

# On the feasibility of an AOSD approach to Linux kernel extensions

Alison Reynolds

Marc E. Fiuczynski

Robert Grimm

Princeton University & New York University

## Abstract

In previous work, we presented a domain-specific version of C, called C4, which was used for capturing extensions to the Linux 2.6 kernel using AOSD techniques as an alternative to the conventional patching approach [10, 19]. The focus of that work was on introducing *new* extensions represented as aspects in system software such as the Linux kernel with a focus on readability, compatibility, performance, and the preservation of existing development workflows. However, other AOSD researchers (e.g. Lohmann et al. [8]) state that “... *Linux, as a monolithic system, provides a low number of join-points for aspects and that those available were semantically ambiguous.*” This worrisome statement motivated us to study the feasibility of applying AOSD techniques to refactor existing Linux kernel extensions. To gain insight we analyzed the AOSD-ness of a large number of configurable options available in the Linux kernel and evaluated whether they could be converted into aspects—and for the AOSD fan the preliminary results are promising.

## 1 Introduction

Our foray into AOSD research was fueled by the need to introduce and maintain “out of tree” extensions for the custom Linux kernel used on PlanetLab [10]. As the systems we are interested in are predominately written in C, we designed a domain specific version of C, called C4, to let software developers more easily manage crosscutting concerns using AOSD techniques. One significant fear of ours has been the feasibility of using AOSD techniques on large existing software systems versus those that have been specifically designed to more easily permit its applicability (such as Pure etc.).

For this reason we started to study how well one might be able to refactor existing crosscutting concerns found in the Linux kernel into C4 defined aspects. Our study firstly analyzes the use of the C preprocessor and configuration options in the Linux kernel to get a better understanding of their extent of use and of their negative impact on code comprehensibility, modularity, and thus extensibility. Secondly, we assess the feasibility of refactoring configuration options and preprocessor conditional regions into aspects and aspect-oriented advice. Our analysis focuses on the predicted difficulty firstly of separating concerns and secondly of translating regions into advice in C4, an aspect-oriented extension to C. We find that Linux makes relatively high use of the C preprocessor, and that the general trend across kernel versions is a gradual increase in the percent of code affected from one version to the next. We find that the criticisms of the current mode of

extension hold true at least enough to motivate research for a better system of extension. In terms of the feasibility of refactoring current configuration-dependent code into aspect-oriented code using C4, we find that while for much of the code refactoring might be relatively straightforward, a minority of cases the C code needs to be rewritten prior to translation to C4.

To conduct this study we have built technology to syntactically and semantically analyze the kernel leveraging the xtc compiler toolkit [20], and have analyzed 20 versions of Linux (2.6.0 - 2.6.19). Specifically, we have analyzed how well the code encapsulated by Linux's configuration options matches AOSD techniques. We focus on configuration options, since each option causes code to be included/excluded and could thus potentially be replaced by an aspect. In this sense, even platform- and hardware-specific configuration options are candidates for encapsulation by aspects. The first phase of our study has just reached its conclusion and we are still analyzing the results. The preliminary results are as follows:

- Configuration dependent code amends 35% to 45% of the mainline functions, and on average about 50% of the mainline data structure definitions.
- Approximately 45% of configuration dependent code affecting a mainline function can be trivially refactored into C4 *before* or *after* aspect. Of the code that cannot be trivially refactored into before/after aspects, nearly 70% have less than 10 potentially prior statements. Further data flow and control flow analysis is necessary to determine whether the configuration dependent code relies on any of the prior statements.
- Approximately 52% of configuration dependent field introductions to aggregate types (ie., C structs) are either at the beginning (top) or end (bottom) of a mainline structure, and therefore can be trivially refactored into a C4 introduction. Of the remaining field declarations, further data use analysis is necessary to determine whether their position within the structure is a cache optimization or simply for readability.

In Section 2, we provide background on the current model for Linux extension and in what way we wish to improve on this model using C4. In Section 3, we describe in more detail the focus of our research along with our approach to data collection and the corresponding tools. In Section 4, we present the data we gathered regarding Linux extension as it is currently done. We use that data both to generally improve our understanding of how the preprocessor and configuration options are currently used to extend Linux. We also discuss the potential for refactoring Linux to use aspect-oriented design in terms of the expected difficulty firstly of separating concerns, and secondly of translating preprocessor code to the AOP syntax of C4, an aspect-oriented extension to C.

