Annotations in Operating Systems with Custom AspectC++ Attributes

Birte Friesel
Embedded System Software Group
TU Dortmund
Dortmund, Germany
birte.friesel@uos.de

Markus Buschhoff
Embedded System Software Group
TU Dortmund
Dortmund, Germany
markus.buschhoff@tu-dortmund.de

Olaf Spinczyk
Embedded System Software Group
TU Dortmund
Dortmund, Germany
olaf.spinczyk@tu-dortmund.de

Abstract

Aspect Oriented Programming (AOP) supports the modular implementation of crosscutting concerns, which are woven into program parts designated by pointcuts, e.g. calls to specific functions. The release of AspectC++ 2.2 introduces the ability to express pointcuts based on C++11-style attributes as well as the definition of custom attributes for annotation purposes. In this paper, we propose the use of such attributes for operating system development. We cover three examples: Replacing non-portable compiler attributes and extending portable ones with domain-specific knowledge, providing implementation-independent joinpoint APIs to core operating system functions, and compile-time support for co-development of source code and corresponding models. We discuss the implementation effort and code size overhead of our ideas on the operating systems CocoOS and RIOT and show that annotations with custom attributes are a helpful addition for system development.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Operating systems*; Reusability; • **Computer systems organization** → *Embedded software*.

ACM Reference Format:
Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. 2017. Annotations in Operating Systems with Custom AspectC++ Attributes. In *Proceedings of PLOS'17: 9th Workshop on Programming Languages and Operating Systems (PLOS'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3144555.3144561

PLOS'17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5153-9/17/10…$15.00
https://doi.org/10.1145/3144555.3144561

1 Introduction

Modern programming languages such as C#, C++[1] and Java[2] allow for the use of attributes to annotate language elements, e.g. functions, methods and data structures. While C# and Java permit declaration of custom attributes and usage of introspection mechanisms on them, C++ has a fixed set of standard attributes plus compiler-specific attributes that merely denote special handling of data and functions. By that, formerly used compiler-specific syntax, e.g. `__attribute__` in GNU compilers, was replaced by a standardized syntax. Anyhow, C++17 specifies just seven standard attributes, compared to around 90 in GCC 8 (architecture-specific attributes excluded), 110 in Clang 6 and 21 in Microsoft Visual C++ 2017.

However, adding user-defined functionality to attributes is still a missing feature, probably due to the fact that C++ provides neither fully-fledged introspection nor static function alteration. The latter is available in aspect-oriented languages with compile-time weaving, so it is a logical step to enhance attributes by using AspectC++.

In the following chapters we present an extension of AspectC++ which is available since version 2.2. It supports binding aspects to code locations denoted by C++ attributes, thus enabling the enhancement and alteration of these locations by code declared in an external aspect.

After giving a brief introduction to the AspectC++ language in Section 2, we will discuss three use cases for attribute-bound aspects. We will show how to replace non-portable compiler attributes and enhance standard attributes in Section 3. In Section 4, we will describe how to use attributes to provide a generic, OS-independent interface to system core functions. This allows e.g. for the implementation of network data accounting without knowledge of the system's network stack. Finally, in Section 5 we will show how source code and model co-development (a subject related to model-driven development) can benefit from AspectC++ attributes.

After presenting the use cases we will discuss their implementation effort and overhead in Section 6, examine related work in Section 7 and give closing remarks in Section 8.

[1] since C++11
[2] since Java 5

2 AOP with AspectC++

AspectC++ is an *aspect-oriented programming* (AOP) language based on C++ [13]. It is designed as a language extension: AspectC++ code is C++ extended with aspects and pointcuts (see below). The aspect code resides in aspect-header files and gets "woven" into the target C++ code by the AspectC++ compiler. The result of this operation is ordinary C++ code, which is passed on to the compiler for the target architecture, e.g. g++ or clang. Due to the definition of aspects in separate header files, the remaining code is also valid C++ and can be compiled without AspectC++ (though aspects will be ignored in this case).

Software projects can benefit from AOP in several ways. The following (incomplete) list shows the most important advantages [3]:

- Crosscutting concerns: Fragments of crosscutting code can be implemented in separate modules (instead of inserting code fragments in numerous locations).
- Inversion of control: Modules can bind themselves to code locations.
- Feature isolation for software product lines: The combination of the aforementioned points allows for activation or deactivation of features without code alteration, just by choosing whether to compile a feature or not, avoiding the so-called "#ifdef hell".

