Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

# An In-Depth Look at ALIA4J

Christoph Bockisch[a] Andreas Sewe[b] Haihan Yin[a]
Mira Mezini[b] Mehmet Akşit[a]

a. Software Engineering group, Universiteit Twente, The Netherlands
b. Software Technology group, Technische Universität Darmstadt, Germany

Abstract New programming languages supporting advanced modularization mechanisms are often implemented as transformations to the imperative intermediate representation of an already established language. But while their core constructs largely overlap in semantics, re-using the corresponding transformations requires re-using their syntax as well; this is limiting. In the *ALIA4J* approach, we identified *dispatching* as fundamental to most modularization mechanisms and provide a meta-model of dispatching as a rich, extensible intermediate language. Based on this meta-model, one can modularly implement the semantics of dispatching-related constructs. From said constructs a single execution model can then be derived which facilitates interpretation, bytecode generation, and even optimized machine-code generation. We show the suitability of our approach by mapping five popular languages to this meta-model and find that most of their constructs are shared across multiple languages. We furthermore present implementations of the three different execution strategies together with a generic visual debugger available to any ALIA4J-based language implementation. Intertwined with this paper is a tutorial-style running example that illustrates our approach.

Keywords Dispatching mechanisms; multiple dispatching; predicate dispatching; pointcut-advice; advanced dispatching; modular language implementation; virtual machines; just-in-time compilation; debugging

## 1 Introduction

To increase the modularity of programs, research in the field of programming languages has introduced different *abstraction* mechanisms, where one concrete program module does not refer to another concrete module, but only abstractly specifies the functionality or data to be used. The mechanisms for resolving such abstract references at runtime are manifold; they include traditional receiver-type polymorphism and reach up to, e.g., multiple and predicate dispatching [CC99], pointcut-advice in aspect-oriented programming [MK03], or layered methods in context-oriented programming [HCN08]. While many of these language mechanisms overlap in their execution semantics, they typically differ syntactically. Compiler frameworks [EH07, ACH$^+$06], alas, only support re-using the implementation of a language's execution semantics if that language is extended syntactically.

## 1.1 Background & Terminology

*Dispatching* is the process of resolving abstractions and binding concrete functionality to their usage. In the following, we thus use the term *dispatch site* for those locations in a program that refer to an abstraction. Consequently, dispatch happens whenever such code locations are executed. We call this execution *join point*, a term borrowed from aspect-oriented programming. A well-known example of dispatching is receiver-type polymorphism: Its dispatch sites are method call sites, with dispatch choosing the applicable method implementation based on the dynamic type of the receiver object. We call languages that go beyond such classic receiver-type polymorphism *advanced-dispatching languages*, as they compose functionality in different, more powerful ways (e.g., before/after advice) and can act on additional runtime state (e.g., argument values/types).

We have identified advanced dispatching as the mechanism suitable to express the majority of language constructs for increased modularity [BHM06]. However, dispatching is a dynamic process; thus, mechanisms that aim at modularly extending the static program structure cannot be satisfactorily realized with it. For example, AspectJ's inter-type declarations can influence the type hierarchy, e.g., by adding interfaces, methods, and fields to a class. While the dynamic semantics of these constructs, namely executing the added methods or accessing the added fields, can be phrased as dispatch, their static semantics cannot; adding an interface changes the type hierarchy and, thus, also impacts the language's type system and the compiler's type checker. Whether this impact may be mitigated by making type checkers aware of advanced dispatching is beyond the scope of this paper.

The implementation of any programming language, advanced-dispatching or not, typically consists of two parts, a *front-end* and a *back-end*, which are decoupled by means of an intermediate language. The front-end processes source code and compiles it into the intermediate language. The back-end either executes this *intermediate representation* (IR) directly or further compiles it into a machine-executable form.

## 1.2 Problem Statement

Typically, implementations of new languages build on the back-ends of already established languages, thereby re-using the implementation of the constructs in that intermediate language. But not all constructs of the new languages have a trivial mapping to the established intermediate language (e.g., Java bytecode), which was tailored to a different source language (e.g., Java). The resulting *semantic gap* between source and intermediate language, i.e., the inability of the intermediate language to directly express the new language's mechanisms, requires compiling that language's high-level concepts down to low-level imperative code.

Compiler frameworks assist in this task, and even enable to re-use the non-trivial code generation for language constructs that have no direct counterpart in the target intermediate language. But this re-use requires the new language to be a syntactic extension of an existing one. Moreover, while code transformations defined on the common intermediate language are shared among all language extensions, they cannot exploit knowledge about new source language constructs; this knowledge is invariably lost during the transformation to the intermediate language. This implies that, while re-use of existing tools like debuggers is possible due the re-use of the intermediate language, the tools' usefulness is greatly reduced. The developer has to observe the program execution in terms of the generated, imperative code in the intermediate language rather than in terms of the new language's source-level abstractions.

## 1.3 Our Contribution

In this article, we present the ALIA4J approach[1] for implementing advanced-dispatching languages. It offers a meta-model consisting of just a small number of well-defined, language-independent abstractions commonly found in advanced-dispatching languages. This meta-model acts as a declarative intermediate language for dispatching-related constructs and removes the existing semantic gap between such source languages and the intermediate language. During language design and implementation, the meta-model has to be extended with the concrete constructs or sub-constructs used in a language. This allows re-using the resulting implementation of a construct's execution semantics without constraining the syntax by which a language exposes this construct. The same is true for any intermediate language: The execution environment of the intermediate language can be re-used without any constraints on the syntax of the source language. The novelty in our approach is the declarativeness of the intermediate language, its dedication to advanced dispatching, and its extensibility. Extensions of the language-independent meta-model can handle intermediate representations in terms of the *language-independent* meta-model and, conversely, implementations of such extensions do not depend on the execution environment.

To execute code defined in the intermediate language, we provide several back-ends, including platform-independent ones. These back-ends instantiate a framework that automatically derives an execution model from the advanced-dispatch's intermediate representation, acting as *Meta-Object Protocol* (MOP) of advanced dispatching. As the MOP retains the IR's declarative nature, the back-end is free to chose from different *execution strategies*, ranging from interpretation to (machine) code generation. Furthermore, it makes implementing tools and analyses for compiled advanced-dispatching programs viable, as we will show by a debugger for advanced-dispatching languages.

The goal of ALIA4J is to ease the burden of programming-language implementation resting upon researchers of new abstraction mechanisms; also several domain-specific programming languages can be mapped to a dispatching problem. It should be noted, however, that our approach is only concerned with the *execution* semantics of the different mechanisms and languages. It imposes no limits on how language researchers and designers can use or combine language (sub-)constructs.

The contributions of this work are as follows:

1. We introduce advanced-dispatching as a declarative yet expressive execution model.

2. We provide a meta-model for advanced dispatching, whose generality and re-usability we show by refining it for AspectJ [KHH$^+$01], CaesarJ [AGMO06], JPred [MFRW09], Compose* [dRHH$^+$08] as well as for several domain-specific languages including ConSpec [AN08].

[1]The Advanced-dispatching Language Implementation Architecture for Java. See `http://www.alia4j.org/`.

3. For executing the advanced-dispatching IR, we provide a framework implementing common work flows. We demonstrate that it supports execution in terms of interpretation and (machine) code generation by providing three back-ends based on different execution strategies: NOIRIn, SiRIn, and Steamloom$^{\text{ALIA}}$.
4. We provide a graphical debugger for programs using our meta-model as IR. This debugger helps to observe the execution of advanced dispatch and to localize all constructs in the source code that influence it.

### 1.4 Structure & Previous Work

The next section introduces a tutorial-style running example, whose subsequent parts are inter-woven with the remainder of this article. Section 3 then fully describes the ALIA4J approach, including the meta-model and framework, and Section 4 takes an in-depth look at ALIA4J's meta-object protocol. Both meta-model and framework are evaluated in Section 5. The generic debugger for advanced-dispatching languages is presented in Section 6. Section 7 discusses related work before Section 8 concludes.

This article is an extension of papers on ALIA4J presented at TOOLS '11 [BSMA11] and VMIL '11 [BSZ11]. Section 6, which describes the ALIA4J-based debugger, both summarizes and generalizes a paper presented at MODULARITY:aosd '12 [YBA12]. The tutorial presented in this article is completely new. Sections 3, 4 and 7 are significantly extended over prior publications.

## 2 Running Example

To illustrate how the ALIA4J approach works in practice, we present, inter-woven with the main matter of the article, a tutorial-style running example. For easy access, the different parts of this tutorial are marked with a vertical bar in the margin.

Tutorial part I – Introduction

In this tutorial, we will design and (partially) implement a small extension to Java that declaratively realizes the Decorator design pattern. In this pattern, a so-called decorator object is associated with a decoratee object, with both objects being instances of a common supertype. Whenever the user invokes a method from the common supertype on the decorator, the decorator can execute its own behavior and also forward the call to the decoratee. The Writer from the java.io library is an excellent example of this:

```
1 Writer decoratee = new FileWriter("...");
2 Writer decorator = new BufferedWriter(decoratee);
3 // Perform I/O
4 decoratee.close(); // Issue: Does not flush the decorator's buffer.
```

However, the above example also highlights a well-known issue with the Decorator pattern: The decoratee remains an autonomous object. It is thus possible to accidentally bypass its decorator (Line 4), which then cannot execute its own behavior, e.g., flushing its buffer. The language designed in this tutorial fixes this issue by redirecting calls, if necessary, to the decorator; accidental mis-use becomes impossible.

