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

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

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

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

Abstract

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

1 Introduction

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

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

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

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

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

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

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

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

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

2 Linux Scheduling Points

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

2.1 Bossa events in Linux

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

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

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

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

2.2 Automating Linux instrumentation

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

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

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

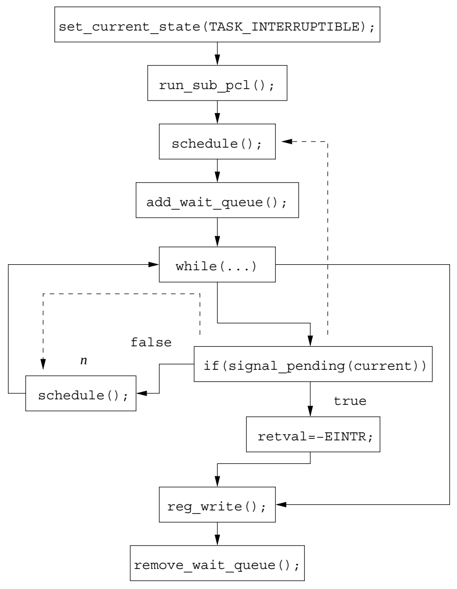

Figure 2: CFG for excerpt of PCILynx driver

3 Describing Crosscuts using Temporal Logic

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

3.1 Rewrite rules

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

We use rewrite rules of the form:

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

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

An example of such a rule is:

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

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

3.2 Predicates

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

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

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

3.3 Describing CFG nodes and paths

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

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

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

A judgment of the form $n \vdash A(\phi_1 \mathsf{U}\, \phi_2)$ is satisfied if for each path beginning at n, every node along the path satisfies ϕ_1 *until* a node n' satisfying ϕ_2 is reached, or the path loops infinitely and every node in the path satisfies ϕ_1.[2] The node n' need not satisfy ϕ_1. For Bossa, we are primarily interested in analyzing nodes whose execution precedes a call to `schedule()`, and thus we consider paths that end, rather than begin, at the given node n. Such a backwards search is expressed by the operator $\triangle$; thus, we typically use judgments of the form $n \vdash A\triangle(\phi_1 \mathsf{U}\, \phi_2)$ rather than $n \vdash A(\phi_1 \mathsf{U}\, \phi_2)$.

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

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

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

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

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

require the existence of a path whose nodes satisfy the subformulas. Here, however, looping is not allowed; the path must contain a node that satisfies ϕ_2. As an example, to express that there *exists* a backward path from node n, that eventually reaches a node satisfying $\mathsf{change\_to\_running}() \vee \mathsf{Entry}()$, we use the judgment:

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

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

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

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

3.4 Instrumenting calls to `schedule()`

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

$$n : (\texttt{schedule();}) \Rightarrow \{\texttt{BOSSA\_YIELD\_SYSTEM\_IMMEDIATE(fn\_name,current);}n\}$$

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

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

We express these conditions as follows:

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

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

$$n : (\texttt{schedule();}) \Rightarrow \{\texttt{BOSSA\_CHECK\_PENDING\_SIGNAL(fn\_name,current);}n\}$$

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

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

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

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

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

4 Related Work

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

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

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

5 Conclusion

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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