# An Overview of ALIA4J

## An Execution Model for Advanced-Dispatching Languages

Christoph Bockisch[1], Andreas Sewe[2], Mira Mezini[2], and Mehmet Akşit[1]

[1] Software Engineering group, University of Twente, The Netherlands
{c.m.bockisch,aksit}@cs.utwente.nl
[2] Software Technology group, Technische Universität Darmstadt, Germany
{sewe,mezini}@st.cs.tu-darmstadt.de

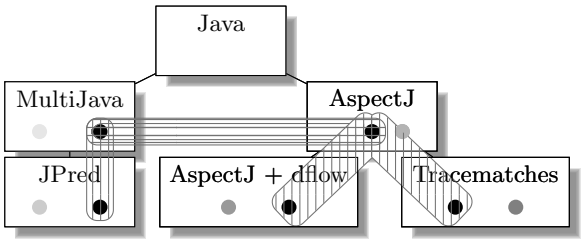
**Abstract.** New programming languages that allow to reduce the complexity of software solutions are frequently developed, often as extensions of existing languages. Many implementations thus resort to transforming the extension's source code to the imperative intermediate representation of the parent language. But approaches like compiler frameworks only allow for re-use of code transformations for syntactically-related languages; they do not allow for re-use across language families. In this paper, we present the ALIA4J approach to bring such re-use to language families with advanced dispatching mechanisms like pointcut-advice or predicate dispatching. ALIA4J introduces a meta-model of dispatching as a rich, extensible intermediate language. By implementing language constructs from four languages as refinements of this meta-model, we show that a significant amount of them can be re-used across language families. Another building block of ALIA4J is a framework for execution environments that automatically derives an execution model of the program's dispatching from representations in our intermediate language. This model enables different execution strategies for dispatching; we have validated this by implementing three execution environments whose strategies range from interpretation to optimizing code generation.

## 1 Introduction

A recent IBM whitepaper [23] identifies complexity as the most relevant factor in the software development process: A reduction of complexity is directly proportional to an improvement of the overall process. Accidental complexity, i.e., complexity not inherent to the problem solved by a program, is mainly caused by the inability to accurately represent the conceptual solution in a given programming language. Thus, research in programming languages produces many new languages with mechanisms to structure a program in a way more suitable to conceptual solutions. The key technique here is *abstraction* where one concrete program module does not refer to another explicitly, but only abstractly specifies the functionality or data to be used. The relevance of abstraction can be seen in the continuous progress in the history of programming language research [24],

resulting in advanced abstraction mechanisms like multiple [10] and predicate dispatching [15], pointcut-advice[3] [20], or context-oriented programming [18].

Many new languages employing these mechanisms are extensions of Java: MultiJava [11], JPred [21], AspectJ [19], CaesarJ [2], Compose*/Java [12], ContextJ [18], etc. Some of these are further extended by others; thus, languages and their extensions can be arranged in a genealogical tree, with languages of different paradigms being siblings, as exemplified below for a few languages.




```
1 Shape intersect(Shape s)
2   when s@Rectangle { ... }
```

```
1 after() : call(* Shape.intersect(..))
2   && args(Rectangle) { ... }
```

Language constructs provided by the individual languages are presented as dots in different shades of gray in the figure. The black dot represents a concept shared by all languages except Java, e.g., resolution of abstractions based of argument values. Vertical and horizontal overlap of the languages with regard to this construct is highlighted by the rounded boxes, hatched vertically and horizontally, respectively. But as the two listings to the right show, languages like JPred (top) and AspectJ (bottom) express the same concept using different notations: a predicate (`s@Rectangle`) respectively a pointcut designator (`args(Rectangle)`).

*Dispatching* is the mechanism that resolves abstractions and binds concrete functionality to their usage, e.g., when invoking `Shape.intersect` above. Abstractions commonly found in programming languages influence the resolution of method calls and field accesses. In the following, we use the term *dispatch site* uniformly to refer to sites of both method calls and field accesses in a program. A common example of dispatching is receiver-type polymorphism: Whenever a virtual method is invoked, the runtime environment chooses from among different functionalities (i.e., the overriding methods) and transfers control to the one alternative applicable in the current program state (i.e., corresponding to the dynamic receiver type). We call languages that go beyond 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).

The implementation of a programming language 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 emits a code representation conforming to the intermediate language. The back-end either exe-

[3] A particular flavor of aspect-oriented programming (AOP).

cutes this *intermediate representation* (IR) directly or further compiles it into a machine-executable form. Typically, implementations of new languages build on the back-ends of established languages; thus, their front-ends have to emit IR in an intermediate language tailored to a different source language. For the aforementioned source languages, e.g., only the parent (Java) provides its own intermediate language (Java bytecode).

The resulting *semantic gap* between source and intermediate language, i.e., the inability of the intermediate language to express the new mechanisms directly, requires transforming the high-level language concepts to low-level imperative code. Compiler frameworks support this task by means of code transformations [22, 14, 3]. They only support re-use along the *vertical* dimension as they require a language to be a syntactic extension of another in order to re-use its implementation; *horizontal* re-use is not possible. While code transformations defined on the common intermediate language are shared among all language extensions, they cannot exploit knowledge about source language constructs, which is lost during the transformation to the common intermediate language.