## 2 Background

This section first briefly discusses the current model for extending the Linux kernel—leveraging CPP based conditionally compiled code. It then provides a brief overview of the C4 language features and in what way they can be easily used to

refactor/restructure CPP conditional code into aspects, thereby opening up the opportunity to leverage more powerful means of extending the Linux kernel.

## 2.1 CPP based extension model

Configuration of the kernel is controlled by configuration options built into the kernel's configuration system. Each configuration option is represented by a variable prefixed with "CONFIG\_" as in CONFIG\_FOO [2]. From one version of Linux to another, many of the changes made to the kernel code are in the form of new configuration options and the addition or modification of configuration dependent code.

When the kernel is built, the values of configuration options dictate what files are compiled and which code within those files is included. Configuration options manipulate the code base through preprocessor directives. The preprocessor directives include #define, #undef, #include, #pragma, #line, and the six conditional directives: #if, #ifdef, #ifndef, #else, #elif, and #endif. The conditional directives let blocks of code be included or excluded from the code at compile time.

The Linux coding style provides guidelines on the use of the preprocessor directives [2]. Specifically, it discourages putting #ifdef directives directly into the baseline C source, as shown on the top portion in the code example shown below. Rather, the suggested alternative is to define foo() to do nothing if CONFIG\_FOO is not set such that it can be unconditionally invoked from the baseline scheduler() code, as shown in the bottom portion below.

```
void scheduler() {
    ...
#ifdef CONFIG_FOO
    foo();
#endif
...}

#ifdef CONFIG_FOO
static int foo(void) { /* extension specific code here */ }
#else
static inline int foo(void) { /* no code here */ }
#endif

void scheduler() {
    ...
    foo();
...}
```

The resulting source code appears cleaner and more readable by the omission of the CPP conditional directives. However, lacking sophisticated IDE tools, it is non-trivial for developers looking at such code to discern what code is part of the baseline and what is conditionally compiled in. We have argued previously that this problem is magnified when dealing with widely crosscutting extensions that need to be added into the kernel.

## 2.2 C4 language features and its mapping onto CPP conditional directives

This section provides a brief overview of the C4 language and how it can be leveraged to refactor CPP conditional code into an aspect. C4 is an aspect-oriented extension to the C language designed to let developers refactor code inline, while also gaining the modularity and other benefits of an aspect-oriented architecture. To do both of these at once, the C4 toolkit (still under development) will let developers define advice inline using

the woven C4 syntax. An unweaver tool could then be used to extract that aspect code from the mainline code and place it in a separate aspect file using the grammar of AspectC. Conversely, a weaver tool could take AspectC and apply it to the mainline C code by injecting the woven C4 code into the appropriate locations.

All aspect code in the C4 language is specified in the same way, using the keyword aspect, followed by the name of the aspect and then a block containing the aspect code as in: aspect(foo) { /\* advice code here \*/};

Each of these blocks of code is called an *advice*. An advice may be used to *introduce* a new global variable, structure or union type definition, function, new fields to a structure or union, or to change the processing of a function. An advice serving either of the first two of these purposes is called an *introduction*, and an example of each is given below:

```
aspect(CONFIG_FOO) {
    /* New type */
    typedef struct bar { int bar_int; } bartype;
    /* Global variable */
    float foo_float;
    /* New function */
    void foo_function(int d){... printf("In foo_function: %d\n", d); }
};
```

Using the CPP based approach, the above C4 based *introductions* would just be encapsulated using a conventional #ifdef CONFIG\_FOO ... #endif sequence.

Advice introducing fields to a structure (or union) using both the C4 and CPP approach is respectively shown below:

<pre>struct point {     float x,y;     aspect (CONFIG_FOO) {         float z;     }; };</pre>	<pre>struct point {     float x,y; #ifdef CONFIG_FOO     float z; #endif /* CONFIG_FOO */ };</pre>
---	--

Finally, an example of how advice acting on a function might be specified inline using C4 and CPP is given below:

<pre>int main(void) {     aspect (CONFIG_FOO){         printf("before advice\n");     };     printf("baseline code\n");     return 0;     aspect (CONFIG_FOO) {         printf("after advice\n");     }; };</pre>	<pre>int main(void) { #ifdef CONFIG_FOO     printf("before\n"); #endif /* CONFIG_FOO */     printf("baseline code\n"); #ifdef CONFIG_FOO     printf("after\n"); #endif     return 0; };</pre>
---	---

