

# On Reconciling Patches and Aspects

Laurent Burgy Marc Ficuzynski  
Department of Computer Science  
Princeton University

Marco Yuen  
Department of Computer Science  
University of Victoria

Robert Grimm  
Department of Computer Science  
New York University

## ABSTRACT

In previous work, we presented a domain-specific enhancement to C, called C4, that lets developers manage program extensions leveraging techniques inspired by the AOSD model as an alternative to the conventional patch approach [3]. Our goal is to offer: (1) tool *compatibility* letting programmers develop, integrate, modify, and debug C4-based extensions that preserve their existing development workflow and leverages their existing tools rather than requiring additional tools; (2) code *understandability* of the C4 syntax such that is it straightforward for an uninitiated C programmer to use immediately; and, (3) runtime *performance* achieving near-zero overhead such that it can be used even in performance critical execution paths. As such C4 source code can be viewed as the result of weaving in AOSD style *introductions* and *advices* inline into C program. However, C4 lacked a proper representation of its unwoven form—i.e., what’s conventionally in AOSD circles referred to as the pointcut language. This paper makes a case for B4: a *patch*-based pointcut representation of *unwoven* C4 and contrasts it with development-oriented pointcut languages belonging to the AspectC family that have been defined for the C programming language.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

## General Terms

Languages

## Keywords

AOSD for C code, C4, patch

## 1 INTRODUCTION

This paper closes the loop of our work on C4, a domain specific version of C that lets developers both easily and safely manage program extensions (particularly those representing a crosscutting concern) leveraging AOSD techniques. The goal of our work is to improve the extensibility approach for the Linux kernel. Towards this goal we focused on maintaining tool compatibility, understandability, and overall performance. Over the past few years, our work has focused on two parts:

1. The design and implementation of the *woven* (inline) C4 language [3,9]. This language represents an AOSD-enhanced

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACP4IS’09, March 2, 2009, Charlottesville, VA, USA.

Copyright 2009 ACM 978-1-60558-450-8/09/03...\$5.00.

version of the C language and is designed to aid developers with defining an extension using *introductions* and *advice* constructs. It departs from established AOSD practice by adopting a reverse workflow based on code unwaving in order to support current systems practice. Without it, developers would have to adopt a considerably less familiar programming paradigm—i.e., pointcut languages such as AspectC [1].

2. The B4 representation of *unwoven* (externalized) C4 code. This representation captures the (potentially crosscutting concerns contained within a single) extension implemented in C4 in a modular, readable and familiar format that can be reintegrated into the mainline C code using conventional tools.

The combination of B4 and C4 addresses a number of adoption challenges that have been explicitly communicated to us and for which there is anecdotal evidence. Specifically, developers such as those working on the Linux kernel would likely reject anything that was perceived as a new programming paradigm / language (even C++), is difficult to debug, imposed undue runtime overhead, or did not follow the well-established model of inline development, extraction, distribution, and reintegration of extensions.

The rest of this paper is structured as follows. Section 2 presents the motivation for B4, revisits our earlier work that railed against *patch*, and then finishes up by reconciling patches and aspects. Section 3 discusses our AOSD inspired approach and contrasts it with prior art more concretely aligned with conventional AOSD approaches. Section 4 offers some concluding remarks.

