond mere pattern matching) and advice. The other frameworks have been designed with more specific crosscutting concerns in mind, where advice reuse is not the issue.

Finally, although all approaches use the terminology and lots of features originally introduced by AspectJ [15], typical C features (or problems) like pointers, dealing with macro's, slightly different dialects (T1), ... have not yet been addressed extensively. This is unlike the Java world, where one can experiment e.g. with the abc-workbench [3]. Built on a solid analysis framework, basic weaving functionality is already provided and people just need to focus on new features which eventually can loop back to the real AspectJ.

Although AOP in C goes back to 2001 [6], no such mature, extensible and general-purpose AOP-framework really took off. This is probably due to C's higher complexity and the lack of a natural higher-level IR like bytecode. As a practical consequence, people need to create a new aspect language like Aspicere from scratch, facing the same low-level groundwork others came across.

5. CONCLUSION

We reviewed the ten currently known AOP-frameworks for C in the context of seven requirements set out for a reverse-engineering case study. Compared to the Java scene, no framework really stands out. Most of the problems are related to the impact on the existing build system and the absence of generic advice.

6. ACKNOWLEDGEMENTS

We would like to thank Kris De Schutter, Andy Zaidman, Marc Fiuczynski (C4) and Pascal Dürr (WeaveC) for their input. Bram Adams is supported by a BOF grant from Ghent University.

7. REFERENCES

[1] Bram Adams, Kris De Schutter, and Andy Zaidman. AOP for Legacy Environments, a Case Study. In *2nd European Interactive Workshop on Aspects in Software*, 2005.

[2] Bram Adams and Tom Tourwé. Aspect Orientation for C: Express yourself. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, 2005.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[4] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. An initial experiment in reverse engineering aspects. In *WCRE*, pages 306–307. IEEE Computer Society, 2004.

[5] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD*, pages 50–59, 2003.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[7] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *AOSD*, pages 27–38. ACM Press, 2005.

[8] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05*, pages 51–62. ACM Press, 2005.

[9] Michael Engel and Bernd Freisleben. Using a LowLevel Virtual Machine to improve dynamic aspect support in operating system kernels. In *4th AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, 2005.

[10] Marc Fiuczynksi, Robert Grimm, Yvonne Coady, and David Walker. patch (1) Considered Harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[11] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD*, pages 36–45. ACM Press, 2004.

[12] Robert Grimm. Systems need languages need systems! In *2nd Workshop on Programming Languages and Operating Systems (ECOOP-PLOS'05), ECOOP*, 2005.

[13] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD*, pages 60–69. ACM Press, 2003.

[14] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[16] Ralf Lämmel and Kris De Schutter. What does Aspect Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[17] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[18] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In Gabor Karsai and Eelco Visser, editors, *Proc. Generative Programming and Component Engineering: Third International Conference*, volume 3286, pages 55–74. Springer, October 2004.

[19] Klaus Ostermann, Mira Mezini, and Christophe Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, 2005.

[20] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *AOSD*, pages 110–119. ACM, 2003.

[21] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[22] Stijn Van Wonterghem. Aspect-oriëntatie bij procedurele programmeertalen, zoals C. Master's thesis, Ghent University, 2004.

[23] Andy Zaidman, Bram Adams, Kris De Schutter, Serge Demeyer, Ghislain Hoffman, and Bernard De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006. Accepted for publication. To Appear.

[24] Charles Zhang and Hans-Arno Jacobsen. TinyC^2:towards building a dynamic weaving aspect language for c. In *FOAL 2003*, Boston, MA, USA.