## 3 Approach

### 3.1 Study Approach

Given that the Linux kernel's current extension approach consists primarily of leveraging preprocessor conditional directives to create configuration-dependent code, our first step is to understand this more quantitatively. We do this in part by counting directly the numbers of configuration options, uses of preprocessor directives, conditional regions and expressions, etc., as well as the sizes of an extension's code regions (i.e., the blocks of code encapsulated within a conditional directive). We additionally consider cases in which baseline code is affected by conditionals, for instance in the case of a function call where the

function is defined within a conditional region (as in the example of Love's coding guidelines [2]). We also look at these statistics across kernel versions to get an idea of how the use of these features has changed over time, and how extensively they have been used to evolve Linux.

The next step is to explore the complexity of preprocessor use. This is done in a number of ways. First is by looking at the complexity of the conditional expressions themselves in terms of the numbers configuration options on which they are dependent. Additional factors considered include nested regions, the distribution of regions with the same variables or expressions among different files, distribution of similar code between different regions, and others. Specifically, our focus on conditional regions as defined by the preprocessor directives is driven by the fact that code found in regions associated with the same conditional expression essentially could correspond to an aspect-oriented concern. With this in mind, some of the data collected with respect to the extent and complexity of preprocessor use, such as the degree to which code of a single concern is spread through different regions and files in the kernel, has aspect-oriented implications. Such data is indicative of the extent to which various concerns are crosscutting and of importance since a major argument for an aspect-based system of evolution is that aspects have the advantage of improving the modularity of crosscutting concerns.

Finally, in addition to looking at the implications of the extensiveness and complexity of preprocessor use for aspect-oriented design, we collect data that focuses more directly on how directly code in conditional regions might be adapted into aspect code. More specifically, we consider how nearly the regions of conditionally compiled in code matches with the aspect-oriented language features of the C4 language by looking at the embedding of particular commands such as declarations of functions, structures, and unions within conditional regions, and the contextual location of conditional regions.

## 3.2 Data collection approach

Our data-collection software was implemented in two major components for syntactic and semantic analysis. The component for syntactic analysis was used to collect the data that required only a textual look at the Linux code, while the semantic analysis component was used to collect context-based information. For the former we developed our own tool (called *c-color*), while for the latter we developed a new module leveraging the *eXTensible C (xtc)* compiler toolkit [20]. We collected data on twenty versions of the Linux kernel from Version 2.6.0 to Version 2.6.19. For each version of the kernel, we used the default kernel configuration for an *i386* kernel. For data-collection on the Linux kernel, the analysis components mentioned above were invoked directly from the kernel's *makefile*. For our *c-color* tool we directly manipulated the original source file, while the *xtc*-based tool was run on each file after it had been preprocessed by *cpp*. Data collection was done over all of the files reached on a kernel build using the command "*make bzImage*" --- i.e., not the device drivers, but nonetheless this was 500K-750K lines of C code.

## 4 Results summary

The goals of this section are to gain a qualitative understanding of the evolution of the Linux kernel using the current extension model using configuration options and C preprocessor directives and to assess the extent to which some of the claims about and criticisms of the current system hold true. We begin by presenting our high-level data on the evolution of the Linux kernel across the

twenty versions of the kernel that we studied (2.6.0 to 2.6.19). From there, we move to an analysis of the extent of use of preprocessor directives in general, specifically how Linux' use of them compares to their use in other C programs. Then we narrow our focus to look more closely at the use of the conditional directives, considering both their frequency of use, and the extent of the code embedded in the regions they define. In the next sections, we present data on configuration options, the preprocessor conditional expressions that contain them, and the regions associated with such conditional expressions. We look then at the code dependent on configuration options both in general, and in terms of specific language constructs (functions definitions, function calls, type definitions, etc.). We finish by presenting our data regarding some of the criticisms of the current model of extension, namely the scattering of code related to particular concerns throughout the code, and the cluttering of the source code with a large number of conditional regions.

### 4.1 High-level summary on the evolution of the Linux kernel

We observed that on average the code kernel (ignoring device drivers) grows approximately 1.5% from version to version, yet recent trends indicate that the pace of change is increasing. To get a sense of how the use of the preprocessor directives in the Linux kernel compares to its use in other open source software packages, we can compare our results to those obtained by Ernst et. al. [7]. Ernst et. al. did an empirical study of the use of the C preprocessor in 26 popular C programs. There are a couple notable things about how the data we gathered on the Linux kernel compares to that gathered by Ernst, et. al. First, we find the total incidence of all preprocessor directives to be about 11.4%, which is significantly above the average incidence of 8.4% of the other C programs studied. This puts Linux among the programs analyzed by Ernst that had the highest preprocessor use. We observed that Linux has a high relative usage of *#define* for which the usage percentages is over 230% of the averages for the packages Ernst et. al. studied. The conditional directives actually had a relatively low usage at only about 63% of the average in the other packages.