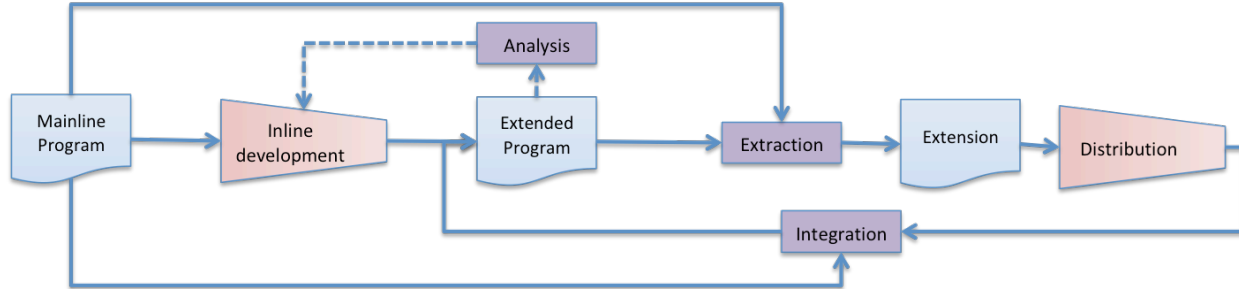
## 2 B4 MOTIVATION & FRAMEWORK

This section provides an overview of our approach to manage extensions of C-based software systems. Our approach includes the C4 language, its B4 representation, and the associated extraction and integration tools that translate between the two. As our prior work [3,4,7,9] motivate, describe and provide usage examples of the C4 language, we only provide minimal background of it where contextually required. The remaining text first outlines the target development process that motivated the design decisions behind B4.

### 2.1 Revisiting “*patch()* considered harmful!”

In a prior paper [3], we railed against the use of *patch* altogether. At the time the state of *patch* set tools was nascent at best. Specifically we felt that two major limitations made them untenable:

1. *Patch sets are brittle.* Even minor textual modification to the mainline code can cause *patch* to fail, forcing the person applying the *patch* set to (a) manually integrate the extension and (b) maintain this integration as the mainline code base evolves.
2. *Patch sets are difficult to compose.* At the textual level, *patch* sets are generated against the baseline code to which the



**Figure 1: Extension Workflow.** Developers modify the mainline program in place, extract their extension with a tool such as `diff`, and then distribute that extension. Other developers integrate the extension into a version of the mainline program with a tool such as `patch` and compile/run/test the extended program.

extension was added, and therefore the integration of extensions will fail when their patch sets modify the same code sequence(s).

Since then patch set tools significantly matured—e.g., the emergence of various new version control interface layers (StGIT, GUILT or TopGIT) built on top of the *git* version control system ease the burdens outlined by the first limitation mentioned above. This maturation was largely fueled by Torvald’s needs to manage a large number of patch sets for the Linux kernel, and from this experience he codified low management processes into *git*. It is indeed a major breakthrough that has popularized a fully decentralized development workflow, which now spreads beyond the Linux kernel to other projects wishing to viably attract a community of developers who improve the quality of their mainline code base as well as add compelling new extensions.

**Figure 1** illustrates this extensions workflow. As a brief overview, adding extensions creates major *variants* to a mainline code base. The current best practice is to implement such an extension by directly modifying the mainline code base. Developers start with mainline and implement their extension by changing the source code to meet the performance or functionality requirements of their target application domain. For instance, Linux-VServer that provides virtualization for Linux is distributed as a patch set that implements kernel level isolation. Upon completing the development of an extension, the developers extract the extension with tools based upon `diff` to create a *patch set*. They can then share the resulting patch set with others, who in turn integrate the extension into their version of the mainline code base using tools based upon `patch`.

Developers utilize this workflow and expect patch sets containing extension that they can apply to their customized kernel versions. Typical Linux kernel extensions can easily cover over a hundred existing files. At the same time, each extension represents a logical unit and often contains a single crosscutting concern. Many developers at companies, government agencies, and academic institutions utilize it worldwide. Clearly, this specific workflow and style of developing extensions matters!

However, these new patch tools do not at all address the second limitation—i.e., *patch sets are difficult to compose*. For this reason, there often is a strong push to get extensions adopted into the mainline code base. This push commonly referred to as *mainlining an extension* sometimes is motivated by engineering excellence when the design and implementation of the extension truly is the best approach to solve a particular problem. But as we conjecture in [4] there may be other motivations to mainline an extension such as simply avoiding the continual overhead of maintaining “out of tree” extensions whose code sequences

conflict with those introduced by other extensions. As a result, an extension may be mainlined when it would be better for it to remain an extension if there were only a solution to the composition problem. Arriving at such a solution is a major goal of our work.