The language's semantics is based on four assumptions: (1) There exist both a *decorator class* and a *decoratee class* whose objects may play the role of decorator and decoratee, respectively. A decoratee class can be associated with multiple decorator classes. (2) One *decoratee object* can have at most one direct *decorator object*, which is an instance of one of the decorator classes associated with the decoratee's class. However, a decorator object can itself be decorated, i.e., play the decorator role and the decoratee role at the same time; thus, decorators can be stacked. (3) The relationship between a decorator object and a decoratee object is established by a *decorating constructor*, i.e., a constructor of the decorator class which takes an instance of the decoratee class as argument. (4) Only the decorator object is allowed to directly call operations on the decoratee object.

For implementing the language design according to the above assumptions we need runtime support with three distinct responsibilities: The runtime maintains a mapping from decoratee to decorator classes, it maintains a mapping from decoratee to decorator objects that is updated whenever a new decoratee-decorator relationship is established, and it redirects calls of operations on a decoratee to its decorator—unless the operation is called by the decorator itself. We can thus rephrase the above example:

```
DecoratorRuntime.enforce(FileWriter.class, BufferedWriter.class);
Writer decoratee = new FileWriter("...");
Writer decorator = new BufferedWriter(decoratee);
// Perform I/O
decoratee.close(); // Issue fixed: Call redirected to decorator.
```

The remainder of this tutorial outlines how to first prototypically implement this language extension and then optimize its execution.

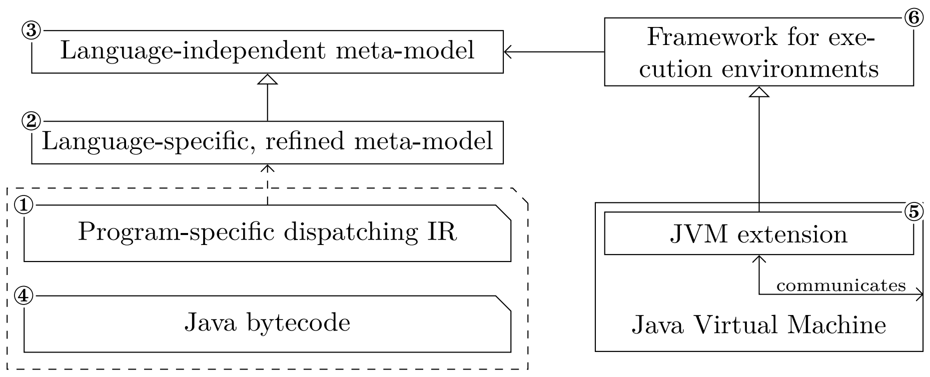
## 3 The ALIA4J Architecture

The ALIA4J architecture realizes our approach to implementing programming languages with advanced dispatching. At its core, it contains a meta-model of advanced dispatching declarations, called *LIAM*,[2] and a framework for execution environments that handle these declarations, called *FIAL*.[3] LIAM hereby defines a *language-independent meta-model* of concepts relevant for dispatching. For example, dispatch may be ruled by predicates which depend on values in the dynamic context of the dispatch. When mapping the concrete advanced-dispatching concepts of an actual programming languages to it, LIAM either has to be *refined* with the *language-specific* semantics or suitable, existing refinements have to be re-used. FIAL defines workflows common to all execution environments able to execute a LIAM-based intermediate representation. In the following two sub-sections, we will discuss both LIAM and FIAL in detail.

But first, we will present ALIA4J's overall architecture, as shown in Figure 1, and illustrate the relationships between its components by outlining the flow of compilation and execution of applications on top of an ALIA4J-based language implementation: First, the compiler processes the application's source code; it produces an intermediate representation ① for the advanced dispatching declarations in the program based on the refined subclasses ② of the LIAM meta-entities ③. Moreover, it also produces Java bytecode ④ for the program parts not using advanced dispatching. Then, at runtime,

[2]The Language-Independent Advanced-dispatching Meta-model. See `http://www.alia4j.org/alia4j-liam/`.

[3]The Framework for Implementing Advanced-dispatching Languages. See `http://www.alia4j.org/alia4j-fial/`.



③ Language-independent meta-model
② Language-specific, refined meta-model
① Program-specific dispatching IR
④ Java bytecode
⑥ Framework for execution environments
⑤ JVM extension
communicates
Java Virtual Machine

Figure 1 – Components and artifacts in an ALIA4J-based language implementations.

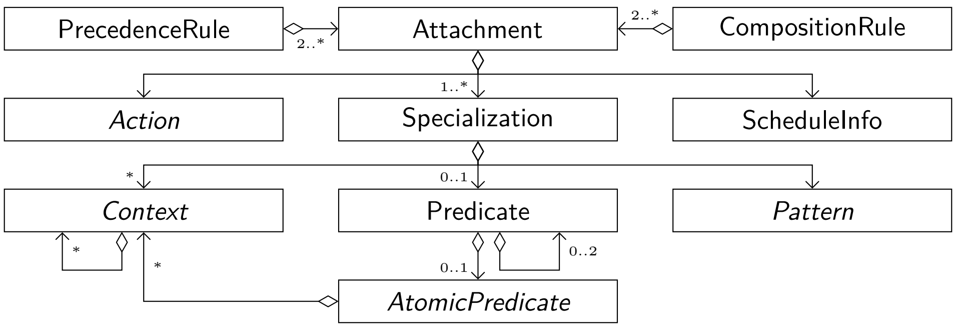


Figure 2 – The LIAM meta-model of advanced dispatching.

both LIAM-based IR and bytecode are passed to a concrete JVM extension ⑤ and thereafter handled by the FIAL framework itself ⑥. Note that this JVM extension can be implemented in a variety of ways, e.g., as a light-weight VM plug-in (NOIRIn, SiRIn) using standard APIs like java.lang.instrument or JVMTI offered by production JVMs, or as a more heavy-weight modification of a particular JVM (Steamloom$^{\text{ALIA}}$). We discuss and evaluate the three JVM extensions in Sections 4 and 5.2, respectively.

## 3.1 A Meta-Model of Advanced-Dispatching

The meta-entities of LIAM capture the core concepts underlying the various dispatching mechanisms, but at a finer granularity than commonly found in the source languages; one concrete concept often maps to a combination of concepts in LIAM. This facilitates mapping diverse mechanisms to just a handful of LIAM entities; re-use of meta-entities between different language mappings increases as well.

LIAM's meta-entities, shown in Figure 2, are implemented as plain Java classes, some of which are abstract. The entities Attachment, Specialization, and Predicate simply group meta-model entities and cannot be refined. In the following, we will explain the meta-entities in a bottom-up fashion, all the way to Attachment, and illustrate them with example refinements (not shown in Figure 2).

**Pattern** Each dispatch site, e.g., field access or method call site, has a signature, These signatures can be quantified over by means of LIAM's Patterns. As one can model the majority of advanced-dispatching languages using just five kinds of dispatch sites (calling a method, constructor, or static initializer, and reading or writing a field), LIAM offers five predefined subclasses of Pattern. While many pointcut-advice languages [Nag06, Chapter 2] support such pattern-based quantification, other advanced-dispatching languages typically use trivial patterns without wildcards; i.e., the used patterns correspond to exactly one signature.

**Context** Dynamic dispatch is context-dependent. In LIAM, the Context meta-entity models this dependence on runtime values: Its refinements embody different kinds of values that are available during dispatch, e.g., a single argument (ArgumentContext) or the receiver object (CalleeContext). Context entities can also model values that are derived from other values, e.g., the reification of an AspectJ join point (ThisJoinPointContext) accessible through the **thisJoinPoint** keyword, which aggregates various values like the caller, callee, and arguments, all of which are modeled by contexts of their own.

**Atomic Predicate** During dispatch, the current context can not only be exposed, but also be subjected to tests. In LIAM, the Atomic-Predicate meta-entity models a single such test parameterized with Contexts specifying the values to test. For example, a dynamic type check (InstanceofPredicate) parameterized with an argument of a call (ArgumentContext) models multiple dispatching.

**Predicate** Since dispatch predicates in the investigated languages can be arbitrarily complex, LIAM also provides composite Predicates, which are trees whose inner nodes are either conjunctions (AndPredicate) or disjunctions (OrPredicate) and whose leafs (LeafPredicate) are either labeled with an Atomic Predicate or its negation. Every predicate is thus a Boolean formula in negation normal form.

**Action** Once all predicates have been evaluated, actions need to be performed. LIAM models the dispatch's targets as collections of Action meta-entities, which are e.g., calling an individual method (MethodCallAction) or raising an exception (ThrowAction).

**Specialization** The Specialization meta-entity selects specific calls by associating a Pattern with a Predicate. It also declares, as a list of Contexts, which runtime values must be exposed to Actions at selected join points. An AspectJ pointcut corresponds to a collection of Specializations which together match all join points selected by the pointcut. Predicate dispatching is modeled using one Specialization per predicate. A JPred **when** clause [MFRW09] then corresponds to the Specialization's predicate.