Overall preprocessor use in the Linux kernel seems to be on the high end as compared to other C programs, the relatively low usage of the conditional directives seems at first to be an aberration. Given the coding guidelines for Linux extensions that appear in Love's book on Linux kernel development [2] – see section 2 – this becomes less of a surprise. Recall that Love's guidelines state that preprocessor directives should not be written directly into the C source. He suggests alternatively that for conditionally dependent code, different implementations of a function should be defined in conditional regions so that the function can then be called from mainline code without being embedded in a region. As function calls should exist in greater numbers than function definitions, and function calls in the Linux kernel should not be within regions, the Love guidelines, if followed, would be expected to lower the usage of the conditional directives. As the Love guidelines are specific to Linux, and not general to C coding using *cpp* in general, we believe they are the cause for this relatively low use of the conditional directives within the baseline code. Note that while the incidence of the directives themselves is low, the total incidence of conditionally-dependent code (including function calls of conditionally dependent functions) could still be at least as high in Linux as it is in the other C programs studied by Ernst et. al..

## 4.2 Summary of Conditional Directives and Configuration Options

In this section, we narrow our focus further to consider specifically those conditional expressions and their associated regions that involve one or more configuration option. We look first at the configuration options themselves, then at the conditional expressions containing them, and finally at the regions associated with a configuration-dependent conditional expression. For each of these, we analyze how quickly they change in number from version-to-version in comparison to how quickly the size of the kernel changes.

In terms of the number of unique CONFIG variables found in preprocessor conditional expressions varied somewhat from version to version with the lowest number being 534 in 2.6.9, and the highest number being 691 in Version 2.6.19. Generally we found that there was no consistent increase in the number of CONFIG variables used in conditional expressions from version to version. The number of CONFIG variables dipped at first, though after 2.6.9, the general trend was an increase that accelerated for the last few versions. We expect that this trend is a product of the types of development occurring between different set of Linux versions. Specifically, between 2.6.0 and 2.6.8, we consider that the 2.6 version of the kernel was still in its beta development phase and did not see widespread adoption by the various Linux distributions as a replacement for the Version 2.4 kernel. After the 2.6.8.1 release, it appears that contributions from these distributions might explain why it is at this point that a trend of fairly consistent increase in the number of configuration options in the kernel code begins. Anecdotal experience informs us that compared to the version-to-version changes made between some of the earlier releases, those made from version-to-version between around Version 2.6.16 and 2.6.19 were more major changes, which could help explain the more rapid increase in the number of CONFIG variables we observed from the results of our analysis, though we do not know which specific changes would have caused this increase.

Next, we looked at conditional expressions related to configurations, which are defined as any conditional preprocessor expression that contain at least one CONFIG variable. The number of unique CONFIG expressions found by our tool varied from a minimum of 811 in 2.6.9 to a maximum of 1100 in the 2.6.19 kernel. We observed growth trends similar to that seen with the number of CONFIG variables. The number of unique CONFIG expressions remains fairly stable, with some version-to-version fluctuation from about 2.6.0 to 2.6.10, from which point there is a consistently large rate of growth until Version 2.6.19. As with the growth in CONFIG variables, we expect that this trend is related to the different phases of Linux development where we might think of the releases up to around Version 2.6.8 as occurring during development phase, and the subsequent releases including increasing changes contributed by distributions that start to switch to the 2.6 kernel.

Finally, we looked at CONFIG regions, which are defined as the code blocks delimited by the preprocessor conditional directives where the associated conditional expression contains a CONFIG expression. The number of regions found ranged from 2,415 in Version 2.6.4 to 3,663 in Version 2.6.19. Compared to the data for growth in the numbers of CONFIG variables and CONFIG expressions, we find that overall the growth in the number of CONFIG regions is much more consistent with the overall growth in the size of the kernel (in terms of the number of processing

lines). What this suggests is that while an increase in the number of configuration options and the creation of new CONFIG expressions did not happen very consistently until about Version 2.6.8, the addition of new CONFIG regions seems to have occurred in all phases. The one place where the growth in the number of regions seems to digress substantially from the growth of the kernel in general is during the last several releases, from around Version 2.6.16 to Version 2.6.19. We explain this digression as again the product of the relatively major changes made to the kernel between each of these versions.