AOP with AspectC++ additionally allows for efficient weaving, as it binds statically and creates woven C++ code. This allows the C++ compiler to use optimizations.

Examples for crosscutting concerns include sanity checks, fault tolerance measures or authorization mechanisms, all of which would require changes to each affected function or variable without AOP. In an AOP-based implementation, each kind of change only needs to be defined once (in an *advice*) and can then be applied to an arbitrary number of code parts designated by *pointcuts*. Sets of related advices can be grouped within *aspects* in the same way that methods belong to classes. The following glossary briefly describes the most important terms of AOP.

joinpoint a distinct (in case of AspectC++ usually static) code location where an advice can weave code

pointcut a denotation of code locations in a declarative syntax

advice a code fragment to be woven into a set of locations specified by a pointcut

aspect a set of advices implementing a common crosscutting concern

As of version 2.2, AspectC++ allows pointcuts to be expressed on C++ attributes as well as the definition of custom attributes. This allows for a change in the development paradigm: Instead of using pointcuts that *directly address* functions or variables they affect, it is now possible for developers to *express intentions* by annotating code fragments with attributes which are matched by attribute pointcuts and implemented by aspects.

```
// sched.h (C++ header)
[[OS::log]] int sched_run(void);

// os.ah (Aspect header)
namespace OS {
  attribute log();
};
aspect PrintfDebugging {
  advice execution(OS::log()) : before() {
    printf("+␣%s\n", tjp->signature());
  }
}
```

Figure 1. Defining and using the `OS::log` attribute for printf debugging.

Figure 1 shows an example of this idiom. Here, a custom *log* attribute is defined, which is meant to output a line containing the function signature each time an annotated function is called, thus acting as a primitive execution trace. Enhancing the execution trace, e.g. by also logging function parameters, only requires changes to the aspect and not to annotated functions.

The aspect causes the printf statement to be executed *before* each *execution* of functions annotated with the attribute `OS::log`. Other pointcut functions and advice types used in this paper include *call* and *within*, which match function calls and allow filtering them by the function they originate from, and *after* and *around* to specify that an advice should run after or instead of a function. For a detailed introduction to AspectC++, we refer to Spinczyk et al. [13].

In the following sections we will show how these attribute-based annotations can be used to aid in the development of operating systems written in C and C++.

3 Portable Compiler Attributes

The least intrusive approach towards AOP-based annotations is augmenting attributes with custom behaviour, such as partially allowing calls to deprecated functions or fine-grained interrupt control.

C++14 specifies that it is discouraged to use entities marked as `deprecated`, which is usually implemented as a compile-time warning or error. However, systems may also contain legacy components, and often only non-legacy code should be prohibited from calling deprecated functions. C++ alone cannot handle this, but AspectC++ (with a newly introduced `Attr::legacy` attribute and pointcuts limited to non-legacy code paths) can.

It also allows developers to enhance attributes with system-specific knowledge, e.g. about available methods for debug output. So, apart from compile-time warnings or errors, it

```
advice call(deprecated()) &&
    !within(Attr::legacy()) : before() {
  uart << "Deprecated␣function␣call:␣"
    << tjp->signature() << endl;
}
```

Figure 2. Runtime logging of calls to deprecated functions inside non-legacy code via UART.

is also possible to log deprecated function calls at runtime (see Figure 2 for an example). Similar behaviour can be implemented for deprecated variables.

Attribute advices also come in handy for interrupt control. Compilers do not generally provide attributes which disable interrupts during execution of a function. Since interrupt handling is not only architecture-, but also use case-dependent (e.g. it may be better to only block a subset of interrupts in some cases), this is not surprising.

Hence, most developers use explicit interrupt control to account for this case. It is based on two primitives: Disabling interrupts with `status = irq_disable()` while saving the previous interrupt configuration in `status`, and using `irq_restore(status)` before each return statement to restore it. The IRQ methods are provided by the developer, which has the convenient side-effect of allowing for fine-grained interrupt control, e.g. by only disabling a specific interrupt source.

This can be simplified by combining function annotation with user-defined interrupt control. By creating an advice which disables interrupts, calls the original function and then restores interrupts using the methods shown above, explicit interrupt control can be removed from functions and replaced by a user-defined attribute. This also makes it impossible to forget an `irq_restore` call in functions with multiple return statements. Fine-grained interrupt control can be achieved by defining multiple attributes and corresponding advices.