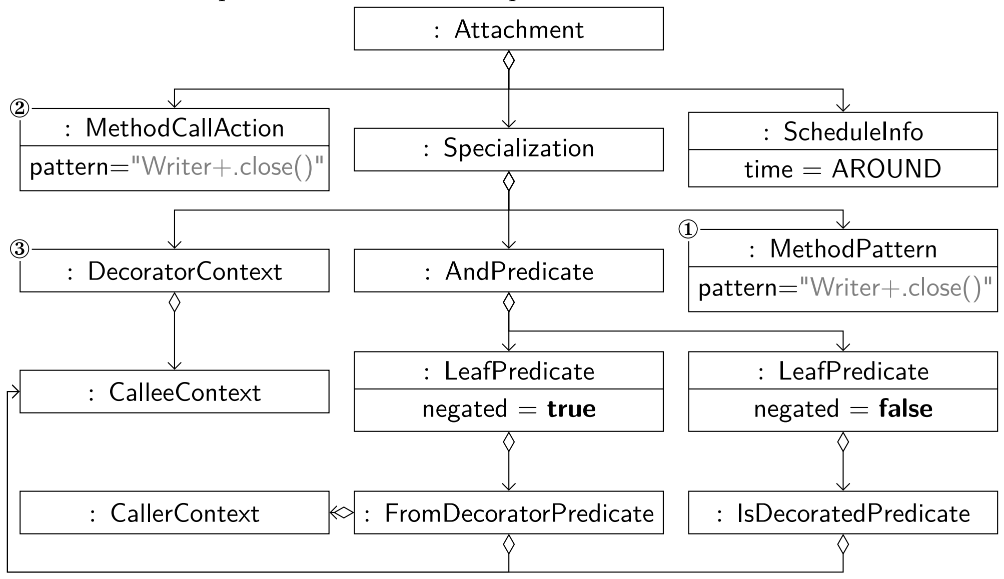
**Attachment and Schedule Information** A collection of Specializations is associated with an Action by the Attachment meta-entity. Whenever both the Pattern and the Predicate of a Specialization match, the context values specified by the respective Specialization are passed to the Action as arguments. An Attachment's associated Schedule Information meta-entity describes when the Action is to be executed. This corresponds to AspectJ's *before*, *after* and *around* keywords.

**Precedence Rule and Composition Rule** LIAM allows to control the execution order of Actions that are together applicable at the join point by means of the Precedence Rule meta-entity. By such rules, a partial order among Attachments is defined which carries over to the associated Actions at shared join points. The Composition Rule meta-entity can be used to define overriding relationships and constraints for jointly executed Actions in a similar way.

Tutorial part II – The Meta-Model

Armed with knowledge about LIAM, we can now model the redirection of calls from decoratee to decorator (cf. part I of the tutorial) with one Attachment for each operation we want to redirect. Below, we show the common structure of all redirection Attachments by the example of the close method. Individual Attachments only differ in the intercepted method (printed in gray) and the reified method arguments; as close takes no arguments, there are none in the example.

These Attachments are created and deployed when a new decoratee-decorator relationship is established, i.e., when the decoratee object is passed to the decorator's constructor. This process will be shown in part III of the tutorial.



Each Attachment's sole MethodPattern ①, shown in AspectJ-like syntax, selects all calls to the method to be redirected, e.g., to all zero-argument methods named close belonging to the type Writer or any subtype thereof (denoted by +). The action to be performed at selected dispatch sites, specified by the MethodCallAction ②, is to call the method with the same name and signature on the associated decorator object, specified by the DecoratorContext ③. Any arguments, of which close happens to have none, would be specified just like the receiver with a Context of their own.

Beyond general-purpose LIAM entities there are several that are specific to our language extension: DecoratorContext, IsDecoratedPredicate, and FromDecoratorPredicate. These allow us to model that the redirection should target the associated decorator object (DecoratorContext) if the original receiver is indeed decorated (IsDecoratedPredicate) and we are not dealing with a call made by the decorator itself (FromDecoratorPredicate); such calls should obviously not be redirected.

## 3.2 A Framework for Implementing Advanced-Dispatching Languages

FIAL defines several common components needed by any execution environment that uses a LIAM-based intermediate representation. Figure 3 illustrates FIAL's abstract components and their concrete implementations in a JVM extension. Both

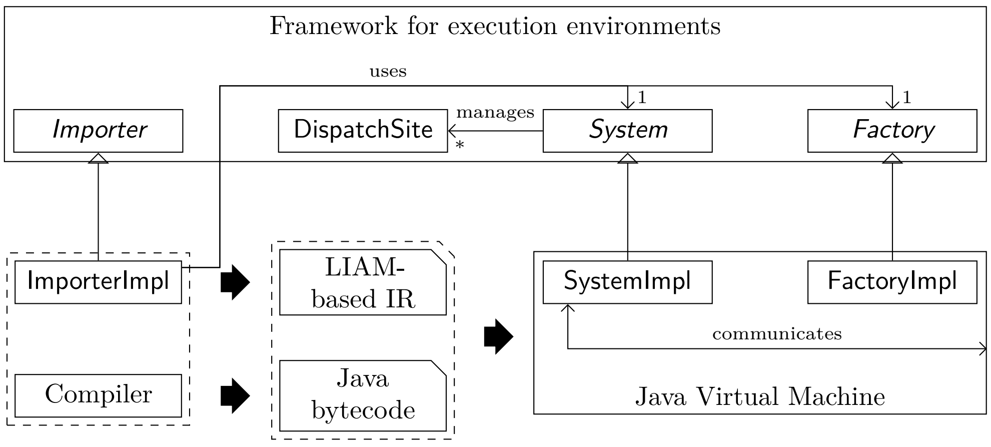


Figure 3 – Abstract and concrete components in FIAL and JVM extensions, as well as the flow (thick arrows) of code representations (chamfered boxes).

communicate through a shared first-class representation of the dispatch sites in the program: Extended JVMs have to create such a representation for every dispatch site encountered in the program under execution. FIAL constructs the dispatch site's execution model and stores it in the shared dispatch-site representation. The execution model is based on LIAM entities and acts as meta-object protocol (cf. Section 4). The following paragraphs explain FIAL's central components and common workflows.

**System** The system component is a Singleton used throughout FIAL and JVM extensions for managing deployment and undeployment of Attachments. When classes are loaded at runtime through Java's dynamic class loading, the JVM extension has to provide FIAL with DispatchSite objects identifying the dispatch sites contained therein. At class loading, the System scans all previously deployed Attachments and matches their Patterns against the newly available dispatch sites. When a Pattern matches a dispatch site FIAL attaches the Action, Predicate and Contexts associated with the pattern to that site. At deployment of an Attachment, the System likewise identifies which of the already-loaded dispatch sites are affected and modifies them accordingly; at undeployment, the corresponding Action, Predicate and Contexts are removed from the dispatch site. After either deployment or undeployment, the JVM extension is notified which dispatch sites have been affected. Execution environments that do code generation for dispatch sites can thus (re-)generate the affected code. The dynamic deployment of ALIA4J supports language mechanisms like the dynamic aspect deployment of CaesarJ [AGMO06] or JAsCo [SVJ03], or behavioral reflection as found in Smalltalk or Reflex [TT06].

## Tutorial part III – Deployment

The Attachments for forwarding method calls from the decoratee object to the decorator object, created as explained in part II of the tutorial, do not take effect until they are deployed. According to the semantics of our Decorator-enforcing DSL (cf. part I of the tutorial) forwarding must be performed after a decorating constructor of the decorator class has been called. With ALIA4J, we can implement these semantics by defining an Attachment whose Action is executed after a decorating constructor has been called.

The two values exposed to the Action must be the newly created object (the decorator object) and the decorating constructor's argument (the decoratee object).

For brevity, we leave out the definition of this Attachment and only show the code of the Action that ultimately gets performed. Its perform method (lines 2–9) takes the decorator object and the decoratee object as arguments. It passes the pair of these objects to the DecoratorRuntime to establish their relationship which can be queried as will be shown in part V of the tutorial. Afterward, it creates new Attachments that describe the forwarding and passes them to the deploy method (line 8) of FIAL's System class; from then on, the Attachments are active and all calls bypassing the decorator are automatically forwarded to the decorator.

```
1 class DecorateAction extends Action {
2   public void perform(Object decorator, Object decoratee) {
3     DecoratorRuntime.establish(decoratee, decorator);
4     Attachment[] attachments =
5         new Attachment[/* number of decoratee methods */];
6     for (int i = 0; i < /* number of decoratee methods */; i++)
7       Attachment[i] = // create Attachment as shown in previous tutorial part
8     org.alia4j.fial.System.deploy(attachments);
9 } }
```

**Factory** The factory component used to create LIAM entities is also a Singleton that interfaces with JVM extensions as well as with clients. As it follows the Abstract Factory design pattern, the implementer of the JVM extension can replace the concrete factory to be used without affecting any client. This way, one can ensure the consistent usage of LIAM entity implementations which exploit low-level APIs of the extended JVM to execute more efficiently than the default implementation (cf. Section 4). This pattern must also be followed whenever a LIAM-based language implementation introduces additional concrete entities.

**Importers** FIAL provides a call-back that is triggered right before the JVM starts the application. Language developers can thus implement and register a call-back in order to define and deploy Attachments according to the semantics of the implemented language. In particular, the call-back can be used to integrate legacy language front-ends that generate a proprietary intermediate representation of advanced-dispatching concepts, which can then be transformed to the LIAM-based intermediate representation by the importer. We have implemented two such importers: one for the AspectJ language that processes bytecode and the annotations added by the AspectJ compiler, and one for the ConSpec language [AN08] that directly processes ConSpec source code. The language developer is thus flexible with respect to the importer's inputs.

Tutorial part IV – Importer

Explicitly instructing the runtime to enforce the Decorator pattern (cf. part I of the tutorial) is cumbersome; a better solution is a language extension. As our case study is simple, so is the resulting language: a single annotation (@Decorates).

```
@Decorates(Writer.class)
class BufferedWriter extends Writer {
  BufferedWriter(Writer writer) { ... }
  void close() { ... }
  ...
}
```

The Java Development Kit provides the Annotation Processing Tool; an Annotation Processor can be implemented for this tool to ensure the correct usage of the annotation, e.g., whether an appropriate constructor is defined. Furthermore, the Annotation Processor can write a summary file listing all relationships between decoratee and decorator classes. Upon import, this summary is then turned into a runtime representation:

```
 1 class DecoratorImporter implements Importer {
 2   void performImport() {
 3     Map<String, String> decorator2Decoratee = process(
 4         getClass().getClassLoader().getResourceAsStream("relations.summary"));
 5     // Populate decorator2Decoratee mapping
 6     for (String decorator : decorator2Decoratee.keySet())
 7       DecoratorRuntime.enforce(
 8           systemClassLoader.loadClass(decorator),
 9           systemClassLoader.loadClass(decorator2Decoratee.get(decorator)));
10 } }
```

When triggered to execute, our importer picks up the summary file from the application's class path (Line 4) and extracts the persisted relationships. It then simply instructs the runtime to create and deploy the necessary Attachments (Line 7).