## 4.3 Code Affected by Configuration Options

The summary of the CONFIG data presented in the previous section has a number of implications for AOSD. Firstly, it provides quantitative proof that the use of configuration options to affect the code via the preprocessor directives is growing along with kernel size. Secondly, it suggests that growth in the first phase largely involves adding new code relating to existing concerns (as represented by configuration options) while later development seems to involve increased addition of new concerns. Finally, it suggests that as development continues in later versions of the kernel, we are likely to find more and more of the mainline code in C files indirectly and not visibly apparent for the uninitiated programmer to contain configuration-dependent code.

To validate this we investigated the percentages of the kernel code that are dependent on configuration options either directly or indirectly. We first consider how much code is affected in general, then focus on the fraction of function definitions, function calls, structure type definitions, and union type definitions. Our results should be seen as a lower bounds on the fractions of mainline that is infected with configuration-dependent code. This is because our analysis using `xtc` occurs *after* running `cpp` on the original source code, and so we do not have complete information on conditional regions that did not evaluate to true when the preprocessor was invoked. (In on-going work, we are extending `xtc` to handle unpreprocessed code.) Our results therefore reflect only the fraction of these various types of commands that are both within CONFIG regions in the initial source code and remain in the code after preprocessing.

In terms of how much of the overall kernel code is configuration dependent we found on average 8.8% was contained in CONFIG regions with a trend of gradual increase from version-to-version. This indicates that though the previous section suggested that the number of CONFIG regions does not seem to grow consistently relative to the size of the kernel, the relative amount of code in these regions does.

Next we evaluated the use of configuration-dependent function definitions (i.e., introductions) from the mainline code base. We found that across the kernel versions about 8-10% of the kernel's functions were defined within CONFIG regions. Note that our tools did not account for function macro definitions – as in `#define foo(){/*body*/}`. We expect the actual percentage of configuration-dependent functions will therefore be greater than what we found, and thus our 8-10% numbers serve as a lower bound.

We then quantified the use cases (calls) to such functions. Recall the coding guidelines for Linux kernel development outlined by Love suggests that mainline code unconditionally invoke configuration-dependent functions. On average across the kernel versions nearly 10% of all direct function calls invoke

configuration-dependent code. This is a lower bound and gives some idea of the effect, indirectly on the mainline code. For the estimated one in ten function calls to configuration-dependent code, it may be unclear to a developer what it does (if anything, as it is defined as a noop if it is not CONFIG'd into the kernel).

We also examined configuration-dependent definitions of structures and their fields. We found that across all versions nearly 7% of all structure definitions were introduced by configuration-dependent code, and that over 8% of the mainline structures contain fields that are configuration dependent.

Finally, we evaluated the extent to which crosscutting concerns exist in the kernel as represented by configuration-dependent code. We observed that across the kernels on average 25% of the CONFIG variables appeared in more than 5 files, 10% in over ten, 5% in over twenty, and 1% in over 100 files.

#### 4.4 Preprocessor Use and AOSD

Our primary goal of collecting the data presented so far is to assess how easily configuration-dependent (CONFIG) code could be refactored into aspect code. As a part of understanding whether AOSD might be a model that could be advantageously used in the Linux kernel, we want to know how closely existing code might match an AOSD approach. This question can be looked at in terms of how easily current code might be refactored into our C4 aspect language. A characteristic of the current model for Linux extension is that there already is some conceptual separation of concerns. Often, a new concern is constructed as a new configuration option, and the preprocessor conditional directives are used to define the code related to that concern. Recall that there is some successful precedent for refactoring code using the C preprocessor and configuration options into aspect code. In the Lohmann, et. al. study of eCos, existing code was refactored into aspect code by translating configuration options into aspects [8]. Moreover, they found the code easy to refactor because they found concerns to be quite separate [8].