On a more abstract level, this approach can also be used for synchronization in general: The attribute specifies what to synchronize and the aspect implements it by whatever means is appropriate – e.g. by globally disabling interrupts or (in systems with software-based interrupt synchronization, such as the top and bottom halves model) locking the bottom half.

Finally, AspectC++ also supports pointcuts on variable read/write access and references. Annotating variables with attributes such as `FaultTolerance::redundant` allows for the optional implementation of fault-tolerance aspects. Compared to pointcuts on variable names, this has the advantage that a variable's fault tolerance status is immediately apparent from its declaration and can be leveraged by IDEs with attribute support.

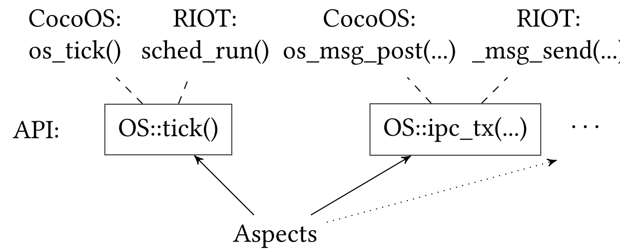


Figure 3. Using attributes to provide an OS-independent joinpoint API.

However, other types of compiler and language attributes cannot easily be augmented. As AspectC++ transforms source code to C++ and not to a low-level machine language, it cannot alter the code generation employed by the compiler and hence cannot change alignment, optimization flags or calling conventions. Making this kind of changes requires support for attribute replacement (i.e., insertion of attributes by aspects). We regard this as future work.

4 OS-Independent Joinpoint APIs

Attributes can also provide an OS-independent interface to system functions, thus increasing portability of features and modules implemented in advices – in the best case, an advice can be used on several systems without having to be aware of any system-specific details.

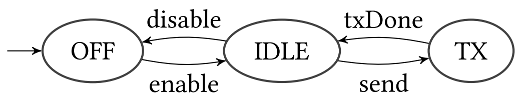
For instance, by annotating the functions responsible for transmitting an IPC message, a single IPC accounting aspect can be used both for CocoOS and RIOT. Figure 3 shows a part of the API for these systems.

By encoding implementation details in attribute parameters[3] (e.g. the index of the function argument containing the message length or, in case it is part of a struct, a pointer to a static function which returns the message length), it is also possible to account the number of transmitted bytes. Since aspects are woven statically, there is zero overhead even if a function pointer needs to be specified – as it is a constant pointer to a static function, it can always be inlined.

In a similar fashion, systems can provide annotated function stubs for modules and drivers to plug into. For instance, by providing annotated (empty) send and receive functions, arbitrary network drivers can be used by targeting the stubs with pointcuts and replacing them with the driver-provided functions. It is also possible to target annotated non-stub functions, e.g. to replace a simple default scheduler with a more sophisticated one.

This is especially useful for (but not limited to) aspect-oriented modules, such as the highly configurable CiAO/IP stack [2]: Adding it to a system only requires two additional aspects specifying the send and receive stubs to replace.

[3]We have a working prototype for parameter support in the upcoming 2.3 release, but note that it is not yet present in AspectC++ 2.2




```
enum states = {OFF, IDLE, TX};
[[Model::transition(OFF, IDLE)]]
  void enable();
[[Model::transition(IDLE, OFF)]]
  void disable();
[[Model::transition(IDLE, TX)]]
  void send(char *data, uint8_t len);
[[Model::transition(TX, IDLE)]]
  [[Driver::interrupt]]
  void txDone();
```

Figure 4. A hardware model (top) and corresponding annotated driver functions (bottom).

So, an annotation-based interface allows for a change of paradigm. Traditionally, modules provide an interface for the OS, but each module has different ideas about function names and signatures. It is the task of the operating system (and its developers) to call them properly. With an attribute API, this responsibility is moved to the modules, so now any module can easily be plugged into a system after the one-time effort of creating the annotation API.

5 Source Code and Model Co-Development

System components frequently correspond to models. For instance, a scheduler is based on task states and transitions, and a peripheral driver needs a model of hardware states and allowed transitions between them to work.

Attributes allow models to be embedded into source code and coupled to the corresponding implementation. Before considering this approach in general, we will first demonstrate how it works by using a radio driver as an example.