## 4 Implementations of ALIA4J's Meta-Object Protocol

From all deployed Attachments, FIAL creates an execution model for each dispatch site in the program. The execution model of a dispatch site combines the Predicates, Actions and Contexts associated with a Pattern that matches the signature of the dispatch site. It has the form of a dispatch function [CC99] determined by the evaluation of the Atomic Predicates jointly occurring in the dispatch site's Predicates. The result of evaluating the dispatch function is an ordered tree of Actions to execute, which is derived by FIAL from three LIAM entities: Schedule Information, Precedence Rule, and Composition Rule [BMAK11]. The Actions at each level have to be executed sequentially; Actions associated with an *around* Schedule Information may have children which are executed when the around Action *proceeds*. This is to accommodate the around Advice found in AspectJ and other aspect-oriented languages.

To represent dispatch functions, FIAL uses binary decision diagrams (BDDs) [Bry86]. They provide an explicit evaluation strategy and can be directly transformed into a branching program, which enables FIAL to generically perform optimizations on the execution model [SBM08]: Each Atomic Predicate will be tested at most once even if used by multiple Predicates; dispatch is redundancy-free. FIAL also automatically optimizes the evaluation strategy with respect to its average runtime cost.

Testing all predicates before performing any action is only possible under two assumptions: The tests are side-effect-free and their outcome is not influenced by the actions to be performed. These assumptions are trivially true for multiple dispatching, but often also hold for predicate dispatching [EKC98] and advice dispatching [SBM08].

Moreover, we have preliminary results in improving our execution model to take action-induced changes of context values into account by splitting the dispatch function into several BDDs whose evaluation is then intertwined with action execution. Further details about this are subject to future work.

The declarative execution model fully describes the semantics of dispatch sites in terms of first-class objects, the LIAM entities, which can be introspected and modified (by deploying and undeploying Attachments) by the application program. It thus forms a meta-object protocol (MOP) defining the control flow and data flow of evaluating the LIAM entities which comprise the execution model. This protocol may be implemented differently by different JVM extensions; we have explored implementation strategies based on interpretation, bytecode generation and just-in-time (JIT) compilation. Thus, ALIA4J's MOP is a runtime MOP *and* a (JIT) compile-time MOP.

The language developer is free to use any level of MOP by choosing among the three strategies when implementing the language's semantics through refined LIAM classes. The JVM extension will then pick the lowest-level implementation which is provided by the LIAM entity in question and supported by the JVM extension itself,[4] working under the assumption that the lowest-level implementation performs the most effective optimizations. It is not necessary, however, to provide a complex, low-level implementation if it does not improve performance over plain interpretation or if performance is not an issue. LIAM entities implemented using different strategies can freely be mixed at runtime.

## 4.1 The Runtime MOP in NOIRIn

NOIRIn[5] is a fully portable JVM extension implemented using the java.lang.instrument API. Using the ASM bytecode engineering library,[6] it replaces every dispatch site in the program with an invocation of a call-back method. When this call-back is invoked, NOIRIn retrieves the execution model associated with the dispatch site in question and interprets it by traversing the execution model's object structure and evaluating every LIAM entity it encounters. For ALIA4J's runtime MOP it is required that each LIAM entity implements a method taking all context values it requires as arguments and returning the evaluation result. While the method name is subject to a naming convention that depends on the entity type, for simplicity we refer to is as the "compute" method throughout this article.

To evaluate a LIAM entity, first, all Context entities it depends on must be evaluated. Next, the "compute" method is invoked passing the result values from the previous step as arguments. During the evaluation of the dispatch function, the traversal of the BDD depends on the boolean value returned by the interpretation of the Atomic Predicates' own "compute" methods.

---

Tutorial part V – Runtime MOP

---

Of the LIAM entities specific to our small language extension, we show how a language designer can implement the FromDecoratorPredicate in plain Java (the

[4]Some JVM extensions may not support the bytecode-generation or JIT-compilation strategies.
[5]The Non-Optimizing Interpretation-based Reference Implementation. See `http://www.alia4j.org/alia4j-noirin/`.
[6]See `http://asm.ow2.org/`.

others are even simpler) in the listing below. The predicate's implementation declares (Lines 3–4) that its evaluation depends both on the caller (CallerContext) and the callee (CalleeContext). Using the runtime's decoratee-to-decorator mapping the predicate can trivially compute whether the caller is the callee's decorator (Line 7).

```
1 class FromDecoratorPredicate extends AtomicPredicate {
2   FromDecoratorPredicate() {
3     super(ContextFactory.findOrCreateCallerContext(),
4         ContextFactory.findOrCreateCalleeContext());
5   }
6   public boolean isSatisfied(Object caller, Object callee) {
7     return caller == DecoratorRuntime.mapDecoratee2Decorator(callee);
8 } }
```

## 4.2 The Compile-Time MOP in SiRIn

Like NOIRIn, SiRIn[7] is a fully portable JVM extension implemented using the java.lang.instrument API. Through ASM-based bytecode transformation, it wraps dispatch sites into special methods; each dispatch site in the original program is transformed into a statically-bound call to a site method containing the actual dispatch logic. This clean separation between base program and dispatch logic has two advantages: It makes the runtime values available during dispatch explicit as arguments to the site method and it allows for straight-forward code generation.

Note that this separation between base program and dispatch logic does not incur a performance penalty, because modern JVMs inline small, statically-bound methods like the wrapped dispatch sites. Within these site methods, however, using the code generation strategy implied by the runtime MOP does incur a performance penalty for many fundamental entities (e.g., CalleeContext, ArgumentContext) which perform little if any computation but, e.g., simply load an argument value. Especially for those entities bytecode generation strategies are obviously more efficient than a reflectively implemented "compute" method. In fact, often bytecode can be generated which is equivalent to code a Java compiler would generate for the selfsame functionality.

SiRIn therefore offers a compile-time MOP. In this MOP, SiRIn passes a description of the syntactic context in which the dispatch takes place to a dedicated bytecode-building method. This method can then use the information together with a simple, assembler-like interface to generate Java bytecode. In particular, the method has full control over when and whether depended-upon context values are computed, as such computations are often only necessary in some rather than all syntactic contexts.

Tutorial part VI – Compile-Time MOP

For the language-specific LIAM entities, explicit bytecode generation offers few optimization opportunities. Thus, in this tutorial, we refrain from demonstrating an implementation using the compile-time MOP; SiRIn will simply rely on the plain Java implementations presented in part V of the tutorial.

[7]The Site-based Reference Implementation. See http://www.alia4j.org/alia4j-sirin/.

## 4.3 The JIT Compile-Time MOP in Steamloom$^{\mathrm{ALIA}}$

Steamloom$^{\mathrm{ALIA}}$ is an extension of the Jikes Research VM (RVM) [AAB+05], a high-performance Java VM. Like SiRIn, Steamloom$^{\mathrm{ALIA}}$ wraps dispatch sites into special method-like constructs. Also Steamloom$^{\mathrm{ALIA}}$ allows LIAM entities to make use of the compile-time MOP, as described in the previous subsection. In addition, Steamloom$^{\mathrm{ALIA}}$ can bypass bytecode generation for LIAM entities and directly generate native machine code for them, using one of the two JIT compilers of the Jikes RVM, the baseline compiler and the optimizing compiler.

Both compilers are exposed through the JIT compile-time MOP offered by Steamloom$^{\mathrm{ALIA}}$. In it, a machine-code building method gets passed a description of the current JIT compilation context. This includes not only all the syntactic information about the currently compiled dispatch but also, in the case of the optimizing compiler, additional information about the chain of calls leading to the dispatch in question.

Tutorial part VII – Just-in-Time Compile Time MOP

According to our requirements analysis in part I of the tutorial, each object has at most one directly associated decorator object, a mapping which is so far maintained by the DecoratorRuntime. One possible optimization is to allocate a dedicated field in instances of a decoratee class to store the reference to the decorator object, if any. LIAM entities that query the mapping, e.g., FromDecoratorPredicate, simply look up the decorator in that field.

This optimization is not possible in a portable fashion, as it affects the object layout, If one is willing to directly interact with a specific execution environment, however, one can perform such an optimization—as well as others beyond what is possible within the limits of Java bytecode. We thus implement the JITSupport interface specific to Steamloom$^{\mathrm{ALIA}}$ and the Jikes RVM's two JIT compilers.