Towards this goal we explored new ways to make fine-grain extensibility of mainstream systems practical. So far we explored in what ways to address the composition problems discussed above while avoiding any disruption to existing development practices. In other words, we focus on the existing model of inline development, extraction, distribution, and integration, while also maintaining tool compatibility, source code understandability, and overall performance.

## 2.2 Reconciling patches & aspects with B4

To ease the adoption of our AOSD inspired extensions model it is important that it minimizes the need for new tools such as compilers, make, etc. or otherwise modifies the development workflow as outlined in the prior section. We express extensions inline at the source level as aspect *introductions* and *advices*<sup>1</sup> using C4. This provides us with a language-supported mechanism to more precisely integrate and extract extensions into and from a C program.

In this way, the development workflow remains the same with our approach with inline development, extraction, optionally distribution and finally integration of an extension in the mainline code. Mapping our approach to the workflow shown in **Figure 1**, the extension is still implemented by directly modifying the *mainline program* to arrive at the *extended program*. An optional analyzer can be run to verify various properties regarding the C4 extension. Next, the *extraction* process involves unweaving C4-defined extensions from the C source code of the extended program resulting in a stand-alone *extension* based on a patch-derived notation called B4. Specifically an enhanced `diff`-like tool that understands the C4 before/after aspect semantics is used to

<sup>1</sup> C4 currently only supports *before* and *after* advice; it does not support *flow*, *call*, or *around* advice. While we acknowledge the flexibility and power of these additional advice constructs, we do not believe explicit support for this is needed. Specifically we believe it is possible for a developer to leverage the C4 provided before/after support to mimic the semantics of these advices, but an in depth discussion of this is not within the scope of this paper.

perform the extraction and produce the B4 files. At this stage the extension can optionally be distributed using conventional means (e.g., email, web, etc.). The *integration* process involves using the patch tool set to apply the B4 extension that supports an enhanced patch notation. Specifically the enhanced notation is only used to integrate C4 after advice, but for C4 introductions and before advice our solution seamlessly works together with existing patch set tools.

Note that our integration – i.e., weaving – process is notably different from the approach taken by other AOSD tools such as AspectC++ and ACC where weaving is done after the code has been processed by CPP. As a result, we could not easily leverage these existing AOSD tools to complement C4—i.e., by unweaving C4 to an AspectC inspired pointcut language. Moreover, we came to realize that the current state of the art pointcut languages as used by ACC and AspectC++ do not fully address aspect composition problems such as hard to predict behavior related to CPP or difficult modular reasoning—albeit for different reasons than the patch set composition problem.

For these reasons we needed to create our own solution that addressed both patch set composition and aspect composition. We ended up with something surprisingly simple that in a nutshell is a basic variant of the patch notation.

## 2.3 B4 Design

The B4 representation of a C4-based extension at its core is a file whose body consists of a set of unified patch directives. These directives differ from those generated by other diff-based tools in that they leverage the semantics of C4’s aspect introductions and before/after advice.

Consider the following before and after advice applied to `some()` function in a file called `example.c`:

```
int some(void) {
  aspect (example){printf("before advice\n");};
  printf("mainline function body\n");
  return 0;
  aspect (example) {printf("after advice\n");};
};
```

Using the development model mentioned in section 2.1, the resulting unified patch generated by conventional diff-based tools to add the C4 advice would look as follows:

```
--- org/example.c 2009-01-06 01:11:41
+++ new/example.c 2009-01-06 01:11:24
@@ -1,5 +1,11 @@
 int some(void) {
+ aspect (example){printf("before advice\n");};
  printf("mainline function body\n");
  return 0;
+ aspect (example) {printf("after advice\n");};
};
```

The problem with these directives is that any code added before the mainline `printf()` call or after the `return 0;` statement would result in a semantically incorrect integration of the C4 advice. That is, the C4 *before* advice would incorrectly appear *after* the new code, and similarly the C4 *after* advice would incorrectly appear *before* the new code.