Like most hardware, the behaviour of radios can be modelled using deterministic finite automata (DFA). There are several states (e.g. off, idle and transmitting) and transitions between them, which are either invoked by calling a driver function (e.g. *enable* and *disable* to turn the radio on/off or *send* to start a transmission) or signalled by an interrupt (e.g. a "transmit complete" interrupt, which in turn calls a driver function such as *txDone*). By defining the states in an enum variable and annotating functions with `Model::transition(`*origin*, *destination*`)`, this model can easily be coupled to the driver, as shown in Figure 4.

This allows aspects to leverage the model without needing to know the names or arguments of functions. For instance, by starting a timer after each transition with $destination = \text{TX}$ and $origin \neq \text{TX}$, and stopping it after each transition with $origin = \text{TX}$ and $destination \neq \text{TX}$, the total transmission time can be accounted and used to assess radio protocol efficiency or other metrics.

Similarly, energy consumption of function calls can be expressed by means of `Transition::energy(...)` annotations. Thanks to the availability of functions and attributes in the machine-readable AspectC++ *project model*, automatic model refinement is also possible. In fact, we have already successfully used a similar approach[4] to iterate over all hardware states and transitions with a test application, measure energy and timing data, and then add energy accounting aspects to corresponding driver functions – all without a single human intervention (paper to be submitted).

Arbitrary DFA-based models can be embedded this way: If a function corresponds to multiple transitions (e.g. because it toggles a feature), it can be annotated with several `Model::transition` attributes.

Generally speaking, this approach is similar to *model driven development* (MDD), which also formalizes the coupling between code and model. However, as the name suggests, MDD follows a "model first, implementation later" approach – in fact, one of its main goals is deriving the implementation from the model with as little manually written code as possible. This is usually done by starting with a coarse model and using multi-stage model refinement to generate the implementation.

Considering the longevity of today's software and the frequent need for quick adjustments e.g. to changes in third-party libraries or consumer needs, this has a drawback: The implementation may evolve faster than the model, resulting in the two slowly becoming out of sync.

Some approaches try to remedy this by deriving the model from source code instead, but can rarely reconstruct the entire model [4]. Others use class structures to embed the model into source code, which makes it available at runtime at the cost of program size and execution time overhead [1].

Our approach includes advantages of both cases. Thanks to the fixed semantics of attributes, the entire model can be unambiguously derived from annotated source code. It can also be made available at runtime through aspects with overhead limited to the model parts the developer actually needs. By default (with no runtime model availability), there is zero overhead.

Also, the tight coupling between source code and model supports their co-development: After changes to the source code (e.g. adding a new low-power listening feature, and hence new state, to a radio driver), all that is left to do is update the annotations of affected functions – which are located right next to the function declarations which need to be updated anyway. This considerably lowers the effort of maintaining a model alongside an implementation.

[4]Without attribute parameters, as they were not yet implemented at that point

6 Discussion

AspectC++ may give the impression that instrumentation is only possible for C++ software. However, plain C still is the dominant language for (embedded) operating systems. To show that our approach is also useful when handling C code, we implemented portable attributes and a joinpoint API in the open-source embedded operating systems CocoOS[5] and RIOT[6], both of which are written in C.

The systems are quite different: CocoOS is a compact cooperative scheduler with support for semaphores, events and message passing. RIOT, on the other hand, has real-time features, UDP/IPv6 support and a wide range of target platforms with corresponding drivers. Both needed to be ported to AspectC++ to support (custom) C++ attributes.

A common statement is that C++ is "basically a superset of C". From our experience, this is only half the truth: It comes close to being one, but the subtle differences can cost a considerable amount of time and nerves. Nevertheless, a C to (Aspect)C++ port should not take an experienced programmer longer than one or two days for these systems. For systems written in C++, switching to AspectC++ is a matter of minutes.

Porting CocoOS is trivial. In fact, by wrapping attributes in preprocessor macros which evaluate to nothing in C and to `[[...]]` attribute syntax in (Aspect)C++, the same source tree can be used both with C and AspectC++ compilers, though without aspect support in the former case.

With roughly 750,000 lines of source code (compared to 1,500 for CocoOS), RIOT is considerably more complex. We did not port the entire system, but limited our efforts to the native POSIX / x86 part and four example applications. This took about a day to complete, with most of the time spent working out the subtle differences between C and C++. We did not attempt to retain C compatibility in this case.