For Jikes RVM's so-called baseline compiler we make use of the fact that the flags set upon comparing the addresses of the caller and the callee's decorator can be loaded into a register. Properly masked, the Zero Flag, which indicates equality, can be directly treated as the predicate's result; no branching is required.

```
class FromDecoratorPredicate extends AtomicPredicate implements JITSupport {
  public void generateBaselineIR(Assembler asm) {
    // Load context value
    asm.emitPOP_Reg(T0);
    // Load field value into T0 and compare with context value at 0(SP)
    asm.emitMOV_Reg_RegDisp(T0, T0, decoratorField.getOffset());
    asm.emitCMP_Reg_RegDisp(T0, SP, Offset.zero());
    // Extract Zero Flag and store it at 0(SP)
    asm.emitLAHF();
    asm.emitAND_Reg_Imm(EAX, ZERO_FLAG_BITMASK);
    asm.emitMOV_RegDisp_Reg(SP, Offset.zero(), EAX);
} }
```

For Jikes RVM's optimizing compiler we instead make use of the INT_COND_MOVE instruction part of the JIT's intermediate-instruction set; the result is set to 1 (**true**) if decorator and caller are identical and to 0 (**false**) otherwise. Again, no expensive branching is required.

```
class FromDecoratorPredicate extends AtomicPredicate implements JITSupport {
  public void generateHIR(BC2IR asm, GenerationContext gc) {
    Operand callee = asm.popRef();
    Operand caller = asm.popRef();
    RegisterOperand decorator = createObjectTemporary();
    asm.appendInstruction(GetField.create(GETFIELD,
        decorator, callee, new AddressConstantOperand(decoratorField.getOffset()),
        new LocationOperand(decoratorFieldRef), . . . ));
    RegisterOperand result = createBooleanTemporary();
    asm.appendInstruction(CondMove.create(INT_COND_MOVE,
        result, decorator.copyD2U(), caller, EQUAL,
        new IntConstantOperand(1), new IntConstantOperand(0)));
    asm.push(result.copyD2U());
} }
```

## 5 Evaluation

We evaluate the ALIA4J approach on two levels: First, we investigate LIAM's ability to realize new as well as existing languages and the degree of re-use facilitated by our approach. Second, we show the independence of both FIAL and our execution model of a concrete environment's execution strategy.

### 5.1 Evaluation of LIAM

LIAM strives to be extensible where necessary and to allow for re-use where possible. With these goals in mind, we have evaluated LIAM in two distinct scenarios: First, we have refined LIAM ourselves with the concrete language sub-constructs found in several existing languages with very different philosophies of modularization (AspectJ, CaesarJ, Compose*, JPred, and ConSpec) [BSMA11]. Second, we have asked students to implement a domain-specific language (DSL) of their own choosing.

#### 5.1.1 Existing (General-Purpose) Languages

To evaluate LIAM's applicability "in the large," we implemented refined meta-models for various advanced-dispatching languages. We have also implemented a further, prototypical refinement supporting the ConSpec language [AN08], which is not an advanced-dispatching language as such but rather a language for policy enforcement.

Table 1 shows the different concrete entities we implemented while mapping the different languages (AspectJ, CaesarJ, JPred, Compose*, and ConSpec) to LIAM, as well as their usage in the different language mappings. Although the entities for the CaesarJ language are a superset of those for AspectJ, Table 1 considers AspectJ and CaesarJ separate languages; the way the entities are used in CaesarJ is sometimes very different from their use in AspectJ. In particular, CaesarJ supports scoped and dynamic deployment, which requires the dynamic creation of mutated Attachments and a mapping between CaesarJ's and FIAL's deployment APIs.

Figure 4 summarizes re-use of entities by means of a Venn diagram showing the overlap in the usage of LIAM entities by four different languages; AspectJ is not shown here because it uses the same entities as CaesarJ. In the Venn diagram, we only consider entities that are non-trivial and explicitly used by the languages' importer, i.e.,

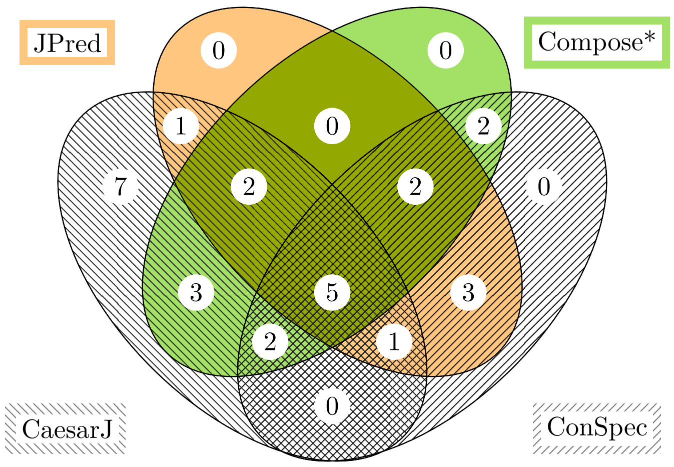


Figure 4 – Venn diagram showing the usage of context entities by four language mappings.

those marked ✓ in Table 1. The diagram shows that 21 of the 28 implemented entities are used by more than one language mapping; of those, seven entities are shared among three languages. The five entities that are shared among all considered languages characterize traditional receiver-type polymorphism and multiple dispatching.

We would like to emphasize, though, that these languages may still differ significantly, even though their execution is based on the same building blocks. In particular, each language defines its own restrictions and provides its own guarantees. JPred, e.g., can guarantee that at each call exactly one method implementation is applicable, an assumption that does not hold in Compose* at all. But this highlights the flexibility of our approach, which does not make any assumptions on the characteristics of dispatch and is thus able to realize languages that fundamentally differ in some respects.

Note furthermore that some of the discussed languages extend Java with mechanisms other than advanced dispatching; AspectJ, e.g., adds so-called inter-type declarations. While we have already mapped inter-type *member* declarations to LIAM, i.e., to advanced dispatching, other mechanisms are out of scope for an architecture focusing on advanced dispatching.

### 5.1.2 Newly-designed DSLs

We asked 22 students,[8] in groups of two or three, to develop prototypes of domain-specific languages embedded into Java. Save for the choice of Java as a foundation, the students' designs were unconstrained and consequently covered a broad range of domains: declarative definition of debugging activities, annotation-defined method-level transactions, asynchronous Future-based inter-thread communication, runtime model checking, as well as authentication and authorization.

All language prototypes could be implemented by re-using the LIAM entity implementations developed in the aforementioned case studies (cf. Table 1), which are shipped with ALIA4J. This serves as evidence that those predefined entities already cover many use cases and makes the ALIA4J approach an attractive proposition for

[8]All students participated in the course "Advanced Programming Concepts" at the University of Twente, taught in both 2009/10 and 2010/11.

| | AspectJ | CaesarJ | JPred | Compose* | ConSpec |
|---|---|---|---|---|---|
| | [KHH$^{+}$01] | [AGMO06] | [MFRW09] | [dRHH$^{+}$08] | [AN08] |
| Pattern | | | | | |
| Method | ✓ | ✓ | ✓ | ✓ | ✓ |
| Constructor | ✓ | ✓ | ✓ | | ✓ |
| StaticInitializer | ✓ | ✓ | | | |
| FieldRead/-Write | ✓ | ✓ | | | |
| Context | | | | | |
| Argument | ✓ | ✓ | ✓ | ✓ | ✓ |
| Callee | ✓ | ✓ | ✓ | ✓ | ✓ |
| Caller | ✓ | ✓ | | ✓ | |
| Result | ✓ | ✓ | | ✓ | ✓ |
| Arguments | ✓ | ✓ | | ✓ | |
| DebugInfo | ✓ | ✓ | | | |
| Signature | ✓ | ✓ | | ✓ | |
| PerTuple | ✓ | ✓ | | ✓ | ✓ |
| PerCFlow-/Below | ✓ | ✓ | | | |
| ObjectConstant | ✓ | ✓ | ✓ | | |
| AspectJSignature | ✓* | ✓* | | | |
| JoinPointKind | ✓* | ✓* | | | |
| SourceLocation | ✓* | ✓* | | | |
| ThisJoinPoint | ✓* | ✓* | | | |
| Thread | | ✓ | | | |
| Constant | | | ✓ | | ✓ |
| Field/ArrayElem. | | | ✓ | | ✓ |
| Binary-/UnaryOp. | | | ✓ | | ✓ |
| MethodResult | | | ✓ | ✓ | ✓ |
| ReifiedMessage | | | | ✓* | |
| AtomicPredicate | | | | | |
| Instanceof | ✓ | ✓ | ✓ | ✓ | |
| Method | ✓ | ✓ | ✓ | ✓ | ✓ |
| ExactType | ✓ | ✓ | ✓ | ✓ | |
| CFlow/CFlowBelow | ✓ | ✓ | | | |
| BinaryRelation | | | ✓ | ✓ | ✓ |
| Action | | | | | |
| MethodCall | ✓ | ✓ | ✓ | ✓ | ✓ |
| FieldRead/-Write | (✓) | (✓) | (✓) | (✓) | (✓) |
| CFlowEnter/-Exit | ✓ | ✓ | | | |
| NoOp | | | | ✓ | ✓ |
| Throw | | | | ✓ | ✓ |

Table 1 – Concrete LIAM entities and their usage in different languages. A ✓ marks non-trivial entities directly used by the language's importer, while a ✓* marks trivial entities which only adapt the computed value to a different interface. A (✓) marks non-trivial entities not used directly by the language's importer but by FIAL itself.

designers of DSLs, who can simply pick the entities they need from LIAM's library. Moreover, the range of domains to which the students successfully applied the ALIA4J approach indicates that our meta-model is expressive enough in practice.

## 5.2 Evaluation of FIAL

We have developed three FIAL-based back-ends (NOIRIn, SiRIn, and Steamloom$^{\text{ALIA}}$) using different execution strategies ranging from interpretation over bytecode generation to machine-code generation.

NOIRIn relies solely on interpretation of the execution model produced by FIAL. Because NOIRIn does not generate code for dispatch sites, it can only handle LIAM entities which implement a "compute" method. This is not a restriction since it can be expected that for each LIAM refinement a "compute" method is initially implemented by the language designer. A language implementer, potentially a different person, may eventually supplant it by optimizing bytecode or machine-code generation.