In this paper, we present the ALIA4J approach[4] 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 can act as an intermediate language, thereby closing the semantic gap that currently exists between these source languages and their parent's intermediate language. Furthermore, re-using the implementation of horizontally overlapping constructs becomes viable.

For executing code defined in the intermediate language, we provide several back-ends, including platform-independent ones. These back-ends instantiate a framework that can automatically derive an execution model from the advanced-dispatch's intermediate representation. As the execution model retains the IR's declarative nature, the back-end is free to chose from different *execution strategies*, ranging from interpretation to optimizing code generation.

The goal of ALIA4J is to ease the burden of programming-language implementation resting upon both researchers of new abstraction concepts and designers of domain-specific languages. It should be emphasized that our approach is concerned with the *execution* semantics of the different languages. They may differ greatly in the way language (sub-)constructs are used or combined. Based on this, the languages can make different guarantees on the program behavior or perform different semantic checks. For example, in the case of predicate dispatching, a compiler ensures that there is always exactly one applicable predicate method at runtime. Performing syntactic and semantic checks is the responsibility of a language's compiler and not covered by our approach.

The contributions of this work are threefold:

1. We introduce advanced-dispatching as an execution model.
2. We provide a meta-model for advanced dispatching. Its generality is shown by refining it with (sub-)constructs of the languages AspectJ, Compose*,

[4] See `http://www.alia4j.org/`.

CaesarJ, JPred, ConSpec, and several domain-specific languages; the overlap in refinements used by these languages shows their re-usability.
3. For executing the advanced-dispatch IR, we provide a framework that does not impose any particular execution strategy on the back-end and demonstrate this freedom of choice by providing three back-ends based on different execution strategies: STEAMLOOM$^{ALIA}$, SiRIn, and NOIRIn.

In the following section, we discuss approaches related to ours and their limitations. The ALIA4J approach, including the meta-model and the framework, is fully presented in Sect. 3 and evaluated in Sect. 4. Section 4.1 describes how to map existing and new languages to our approach, thus demonstrating re-usability of meta-model refinements. Section 4.2 outlines the different framework instantiations, proving the independence of our execution model from a back-end's execution strategy. Finally, Sect. 5 concludes and discusses future work.

## 2 Related Work

Several approaches provide abstractions in the intermediate language that are closer to the source-language constructs of aspect-oriented, context-oriented, or similar languages than established intermediate languages. The immediate goals of these approaches range from improving performance to providing a precise operational semantics of the intermediate language. Nevertheless, they also facilitate horizontal re-use of the implementation of the constructs added to the intermediate language. But as the granularity of the added abstractions is very coarse, many re-use opportunities are still missed. Furthermore, intermediate languages and the definition of their semantics are tied to a specific execution strategy in all cases; this hinders moving to back-ends with different strategies.

The *Nu* project [13] extends Java bytecode with two instructions supporting aspect-oriented programming: *bind* and *remove*. By means of these primitives, dynamic deployment and undeployment of aspects can be realized. The bind instruction expects two arguments: a *Pattern* object selecting relevant code locations by means of their syntactic and lexical properties and a *Delegate* object specifying a method to execute as advice. It returns a *BindHandle*, which then may be passed as argument to the *remove* primitive to undo a specific binding. Nu requires an imperative definition of Delegates and other concepts like the execution order of aspects; it only supports access to a limited set of context values. Nu's two primitives are implemented on top of the HotSpot Java virtual machine, which has been modified to accept the extended IR.

The *Reflex* project [27] provides behavioral reflection implemented through dynamic bytecode instrumentation. *Hooksets* are expressions over properties of structural abstractions of the code, like classes or methods. *Links* associate hooksets and *metaobjects* which are Java classes that may be implicitly instantiated. A link specifies which method of the metaobject is to be called and is configured by link attributes. While some attributes are first-class entities in Reflex, this model is not very fine-grained. As a consequence, their implementation cannot be re-used in the implementation of language (sub-)constructs that *partially*

map to existing activation conditions or parameterizations. Parameters as well as scopes cannot be user-defined and extending the available parameters and scopes requires a modification of the Reflex framework.

Schippers et al. [25] present a delegation-based execution model for the *Multi-Dimensional Separation of Concerns* (delMDSOC). They define primitive operations in their execution model and provide an operational semantics that allows 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 missing.

The *Java Aspect Metamodel Interpreter* (JAMI) [17] defines a meta-model to capture the semantics of features in aspect-oriented languages. Due to JAMI's interpreter approach, meta-model refinements must resort to using reflection and optimizing code generation cannot be realized.

## 3 The ALIA4J Architecture

In this section, we present the *Advanced-dispatching Language-Implementation Architecture for Java* (ALIA4J) that facilitates both vertical and horizontal reuse of implementations of all language (sub-)constructs governing dispatch. Predecessors of ALIA4J have been the subject of earlier work [8, 5]. ALIA4J has two main components: The *Language-Independent Advanced-dispatching Meta-model* (LIAM), a common meta-model for expressing advanced-dispatch declarations as well as relations between them, and the *Framework for Implementing Advanced-dispatching Languages* (FIAL), a framework for execution environments that handle LIAM-based advanced-dispatch intermediate representations.