We found the same to be true for Linux: 95% of the CONFIG regions are controlled solely by CONFIG variables with 82% of those by a single CONFIG variable. We also found that over 86% of the CONFIG regions had a nesting depth of 1. Approximately 65% of the CONFIG regions would be considered as introductions, while the other 35% of the regions fall within existing functions and would need to be captured as advice. CONFIG regions defining global structures, functions, variables can trivially be converted to C4 introductions, assuming their relative position within a particular file is independent of CPP macros. We observed that over 51% of fields added by configuration-dependent code in structures are also trivially expressed as C4 introductions. Further analysis is required to understand whether the other 49% of field introductions are position dependent relative to other fields (e.g., optimized for better cache hit rates). Finally, we found that nearly 24% of the configuration-dependent code recognized by our tools – recall that we are not handling macro-based definitions – can be trivially converted to C4 before/after or around advice. Further data and control flow analysis is required to ascertain how much more of the current code base can be trivially converted as well as to gain further insight into the refactoring complexity to support a AOSD based extension model. While no clear conclusions can be drawn so far, we find these results encouraging so far!

### 5 Conclusion

In this paper, we have analyzed the use of the C preprocessor and configuration options in extending the Linux kernel. We believe

that our assessment of the extent of preprocessor use supports claims of its obfuscation of source code, indirect impact on mainline code, and scattering of concerns sufficient to motivate a consideration of alternative systems of extension. Our analysis of the potential refactorability of current preprocessor-based code into aspect code, specifically using the language features of C4 led us to the conclusion that the majority of translation into aspect code could be straightforward, some explicit rewriting (or at least reordering) of code prior to translation will be necessary. We expect the separation of concerns, or alternatively, development of a mechanism for aspect composition, to be among the most substantial barriers to a refactoring to AOSD. The other big challenge will be the restructuring of functions to enable the join-points for advice acting on it to be specified. Our study so far does not provide the control-flow analysis that would be needed to more thoroughly adequately assess the extent to which these issues are likely to hamper the use of AOSD in Linux.

### References

- [1] O. Koren, A study of the Linux kernel evolution. SIGOPS Oper. Syst. Rev. 40 (2006) 110-112.
- [2] R. Love, Linux Kernel Development, Novell Press, Indianapolis, Ind., 2005.
- [3] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar, Aspect-Oriented Programming, In Proceedings of the 11th European Conference In Object-Oriented Programming LNCS, Jyväskylä, Finland, 1997
- [4] K.N. King, C programming: a modern approach, Norton, New York, 1996.
- [5] J. Pranevich, Kernel Korner: Contributing to the Linux Kernel--The Linux Configuration System. Linux Journal (2000) 11.
- [6] Linux Journal Staff, Linux Distributions Compared. Linux Journal (1996) 1.
- [7] 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.
- [8] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat, A quantitative analysis of aspects in the eCos kernel, Proceedings of the 2006 EuroSys conference, ACM Press, Leuven, Belgium, 2006.
- [9] M.E. Fiuczynski, R. Grimm, and D. Walker, Managing OS Extensibility via Aspect-Oriented Programming Technology, CSR-PDOS, 2006.
- [10] M.E. Fiuczynski, patch (1) Considered Harmful, Tenth Workshop on Hot Topics in Operating Systems, Sante Fe, NM, 2005.
- [11] 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.
- [12] 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, 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 196-204.
- [13] M.E. Fiuczynski, Better Tools for Kernel Evolution, Please! ;login: 30 (2005) 8-10.
- [14] C. Matthews, O. Stampflee, Y. Coady, J. Appavoo, M.E. Fiuczynski, and R. Grimm, HEY... You got your Paradigm in my Operating System!, Proceedings of the 2nd ECOOP

Workshop on Programming Languages and Operating Systems, Glasgow, UK, 2005.

- [15] O. Krieger, M. Auslander, B. Rosenburg, R.W. Wisniewski, J. Xenidis, D.D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, K42: building a complete operating system, Proceedings of the 2006 EuroSys conference, ACM Press, Leuven, Belgium, 2006.
- [16] D. Da Silva, O. Krieger, R.W. Wisniewski, A. Waterland, D. Tam, and A. Baumann, K42: an infrastructure for operating system research. SIGOPS Oper. Syst. Rev. 40 (2006) 34-42.
- [17] Á. Balogh, and Z. Csörnyei, SysObjC: C extension for development of object-oriented operating systems, Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems, ACM Press, San Jose, California, 2006.
- [18] M.Y. Coady, Improving Evolvability of Operating Systems with AspectC, Computer Science, University of British Columbia, 2003, pp. 135.
- [19] M.E. Fiuczynski, and M. Yuen, C4 Toolkit, 2007.
- [20] <http://cs.nyu.edu/rgrimm/xtc/>.