SiRIn wraps every dispatch site into a special method and generates bytecode for it. Each wrapper method contains code derived from FIAL's execution models. SiRIn may duplicate code if several leaf nodes share an Action. This code-splitting approach opens up new optimization opportunities for the JVM's just-in-time compiler. SiRIn does not require a native component and is, like NOIRIn, fully portable.

The Steamloom$^{\text{ALIA}}$ Virtual Machine, a re-design of an earlier execution environment developed, among others, by the first author [BHMO04], is a JVM extension of the high-performance Jikes Research Virtual Machine (RVM). The JIT compile-time MOP of Steamloom$^{\text{ALIA}}$ offers the possibility to tailor the machine-code generation of both just-in-time compilers of the RVM, the baseline compiler and the optimizing compiler. To give an impression of the performance gain possible by using ALIA4J's JIT compile-time MOP, we present preliminary measurements for the FromDecoratorPredictate from the Decorator-enforcing DSL (cf. parts V and VII of the tutorial) in Table 2. The table shows the execution times we measured for that Atomic Predicate in four different configurations. The two columns contain the measured times when the dispatch site was compiled with the baseline compiler respectively with the optimizing compiler; the two rows contain the measurements for an implementation of the Atomic Predicate using the default runtime MOP respectively the JIT compile-time MOP. This experiment shows that even for relatively simple entities notable speed-ups may be observed: The implementation using the JIT compile-time MOP is more than an order of magnitude faster for the baseline compiler and still more than six times as fast for the optimizing compiler. For other LIAM entities that we have developed for the Decorator-enforcing DSL, native machine-code generation does not significantly improve performance. Nevertheless, for more complex language constructs the speed-up achieved by integration with the just-in-time compiler can be a lot higher [BKH$^{+}$06]; this particular optimization, however, has yet to be ported to Steamloom$^{\text{ALIA}}$.

All FIAL-based execution environments can be tested using the same, extensive test suite. Each of the tests uses the framework to define and deploy LIAM-based dispatch declarations, execute an affected dispatch site, and verify the correct execution. Almost all of the 4,083 tests are systematically generated to cover all relevant variations of dispatch sites and LIAM entities; the test suite thus ensures compatibility between different execution environments like NOIRIn, SiRIn, and Steamloom$^{\text{ALIA}}$.

| | Compiler | |
|---|---|---|
| Code Generation Strategy | Baseline [ns] | Optimizing [ns] |
| Runtime MOP | 47.54 | 23.10 |
| JIT Compile-Time MOP | 4.32 | 3.49 |

Table 2 – Execution times of computing a sample Atomic Predicate's value, using different compilers and code generation strategies.

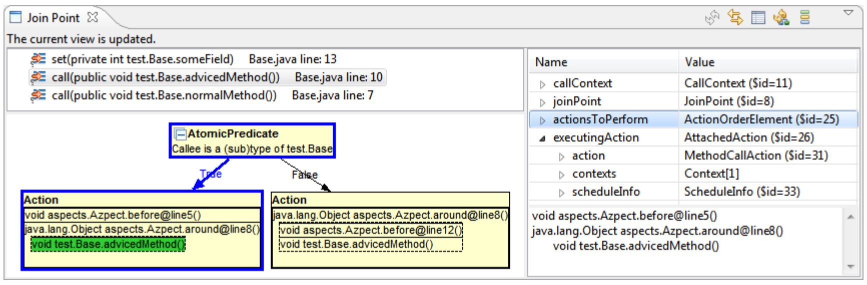
## 6 A Generic Debugger for Advanced Dispatching

As dispatching mechanisms in programming languages become more expressive, the resulting programs become more flexible. At the same time, however, it gets harder for the programmer to predict the target of a dispatch. Therefore, dynamic tools are required to observe and comprehend the program execution. Especially when the program is erroneous and faults must be located in the source code, programmers use so-called debugger tools to observe and intercept the execution and to inspect runtime values while the execution is paused. Because most new programming languages with advanced-dispatching mechanisms provide a compiler that produces intermediate code of an established programming language, the debugger of that underlying language, e.g., Java, can also be used to debug programs with advanced dispatching. As discussed already, in this code transformation approach the structure of the generated code is different than the source code and what is inspected is the synthetic, transformed code instead. Furthermore, not all source code constructs have a counterpart in the intermediate code [YBA$^+$11], e.g., the **declare precedence** statement in AspectJ is evaluated during compilation and the influence of such constructs on the execution cannot be observed with the Java debugger.

An ALIA4J intermediate representation, in terms of LIAM Attachments, provides a representation of every construct influencing dispatch and is available as first-class objects at runtime. The execution model of each dispatch site is also available as first-class objects and is linked with the intermediate representation. Therefore we provide a debugger which allows observing the execution of dispatch in terms of our execution model and LIAM Attachments. While the granularity of the LIAM meta-model and its terminology often differ from those of the actual source language, the structure is similar and each LIAM entity can be traced back to the source code.

Our debugger defines a communication interface that decouples the front-end and the back-end. The back-end is currently only implemented by the NOIRIn execution environment. The front-end of the ALIA4J debugger is integrated into the Eclipse IDE, where it extends the built-in Java debugger with additional views specific to visualizing and interacting with ALIA4J's execution models: the Join Point view, the Attachments view, and the Pattern Evaluation view.

**Join Point View** The Join Point view, shown in Figure 5, is the central view of the ALIA4J debugger. It shows runtime information about the join point at which the debuggee is currently suspended, including the entire stack of join points (top left), the context values needed to evaluate the dispatch function and exposed to the actions (right), and a graphical representation of the execution model for the selected join point (bottom left). To reason about the composition of Attachments at a join point, the ALIA4J debugger can also visualize the precedence relationships between them (not shown). In the case of Figure 5, the execution model's root node represents



The current view is updated.

set(private int test.Base.someField) Base.java line: 13
call(public void test.Base.advicedMethod()) Base.java line: 10
call(public void test.Base.normalMethod()) Base.java line: 7

Callee is a (sub)type of test.Base

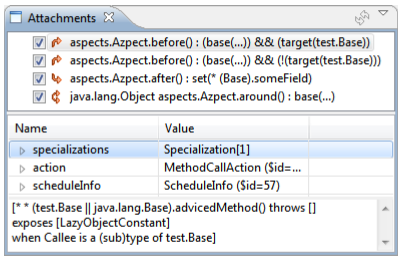
void aspects.Azpect.before@line5()
java.lang.Object aspects.Azpect.around@line8()
void test.Base.advicedMethod()

java.lang.Object aspects.Azpect.around@line8()
void aspects.Azpect.before@line12()
void test.Base.advicedMethod()

| Name | Value |
|---|---|
| callContext | CallContext ($id=11) |
| joinPoint | JoinPoint ($id=8) |
| actionsToPerform | ActionOrderElement ($id=25) |
| executingAction | AttachedAction ($id=26) |
| action | MethodCallAction ($id=31) |
| contexts | Context[1] |
| scheduleInfo | ScheduleInfo ($id=33) |

```
void aspects.Azpect.before@line5()
java.lang.Object aspects.Azpect.around@line8()
    void test.Base.advicedMethod()
```

Figure 5 – A screenshot of the Join Point view.

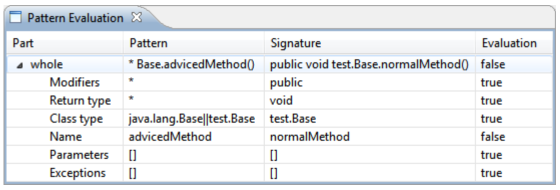


☑ aspects.Azpect.before() : (base(...)) && (target(test.Base))
☑ aspects.Azpect.before() : (base(...)) && (!(target(test.Base)))
☑ aspects.Azpect.after() : set(* (Base).someField)
☑ java.lang.Object aspects.Azpect.around() : base(...)

| Name | Value |
|---|---|
| specializations | Specialization[1] |
| action | MethodCallAction ($id=... |
| scheduleInfo | ScheduleInfo ($id=57) |

```
[* * (test.Base || java.lang.Base).advicedMethod() throws []
exposes [LazyObjectConstant]
when Callee is a (sub)type of test.Base]
```

Figure 6 – The Attachments view.



| Part | Pattern | Signature | Evaluation |
|---|---|---|---|
| whole | * Base.advicedMethod() | public void test.Base.normalMethod() | false |
| Modifiers | * | public | true |
| Return type | * | void | true |
| Class type | java.lang.Base\|\|test.Base | test.Base | true |
| Name | advicedMethod | normalMethod | false |
| Parameters | [] | [] | true |
| Exceptions | [] | [] | true |

Figure 7 – The Pattern Evaluation view.

an Atomic Predicate, whose evaluation may lead to one of two leaf nodes, each of which executes different Actions. The currently active execution path is highlighted (bold outline) together with the currently executing action (dark background).

**Attachments View** In order to dynamically deploy and undeploy attachments at runtime, the debugger provides the Attachments view shown in Figure 6. It shows all attachments that are defined in the executing program together with a checkbox that controls its deployment status in the debugged program (top), along with detailed information about the selected Attachment (center, bottom)

**Pattern Evaluation View** To debug the matching of Patterns at a join point, the Pattern Evaluation view, shown in Figure 7, visualizes their evaluation at sub-patterns granularity. Since Patterns that do not match at a join point are not shown in the Join Point view, this functionality is accessible through the Attachments view which contains all dispatch declarations in the program.

## 7 Related Work

This article's presentation of ALIA4J revolves around ALIA4J's meta-object protocol, which links LIAM and FIAL, and the improvements in debugging support facilitated by a common meta-model. The discussion of related work is thus structured accordingly.