Our C4-aware extraction tool produces a B4 patch file with the appropriate chunk header information, i.e., the metadata encapsulated by `@@`. Each chunk range, the one preceded by ‘-’ for the original file and the one preceded by ‘+’ for the new file respects the *l,s* format, where *s* is the number of lines the change

hunk applies to for each respective file. Hence, the C4 before/after advice always semantically appears before/after when applied to such a function, respectively.

For the C4 *before* advice from the above example the resulting directive that is natively understood by all patch tools looks as follows:

```
@@ -1,0 +2,1 @@ int some(void) {
+ aspect (example){ printf("before advice\n");};
```

However, for the C4 *after* advice we specify a new style patch directive as follows:

```
@@ -3,0 +4,1 @@ int some(void) {...}
+ aspect (example) { printf("after advice\n");};
```

To process this directive, which primarily differs by the “{...}” shown at the end of the first line, we developed a straightforward modification to the patch toolset that understood how to find the end of a C function (i.e., the closing curly) and apply the provided change from the bottom up.

Note that it is in this way that we address both patch composition and aspect composition problems. For the former, we avoid both i) false positive integration of code in the wrong spot and ii) false negative rejections of the C4 before/after advice when the same code sequence has been modified in the mainline codebase either by other extensions or a mainline evolution. For the latter, we leverage the existing tool support and best practices to apply patch sets in a particular sequence.

While at its core the B4 representation is a set of patch directives, we are still exploring the utility (especially for very large extensions) to include an AspectC inspired pointcut section at the beginning of the file. This optional section within a B4 file includes a summarized version of all advices and introductions that comprise the C4-based extension. It is optional in that it primarily facilitates better documentation of what the extension is doing with the intent to let developers more concisely discuss/debate the extensions core functionality. But otherwise this optional pointcut section is not used in an actionable manner. Whether this is useful or needed in the future is not clear, as developers today are very comfortable discussing/debating the design and implementation of their extension based upon the raw patch directives.

## 3 APPROACH COMPARISON

The previous section presented B4, a patch-based pointcut but AOSD-inspired representation for C-based software extensions. When considering such software extensions, there are three main concerns: 1) composition of extensions, 2) CPP issues as macros and configuration dependent code is used pervasively throughout most C-based systems, and 3) accuracy of naming the join points to which extension code can be applied.

While the AOSD model inspired our approach, the way we approach the above concerns is fundamentally different compared to two representative and relevant AOSD tools for C-based languages: ACC [5] and AspectC++ [8]. The rest of this section contrasts our and their approach.

### 3.1 Extension Composition

It is common for developers of large C-based systems such as the Linux kernel to compose multiple extensions to produce a specific, customized system. As discussed in section 2, the current development workflow used by such developers composes multiple, independently developed patch sets. In our context where it is possible to have multiple extensions apply advice to

the same joint points, we wish to avoid false conflicts that lead to rejecting the integration of multiple extensions. This may require a composition framework with which one can define how multiple extensions (aspects) are to be applied to the same joint points.

To the best of our knowledge, there is no discussion of how ACC handles the composition of multiple aspects applied to the same code base. Our assumption is that it is handled implicitly by the order with which it is woven into the code base at compile time by ACC compiler.

AspectC++, while not directly applicable to C, nonetheless presents a more powerful composition mechanism to define aspect precedence that could be adopted by ACC or other AOSD for C based solutions. The precedence, defined as an attribute of a join point in the pointcut language, is used to determine the execution order of advice code if more than one aspect affect the same join point. The drawback with this approach is that one needs to explicitly know the names of all the aspects, which in our context is undesirable as often extensions (each containing an aspect) are independently developed in a manner where the developers are completely oblivious of each other. AspectC++ also supports the notion of an integration aspect, which can also be used to declare a separate ordering; however, the patterns used to filter out the aspects are based on wildcard matching of names, raising the same issue concerning name knowledge.