To assess code size and runtime overhead, we compared C++ versions of CocoOS and RIOT to annotated AspectC++ variants. When using the `-Os` compiler flag, both versions resulted in binaries of the same size and with the same functionality. This confirms that annotating code with custom attributes causes zero overhead. For the C to C++ port we observed a 1.7 kB code size increase in CocoOS and a 0.6 kB decrease in RIOT. However, differences between C and C++ are not the scope of this paper.

When using advices, e.g. for accounting network transmissions, logging write access to variables or disabling interrupts, AspectC++ generates a class structure and puts each advice into a separate function. It takes care to create inline functions and allows generated structures to be optimized. In our proposed use cases, this works out well: We ended up with binary sizes mostly identical to manual implementation of the crosscutting concerns and consistently observed the same system behaviour in both variants.

	Plain	Manual	Aspect-based	
interrupt control	5,374	5,448	5,446	(-2)
accounting 1	192,530	192,826	192,842	(+16)
accounting 2	82,257	82,409	82,409	(±0)

Table 1. Binary size of CocoOS (interrupt control) and RIOT (UDP/IPC transmission accounting) without, with manual and with aspect-based implementations. All values in Bytes.

Table 1 shows parts of our results: an implementation of interrupt control in CocoOS (see Section 3) and accounting of UDP and IPC transmissions (including log output to make sure the accounting variable was not left out by the optimizing compiler) in two example applications in RIOT. The table first shows the system size without the feature, then with manual and with an aspect- and annotation-based implementation.

We assume that the variations between manual and aspect-based implementations exhibit corner cases in the C/C++ compiler's optimizer. Anyhow, with a size difference of less than 0.5% in all of our test cases, we consider them to be negligible.

So, the price for using AspectC++ only needs to be paid once when porting the system. Considering the benefits we observed when implementing our examples, this can be a worthwhile undertaking. For instance, annotation-based interrupt control eliminates a source of programming errors by making sure that each function which disables interrupts also restores them when returning.

Joinpoint APIs save time and effort when introducing new developers to a project: After examining the source code once to find and annotate relevant functions, other developers can implement features based on those without needing in-depth knowledge about the system. Given the size of code bases such as RIOT, we found this to be quite helpful. API annotations can also serve as a small machine-readable documentation layer for use e.g. by IDEs.

Finally, annotation-based models don't just support source code and model co-evolution, but also simplify the addition of models to external code. In our energy modelling work, we relied on an open source driver for one of the radio chips we used. Adding an energy model to it only required extending its header file with the appropriate annotations. In contrast, a traditional MDD approach would have required us to refactor the driver for the respective MDD framework.

7 Related Work

AspectC++ is not the first AOP language and certainly not the last either. However, the number of aspect-oriented extensions for languages which are commonly used for operating system development remains small.

[5]http://www.cocoos.net/

[6]https://riot-os.org/

OS development aside, another notable language is AspectJ [8]. AspectJ is an AOP extension of Java and not only precedes AspectC++, but also served as an inspiration during its development. It supports annotations by means of attributes and also allows for attribute arguments, which can then be accessed inside aspects [9]. However, despite systems such as the now discontinued JavaOS [11] or JX [5], Java is not a common language for OS development. It also lacks support for compile-time introspection, which is present in AspectC++.

Similarly, C# and Python also support attributes with arguments (which are called *decorators* in Python) and introspection on them. As Python decorators allow replacing and extending annotated functions with custom code, they behave similar to aspects. C# is also suitable for writing operating systems (e.g. Singularity [6], which uses an extension of C#), but that is seldomly done.

InterAspect provides AOP support for many low level languages – in fact, it can be used with all languages supported by GCC as it operates on the target-independent GCC intermediate language GIMPLE [12]. This allows for powerful aspects which can rely on static code analysis provided by GCC and may also insert low-level instructions into the program. InterAspect uses plain C as an aspect language. Its pointcuts are similarly expressive as those of AspectC++, though to our knowledge it does not support pointcuts on attributes – which is not surprising, as many of GCC's target languages do not support attributes.

LARA also is a language-independent approach usable for low-level languages, but achieves this by means of source-to-source transformation [10]. Since it is not bound to any compiler or language, LARA aspects are defined in a custom domain-specific language. They operate on the abstract syntax tree (AST) of target languages and require a separate weaver for each supported target. Target languages include Java, C and MATLAB. Unlike AspectC++ and InterAspect, it also supports pointcuts on language structures such as loops and conditionals and optimizations such as loop unrolling. However, as C does not support native attributes, annotation support in LARA is currently in a proof of concept stage and implemented using comments. So, unlike AspectC++, it does not have native language or IDE support.