### 7.1 Meta-Object Protocols

Several approaches, with goals as diverse as improved performance and rigorous semantic foundations, provide abstractions in the intermediate language that are closer to the source-language constructs of aspect-oriented, context-oriented, or similar languages than the abstractions offered by established intermediate languages, e.g., Java bytecode. What all these approaches have in common is that the intermediate languages are accompanied by a MOP at one of the three levels addressed by ALIA4J: runtime MOP, compile-time MOP, and just-in-time compile time MOP (cf. Section 4). Unlike ALIA4J, however, none of presented approaches supports all three levels of MOPs; in particular, no systematic approach is provided for transitioning from one level to another when switching from the language design to the optimization phase. Many of these intermediate languages and MOPs are also intrinsically tied to a specific execution strategy, which makes moving to back-ends with different strategies difficult.

### 7.1.1 Runtime MOPs for Dispatching

*JSR 292* specifies the invokedynamic instruction and its supporting API [Ros09], which can be seen as an extension to Java bytecode with an associated MOP. Together, they can be used to specify advanced-dispatching semantics, albeit at a much lower level than using LIAM. The language implementer is nevertheless barred from the optimization opportunities offered by user-defined bytecode or machine code generation; despite its low-level design, the JSR 292 meta-model is a pure runtime MOP.

The *Nu* project [DR10] extends Java bytecode with two instructions which realize dynamic aspect deployment and undeployment: bind and remove. The bind instruction hereby takes both a Pattern and a Delegate object, which act as pointcut and advice, respectively, and returns a BindHandle. The remove takes this handle and undoes the corresponding binding. While parts of this runtime MOP are declarative (Pattern), others require an imperative definition (Delegate, execution order of aspects).

The *Reflex* project [Tan06] uses dynamic bytecode instrumentation to provide behavioral reflection. It's runtime MOP is used to link so-called hooksets, which select, e.g., classes or methods, to meta-objects, which are Java classes that may be implicitly instantiated. Each link specifies which method of the meta-object is to be called and is further configured by link attributes. However, this model is not very fine-grained. Also, despite the fact that Reflex relies on dynamic bytecode instrumentation, it does not expose this capability to language implementers.

The delegation-based execution model for the *Multi-Dimensional Separation of Concerns* (delMDSOC) [SJHH08] defines several primitive operations along with operational semantics that allow formal reasoning about language constructs. The model's expressiveness is shown by realizing Java-like, AspectJ-like, and context-oriented languages in it. The delMDSOC model is not declarative in the definition of dynamic behavior; instead, language constructs are represented by imperative and often program-specific code. A declarative model of context exposure is also missing.

The *Java Aspect Metamodel Interpreter* (JAMI) [HBA08] defines a meta-model, not unlike LIAM, to capture the semantics of features in aspect-oriented languages. But unlike LIAM, JAMI is tied to a specific execution strategy: interpretation. Meta-model refinements must resort to using reflection; code generation cannot be realized.

*Pinocchio* [VBG$^+$10] uses (a tower of) first-class interpreters to allow for changes to a languages semantics, including but not limited to dispatching. Pinocchio uses abstract syntax trees as its single intermediate language, whereas ALIA4J relies on a combination of Java bytecode and LIAM-based IR, which, just like Pinocchio's tower of interpreters, ensures that base and meta-level are cleanly separated.

### 7.1.2 Compile-Time MOPs

The AspectBench Compiler (abc) [ACH$^+$06] is an extensible compiler for the AspectJ language. Its front-end and back-end are decoupled by using an abstract syntax tree (AST) as intermediate language. To implement a language extension, the language designer has to introduce new AST node types. These are, along with predefined types representing key aspect-oriented concepts, transformed to instances of abc-specific classes similar to ALIA4J's meta-model. As is the case in ALIA4J, the classes have to be refined for new language concepts. Such a refinement must provide both a method that attempts to statically evaluate the concept and a fallback method that generates Java bytecode if static evaluation is not possible. The language designer is thus *always* confronted with low-level implementation details. In our approach a language designer can choose to stay at the high abstraction level of our runtime MOP, e.g., to create quick prototypes of new language features or if the performance achieved in this approach is sufficient; if needed, a lower-level and more efficient implementation can later be supplanted—possibly by another specialist—without breaking clients.

### 7.1.3 Just-in-Time Compile Time MOPs

The *org.vmmagic framework* [FBC$^+$09] in the Jikes RVM provides low-level primitives to a VM written in a high-level language, namely Java. The framework achieves extensibility by so-called intrinsic methods following a special naming convention. The methods' Java implementation serves merely as a placeholder; when the just-in-time compiler encounters a call to such a placeholder, it instead inserts the corresponding machine code. In the org.vmmagic framework, this translation is hard-wired in the JIT compiler, whereas in ALIA4J each primitive language concept is implemented modularly and separate from the JIT.

The *Klein virtual machine* [USA05] is a Self VM written in Self. Like the org.vmmagic framework it offers primitives not implemented (or even implementable) in application code. For those, Klein provides several ways of implementing them: generating corresponding machine code or implementing the primitive in a lower-level, restricted variant of Self. Moreover, Klein provides an API, which can be used by the compiler implementer to factor out the code sequences realizing common functionality. The ability to implement a primitive through code-generation or through a callout to a method is similar to ALIA4J's notions of JIT compile-time MOP and runtime MOP. In contrast to ALIA4J's extensible meta-model the API of Klein is fixed.

The *XIR language* [TWSC10] is a recent example mainly driven by the goal of improving compiler-runtime separation. With it, the developer of the runtime uses so-called snippets written in an assembler-like embedded DSL to communicate the implementation strategy for essential runtime features like method calls to the developer of the compiler. Like XIR, ALIA4J's MOP was designed as a clear interface, albeit not between implementers of runtime and compiler but between the language designer and language implementer. As such, ALIA4J by default exposes the language designer to less low-level detail; it is easily possible to realize a concept's semantics in a reflective, interpretative style only. If, however, more control over code generation for LIAM entities is desired, a language like XIR could be employed to good effect.

## 7.2 Debuggers

As the ALIA4J approach can be used to implement both existing general-purpose programming languages of, e.g., the aspect-oriented or predicate-dispatching paradigms and domain-specific languages (cf. Section 5.1), we will now review the state-of-the-art in debugging support for these classes of languages.

### 7.2.1 Tools Dedicated to one Language

Several researchers discuss debuggers for aspect-oriented programming (AOP) that provide information closer to the source code, such as the composite source code in Wicca [EAH+07], the aspect-aware breakpoint model in AODA [DBLJ09], or the identified AOP activities in TOD [PT08]. Nevertheless, all of these debuggers use only the transformed IR of the underlying language. AOP-specific abstractions, e.g., aspect-precedence declarations, and their locations in the source code are partially or even entirely lost after compilation.

For some advanced-dispatching languages, dedicated integrated development environments (IDEs) offer tools for analyzing the programs. Common IDE tools for AOP languages, like the JPred Eclipse plug-in,[9] the AspectJ Development Tools,[10] and the CaesarJ Development Tools,[11] only provide static code visualizations. The *ObjectTeams Development Tools* (OTDT) [HHM+06] go a step further and enhance the user interface of the standard Java debugger by filtering out call frames that belong to infrastructural code and subsequently adapting the placement of breakpoints. They also provide a view that shows so-called "Teams" and allows to dynamically enable and disable them, similar to the Attachments view of our debugger (cf. Figure 5).

### 7.2.2 Tool generators for domain-specific languages

To compile new languages embedded in host languages, *Helvetia* [RGN10] defines transformation rules used in the compilation process. Helvetia provides tools for the host language which have explicit extension points for usage by the embedded languages. By using these transformation rules and extensions points, existing tools like a debugger can be applied to new languages. The *TIDE* [vdBCOV05] environment is a generic debugging framework that can be instantiated for new DSLs. *LISA* [HPV+04] is an attribute-grammar-based compiler generator that can automatically generate several language-based tools, like editors, debuggers, and animators, from language specifications by identifying generic and specific parts.

Just like conventional compilation approaches, these tools require that code is generated in an established language, with the same limitations as discussed in already this paper: Not all new language constructs are explicit in the generated code and source locations may be lost. Furthermore, with both TIDE and LISA, it is the generation process that is generic rather than the product. In contrast, our debugger is based on a semantic meta-model, which can be targeted by many syntactic variations.

## 8 Summary and Directions for Future Research

In this article, we have presented the ALIA4J approach to implementing language extensions. Phrasing them in terms of advanced-dispatching enables us to implement numerous languages, from AspectJ to new, domain-specific languages, using just a few core abstractions. With a fine-grained intermediate representation close to the source-level abstractions, re-using the implementation and optimization of language sub-constructs is possible without the need to syntactically extend another language.

[9]See `http://eclipse-plug130.sourceforge.net/`.
[10]See `http://www.eclipse.org/ajdt/`.
[11]See `http://caesarj.org/`.

ALIA4J provides an execution model independent of any concrete JVM extensions and facilitates the modular implementation of a language construct's semantics. Language implementers can freely choose the most appropriate implementation strategy, be it interpretation, bytecode generation, or machine code generation. Because optimizations are implemented modularly, they can also be re-used. Likewise, dedicated tools for analyzing advanced-dispatching-based programs can be re-used for new languages, as is shown by the debugger presented in this paper; due to ALIA4J's genericity, the debugger allows reasoning about all source language constructs that affect dispatch while being completely independent of any particular source language.

This increase in re-use allows programming-language researchers and designers of domain-specific languages to focus on their immediate task: developing source languages that solve the problem at hand. Already established language sub-constructs do not have to be implemented anew. We believe that this can improve the quality of language prototypes, not only in terms of performance but also in terms of correctness, as low-level details need to be implemented only once. Our approach also facilitates using the prototypical, unoptimized implementations as *test oracles* because they share the interface with the optimized implementation, namely the factory used to define dispatch declarations; these test oracles can then be used to ensure correctness. However, future studies are needed to quantify these improvements.