In contrast, the ordering of C4 aspects is done as part of the separate integration phase—i.e., the application order of the B4 based patch sets explicitly define the execution order of advice applied to the same join points which is sufficient due to the very nature of our approach. In this way, it is completely independent of the target build or compilation system, whereas ACC for example requires that developers switch to their supplied compiler and make tools. Moreover, while our approach lacks the fine-grained precision offered by the AspectC++ ordering attribute to explicitly define the execution ordering of advice, a person responsible for the integration of multiple extensions can manually mimic this by breaking the B4 patch into a fine-grained patch set and then apply it in the appropriate order to obtain the same result. This may seem clunky, but we do not expect to need such fine-grained ordering precision in the common case. For this reason we feel that it is best to leave out such explicit ordering attributes.

### 3.2 Addressing CPP Challenges

Ernst et al. [2] studied preprocessor usage in the context of 26 packages comprising 1.4 million lines of C code, and the basic result of this study shows that CPP usage in C programs is pervasive. For example, some programs such as `gzip` make have use of CPP, with one preprocessor directive found for every five lines of C code. Moreover, among all the considered packages, preprocessor directives represent at least 3% of the total line number. There is currently no exhaustive study about preprocessor usage in the Linux kernel but based on our own preliminary analysis preprocessor directives are pervasive and cover a wide spectrum of crosscutting concerns, such as debugging, configuration-dependant code or readability. For this reason, a comprehensive solution that permits the definition of extensions to C-based systems must handle CPP with care.

The basic challenge caused by CPP boils down to configuration dependent definitions of types, macros, record fields, etc. Such definitions may change depending upon the context in which they are used in a single C program. The

problem is that these definitions can change. As one example consider the following GETINT macro definition in `awful.h`:

```
#define GETINT(v) (((struct footype*)v)->one)
...
```

and `awful.c` code, which redefines GETINT from `awful.h`:

```
#include "awful.h";
void one(void *v) {
    printf ("INT = %d\n", GETINT(v));
}

#undef GETINT
#define GETINT(v) (((struct footype*)v)->two)
void other (void *v) {
    printf ("INT = %d\n", GETINT(v));
}
```

We admit that the example above appears a bit contrived, both in terms of its function as well as that no one would apply advice to it. Nonetheless, it serves as one of many heinous examples of realistic CPP usage that must be handled correctly. Specifically, lets assume that a single advice applied to both functions `one` and `other` must use the GETINT macro to access some field of `v`. For C4 this is inherently not a problem, as the C4 advice is directly inlined into the functions and the correct GETINT macro will automatically be selected. In contrast, ACC (and AspectC++) run CPP-like tools in a separate compilation phase for their advice definitions, are oblivious of these redefinitions, and therefore would use the wrong GETINT macro for `one` or the `other` functions. Ignoring CPP directives when considering the C language is a major flaw in a general context. For the AOSD community, these directives must be considered to guarantee the correctness of aspects regardless of the compilation path followed.

While the inlined C4 aspect approach more naturally handles CPP, it does not handle all situations. For example, there are cases where developers redefine a macro somewhere within a function (just look the `unwind` function in the Linux kernel/`unwind.c` file). This is one reason why we include as part of our tool set a C4 analyzer that includes heuristics to warn C4 developers of this and other possibly CPP evil.

### 3.3 Naming of Join Points

ACC provides two mechanisms for matching pointcuts with join points: simple character matching and wildcard character matching. For the former when a plain string is specified in a pointcut's declaration, ACC uses simple case-sensitive string comparison for matching with which advice is applied to all matching join points. For the latter, ACC uses "\$" and "..." as wildcard characters to enhance the matching of advice to join points.

The potential concern with this approach is to it may lead to false positives—i.e., matching to unintended join points. An example of this is when an unrelated function is renamed in a future version of the mainline code to which the advice should not be applied. While this situation can be mitigated using the `infile()` pointcut attribute, it nonetheless remains a possibility.