We have not covered design details of a joinpoint API and how to ensure that system code is (and remains) compatible with annotation-based aspects. This problem has already been tackled by others: Crosscut programming interfaces (XPIs) serve as an abstraction between crosscutting aspects and affected code [14]. They were not designed with pointcuts on annotations in mind, but can easily be adapted towards them. Most importantly, XPIs specify how programmers should write code to ensure that joinpoints remain usable regardless of changes in the system code.

Model-based pointcuts aim to retain aspect usability after code changes by specifying pointcuts on a conceptual program model [7]. This is similar to our approach of embedded models inside source code.

8 Conclusion

We have shown that annotations with custom C++ attributes, as implemented in AspectC++ 2.2, are a helpful addition for operating system development. They can not only be used to augment already supported C++ attributes with additional features and make compiler-specific attributes portable, but also for joinpoint interfaces and co-evolution of code and models.

Most notably, attributes allow developers to express intentions and semantics of program code. Parts of this have a side effect on the AOP notion of *obliviousness* (which demands that code should not know about the aspects affecting it): In a joinpoint API, program parts clearly express that they may be affected by aspects. However, they do not contain code to support weaving of aspects and do not know which aspect (if any) will affect them. So, we feel that this does not violate core principles of aspect-oriented design.

Considering penalties incurred by using AspectC++ instead of plain C++, our discussion shows that differences in the compiled binaries are negligible. The effort of porting C software to AspectC++ (and optionally remaining compatible with C compilers) is sufficiently low for this approach to be viable – especially for reasonably-sized embedded operating systems.

Acknowledgments

This work was partly supported by the German Research Council (DFG) within the Collaborative Research Centre SFB 876, project A4.

References

[1] Moritz Balz, Michael Striewe, and Michael Goedicke. 2009. Embedding Behavioral Models into Object-Oriented Source Code.. In *Software Engineering 2009*. Gesellschaft für Informatik, 51–62.

[2] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. 2012. CiAO/IP: A Highly Configurable Aspect-oriented IP Stack. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (Low Wood Bay, Lake District, UK) *(MobiSys '12)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/2307636.2307676

[3] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. 2001. Discussing aspects of AOP. *Commun. ACM* 44, 10 (2001), 33–38.

[4] Michael Goedicke, Michael Striewe, and Moritz Balz. 2009. Support for Evolution of Software Systems using Embedded Models. In *Design for Future–Langlebige Softwaresysteme*.

[5] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. 2002. The JX Operating System. In *USENIX Annual Technical Conference, General Track*. 45–58.

[6] Galen Hunt, James Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. 2005. *An Overview of the Singularity Project.* Technical Report.

[7] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. 2006. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP 2006 – Object-Oriented Programming*. 501–525.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. 2001. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming*. 327–354.

[9] Gary Mak, Josh Long, and Daniel Rubio. 2010. *Spring AOP and AspectJ Support*. Apress, Berkeley, CA, 117–158. https://doi.org/10.1007/978-1-4302-2500-3_3

[10] Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. 2017. LARA As a Language-independent Aspect-oriented Programming Approach. In *Proceedings of the Symposium on Applied Computing* (Marrakech, Morocco) *(SAC '17)*. ACM, New York, NY, USA, 1623–1630. https://doi.org/10.1145/3019612.3019749

[11] S. Ritchie. 1997. Systems programming in Java. *IEEE Micro* 17, 3 (May 1997), 30–35. https://doi.org/10.1109/40.591652

[12] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. 2012. InterAspect: aspect-oriented instrumentation with GCC. *Formal Methods in System Design* 41, 3 (01 Dec 2012), 295–320. https://doi.org/10.1007/s10703-012-0171-3

[13] Olaf Spinczyk and Daniel Lohmann. 2007. The design and implementation of AspectC++. *Knowledge-Based Systems* 20, 7 (2007), 636 – 651. https://doi.org/10.1016/j.knosys.2007.05.004

[14] Kevin Sullivan, William Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. 2010. Modular Aspect-oriented Design with XPIs. *ACM Trans. Softw. Eng. Methodol.* 20, 2, Article 5 (Sept. 2010), 42 pages. https://doi.org/10.1145/1824760.1824762