A future direction for research enabled by our work on ALIA4J are additional tools, besides the aforementioned debugger, that exploit ALIA4J's declarative, first-class intermediate representation and execution model: Profilers or code coverage analyses come to mind here. One can even imagine using model checking based on ALIA4J's intermediate representation to reason about the possible outcomes of the dispatch function; for programs that utilize dynamic deployment, all possible combinations of deployed and undeployed Attachments can be enumerated by the model checker.

Another direction for future research is the composition of multiple source languages that have all been mapped to the same language-independent meta-model: LIAM. It may thus be possible to combine, e.g., AspectJ and JPred within a single program without unwanted interferences caused by low-level code transformations. A detailed study of such interactions between different languages, however, has yet to be done.

## References

[AAB$^+$05] Bowen Alpern, Steve Augart, Steve M. Blackburn, Maria Butrico, Antony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn S. McKinley, MarkMergen, J. Eliot B. Moss, Ton Ngo, and Vivek Sarkar. The Jikes virtual machine research project: Building an open-source research community. *IBM Systems Journal*, 44:399–417, 2005. `doi:10.1147/sj.442.0399`.

[ACH$^+$06] Pavel Avgustinov, Aske Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I (TAOSD)*, volume 3880 of *LNCS*, pages 293–334. Springer Berlin/Heidelberg, 2006. `doi:10.1007/11687061_9`.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of CaesarJ. In *Transactions on Aspect-Oriented Soft-*

*ware Development I (TAOSD)*, volume 3880 of *LNCS*, pages 135–173. Springer Berlin/Heidelberg, 2006. doi:10.1007/11687061_5.

[AN08] Irem Aktug and Katsiaryna Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of the International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM)*, 2008. doi:10.1016/j.entcs.2007.10.013.

[BHM06] Christoph Bockisch, Michael Haupt, and Mira Mezini. Dynamic virtual join point dispatch. In *Proceedings of the International Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2006.

[BHMO04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2004. doi:10.1145/976270.976282.

[BKH$^+$06] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006. doi:10.1145/1167473.1167484.

[BMAK11] Christoph Bockisch, Somayeh Malakuti, Mehmet Akşit, and Shmuel Katz. Making aspects natural: Events and composition. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2011. doi:10.1145/1960275.1960312.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986. doi:10.1109/TC.1986.1676819.

[BSMA11] Christoph Bockisch, Andreas Sewe, Mira Mezini, and Mehmet Akşit. An overview of ALIA4J: An execution model for advanced-dispatching languages. In *Proceedings of the Conference on Objects, Models, Components, Patterns (TOOLS)*, 2011. doi:10.1007/978-3-642-21952-8_11.

[BSZ11] Christoph Bockisch, Andreas Sewe, and Martin Zandberg. ALIA4J's [(just-in-time) compile-time] MOP for advanced dispatching (position paper). In *Proceedings of the Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2011. doi:10.1145/2095050.2095101.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999. doi:10.1145/320384.320407.

[DBLJ09] Wouter De Borger, Bert Lagaisse, and Wouter Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2009. doi:10.1145/1509239.1509263.

[DR10] Robert Dyer and Hridesh Rajan. Supporting dynamic aspect-oriented features. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):7:1–7:34, 2010. doi:10.1145/1824760.1824764.

[dRHH+08] Arjan de Roo, Michiel Hendriks, Wilke Havinga, Pascal Dürr, and Lodewijk Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *Proceedings of the Workshop on Academic Software Development Tools and Techniques (WASDeTT)*, 2008.

[EAH+07] Marc Eaddy, Alfred V. Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In *Software Composition*, volume 4829 of *LNCS*, pages 200–215. Springer Berlin/Heidelberg, 2007. `doi:10.1007/978-3-540-77351-1_17`.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007. `doi:10.1145/1297027.1297029`.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998. `doi:10.1007/BFb0054092`.

[FBC+09] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the Conference on Virtual Execution Environments (VEE)*, 2009. `doi:10.1145/1508293.1508305`.

[HBA08] Wilke Havinga, Lodewijk Bergmans, and Mehmet Akşit. Prototyping and composing aspect languages: Using an aspect interpreter framework. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2008. `doi:10.1007/978-3-540-70592-5_9`.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology (JOT)*, 7(3), 2008. `doi:10.5381/jot.2008.7.3.a4`.

[HHM+06] Stephan Herrmann, Christine Hundt, Marco Mosconi, Jan Wloka, and Carsten Pfeiffer. Das ObjectTeams Development Tooling. *Softwaretechnik-Trends*, 26(4):42–43, 2006.

[HPV+04] Pedro Rangel Henriques, Maria Joao Varanda Pereira, Maria João Var, Mitja Lenič, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using the LISA system. *IEE Proceedings: Software*, 152(2), 2004. `doi:10.1049/ip-sen:20041317`.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001. `doi:10.1007/3-540-45337-7_18`.

[MFRW09] Todd Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2), 2009. `doi:10.1145/1462166.1462168`.

[MK03] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the European Conference* *on Object-Oriented Programming (ECOOP)*, 2003. `doi:10.1007/978-3-540-45070-2_2`.

[Nag06] Istvan Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* PhD thesis, CTIT, University of Twente, May 2006. ISBN 90-365-2368-0.

[PT08] Guillaume Pothier and Éric Tanter. Extending omniscient debugging to support aspect-oriented programming. In *Proceedings of the Symposium on Applied computing (SAC)*, 2008. `doi:10.1145/1363686.1363753`.

[RGN10] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2010. `doi:10.1007/978-3-642-14107-2_19`.

[Ros09] John R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2009. `doi:10.1145/1711506.1711508`.

[SBM08] Andreas Sewe, Christoph Bockisch, and Mira Mezini. Redundancy-free residual dispatch: Using ordered binary decision diagrams for efficient dispatch. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2008. `doi:10.1145/1394496.1394497`.

[SJHH08] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008. `doi:10.1145/1449764.1449806`.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2003. `doi:10.1145/643603.643606`.

[Tan06] Éric Tanter. An extensible kernel language for AOP. In *Proceedings of the Workshop on Open and Dynamic Aspect Languages (ODAL)*, 2006.

[TT06] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Distributed Applications and Interoperable Systems (DAIS)*, volume 4025 of *LNCS*, pages 316–331. Springer Berlin/Heidelberg, 2006. `doi:10.1007/11773887_25`.

[TWSC10] Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. Improving compiler-runtime separation with XIR. In *Proceedings of the Conference on Virtual Execution Environments (VEE)*, 2010. `doi:10.1145/1735997.1736005`.

[USA05] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005. `doi:10.1145/1094855.1094865`.

[VBG$^+$10] Toon Verwaest, Camillo Bruni, David Gurtner, Adrian Lienhard, and Oscar Niestrasz. Pinocchio: bringing reflection to life with first-class interpreters. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA/Onward!)*, 2010. doi:10.1145/1869459.1869522.

[vdBCOV05] Mark van den Brand, Bas Cornelissen, Pieter Olivier, and Jurgen Vinju. TIDE: A generic debugging framework. In *Proceedings of the Workshop on Language Descriptions, Tools, and Applications (LDTA)*, 2005. Tool Demo. doi:10.1016/j.entcs.2005.02.056.

[YBA+11] Haihan Yin, Christoph Bockisch, Mehmet Akşit, Wouter De Borger, Bert Lagaisse, and Wouter Joosen. Debugging scandal—the next generation. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, 2011.

[YBA12] Haihan Yin, Christoph Bockisch, and Mehmet Akşit. A fine-grained debugger for aspect-oriented programming. In *Proceedings of MODULARITY:aosd*, 2012. doi:10.1145/2162049.2162057.

**Acknowledgments** We would like to thank everyone who has contributed to ALIA4J in the past few years (in alphabetical order): Matthew Arnold, Remko Bijker, Tom Dinkelaker, Sebastian Eifert, Sarah Ereth, Pascal Flach, Michael Haupt, Michael Hausl, Jannik Jochem, Sebastian Kanthak, Michael Krebs, Andre Loker, Markus Maus, Suraj Mukhi, Heiko Paulheim, Nico Rottstädt, Christian Rüdiger, Jan Sinschek, Kai Stroh, Zied Trabelsi, Nathan Wasser, and Martin Zandberg. This work was supported by the Center for Advanced Security Research Darmstadt (www.cased.de).

## About the authors

**Christoph Bockisch** is an assistant professor in computer science at the Software Engineering group, Universiteit Twente. His research focus is on the design and implementation of programming languages with advanced modularity mechanisms. Contact him at c.m.bockisch@utwente.nl.

**Andreas Sewe** is a PhD candidate in computer science at the Software Technology Group, Technische Universität Darmstadt. His research focus is on virtual machines and their interaction with high-level languages. Contact him at sewe@st.informatik.tu-darmstadt.de.

**Haihan Yin** is as PhD candidate in computer science at the Software Engineering group, Universiteit Twente. His research focus is on software comprehension tools for aspect-oriented programs. Contact him at h.yin@cs.utwente.nl.

**Mira Mezini** is a full professor in computer science at Technische Universität Darmstadt, where she leads the Software Technology Group. Her research interests encompass programming language design and implementation, with a focus on advancing mechanisms for modular software development. Contact her at mezini@cs.tu-darmstadt.de.

**Mehmet Akşit** is a full professor in computer science at the Universiteit Twente, where he leads the Software Engineering group. His research interests encompass programming language design, with a focus on object-based composition. Contact him at aksit@cs.utwente.nl.