In contrast, the patch based approach used by B4 provides sufficient context to avoid such false positives altogether. On the flip side it may result in false negatives, but those can be easily and quickly resolved.

## 4 RELATED WORK

Coccinelle [6] is a language-based approach aimed at addressing the problem of collateral evolution – i.e., changes that must be made at call sites in response to update API semantics – in Linux device drivers. Its transformation language lets developers describe such evolution in a *semantic patch* that is then applied by a source-to-source transformation tool. Coccinelle patches are said to be semantic as opposed to regular patches since they are based on the control-flow graph of the program considered. Coccinelle differs from our approach notably by its workflow; the semantic patch is separately developed and then integrated to the mainline code (as far as we know the semantic patch inference is still nascent research). However, Coccinelle can be considered as an approach complementary to ours, with modified semantic patches as an alternative to B4.

While AOSD quantification as incorporated by ACC is based on wildcard matching that we previously considered as too brittle in the general case, it is nonetheless possibly quite powerful in narrow cases to quickly apply a crosscutting concern into a mainline program (e.g., turning a set of procedure calls into remote procedure calls). Our approach with B4 and C4 can complement AspectC-inspired approaches such as ACC by acting as their backend. In other words, ACC-based aspects would be developed in the ACC pointcut language and then woven as C4 advices at the identified join points into the mainline code base with a CPP-aware source-to-source compiler. In this way a developer could use a more traditional AOSD approach to developing a crosscutting concern, thereby serving as a stepping stone towards for Aspect-Oriented friendly developers to seamlessly contribute to the Linux Kernel.

## 5 CONCLUSION

In this paper we have presented a step towards the reconciliation of syntactic patches and aspects. Specifically we utilize an AOSD-inspired model for representing inlined extensions as woven aspects, while then codifying unwoven aspects using a patch-based pointcut representation. We thereby are finally able to close the loop of our work on C4, a domain-specific version of C to manage program extensions.

To ease the adoption of our AOSD-inspired extensions model,

it is important that it minimizes the need for new tools such as compilers and build systems. In other words, it must naturally fit together with current best practices for development workflows. In our approach, the only requirement is an update to the diff tool that understands the C4 before/after aspect semantics to produce the B4 files and an update to the patch tool set that supports the enhanced patch notation to integrate extensions in the mainline code!

## References

- [1] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn, Using aspectC to improve the modularity of path-specific customization in operating system code. SIGSOFT Softw. Eng. Notes 26 (2001) 88-98.
- [2] M.D. Ernst, G.J. Badros, and D. Notkin, An Empirical Analysis of C Preprocessor Use. IEEE Transactions on Software Engineering 28 (2002) 1146-1170.
- [3] M.E. Fiuczynski, patch (1) Considered Harmful, Tenth Workshop on Hot Topics in Operating Systems, Sante Fe, NM, 2005.
- [4] M.E. Fiuczynski, Better Tools for Kernel Evolution, Please! ;login: 30 (2005) 8-10.
- [5] M. Gong, C. Zhang, and H.-A. Jacobsen. Systems Development with AspeCt-oriented C (ACC). Connections 2007 (ECE Symposium, University of Toronto), June 2007.
- [6] Y. Padioleau, J. Lawall, G. Muller, René Rydhof Hansen. Documenting and Automating Collateral Evolutions in Linux Device Drivers. EuroSys 2008, 247-260.
- [7] A. Reynolds, M. E. Fiuczynski, and R. Grimm. On the feasibility of an AOSD approach to Linux kernel extensions. 7th Workshop on Aspects, Components, and Patterns for Infrastructure Software (2008).
- [8] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban, Advances in AOP with AspectC++, Software Methodologies, Tools and Techniques (SoMeT 2005), IOS Press, September, 2005, Tokyo, Japan.
- [9] M. Yuen, M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Making extensibility of system software practical with the C4 toolkit. Workshop on Software Engineering Properties of Languages and Aspect Technologies (2006).