### 3.1 Components of ALIA4J

Figure 1 shows the architecture of our proposed approach. It is centered around LIAM, a meta-model of primitive concepts participating in advanced dispatch. When implementing a new language following the ALIA4J approach, the building blocks of the language's semantics must be concretized by either re-using existing meta-model refinements, implementing new refinements, or a mixture of both; this yields a language-specific LIAM refinement. When compiling a program in the new language, the compiler needs to separate the advanced dispatch declarations from those parts directly expressible in Java. From the former, a program-specific advanced-dispatch IR conforming to the refined, language-specific meta-model is created; the latter are turned directly into Java bytecode.

When executing a program, the FIAL framework (top right) derives an execution model for each dispatch site—i.e., for each method call, field read or write—from the program-specific advanced-dispatching IR. To this end, FIAL processes the IR but only refers to it in terms of the language-independent LIAM

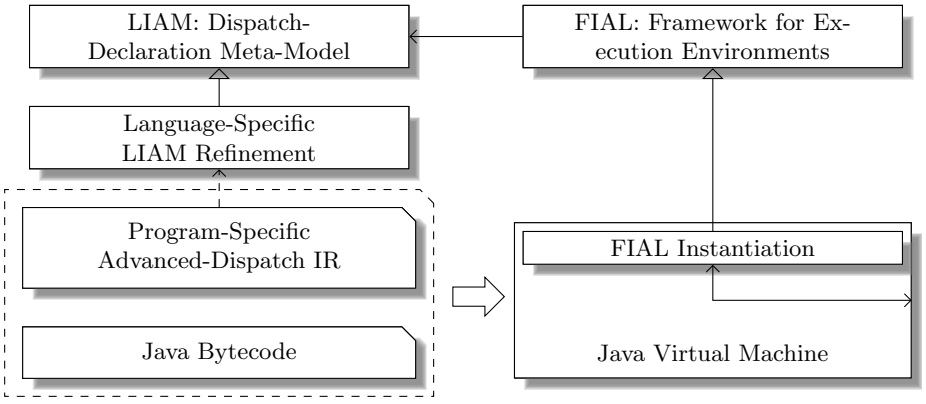



**Fig. 1.** Overview of an ALIA4J-based language implementation.

entities; thus, FIAL and its instantiations are de-coupled from the given source language.

Since our targeted languages are based on the Java platform, we expect that FIAL is instantiated as a plug-in or extension for an existing Java virtual machine (bottom right). By interacting with this JVM, the FIAL instantiation implements dispatch as mandated by the provided execution model, e.g., by interpretation or different code generation approaches (cf. Sect. 4.2). FIAL itself handles services like dynamic class loading and dynamic deployment, i.e., to add or remove intermediate representations of advanced-dispatch at runtime. FIAL instantiations only need to implement a few well-defined interfaces and LIAM refinements are not at all concerned with these services' implementation.

FIAL offers four generic services required by any execution environment supporting LIAM-based advanced-dispatch IR:

1. FIAL assists in deploying and undeploying such IR at runtime.
2. It handles dynamic class loading in the presence of dispatch IR already deployed.
3. It can trigger an *importer* component which transforms advanced dispatch declarations from the source language to the intermediate representation.
4. From the currently deployed advanced-dispatch IR it derives an execution model for each dispatch site in the executed program.

To derive a dispatch site's execution model, FIAL partially evaluates the LIAM-based IR and constructs the *dispatch function* for the dispatch site combining all individually declared dispatch predicates. In the ALIA4J approach, the result of a dispatch function can be composed of multiple actions; it is a Boolean function $f : \mathbb{B}^n \to \mathbb{B}^m$ that characterizes which of the $m$ actions should be executed when the dispatch site is reached, depending on the evaluation of $n$ predicates. A detailed discussion of the construction of dispatch functions [26], and of partially evaluating LIAM-based IR and resolving relations between dispatch declarations [7, Sect. 5] is found elsewhere.

### 3.2 The Meta-Model of Advanced Dispatching

Figure 2 shows a UML class diagram of LIAM's meta-entities for the declaration of advanced dispatch, termed an *Attachment*, and relations between such declarations. An Attachment specifies which *functionality* should execute (*Action*) at which *join points*[5] (*Specialization*) and *when* it should execute relative to the join point (*Schedule Info*), i.e., before, after, or around. The Specialization entity is divided into entities specifying static (*Pattern*) and dynamic (*Predicate*) properties of selected join points as well as a list of values (*Context*) which must be exposed to the Action at selected join points. Hereby, a Pattern specifies syntactic and lexical properties of instructions executing at a join point. These instructions are generally connected to a member, e.g., the target method for an invocation. Patterns are composed of multiple sub-patterns matching on the different elements of the member's signature like the name or parameter types [4]. A Predicate is a Boolean expression of *Atomic Predicate* entities modeling conditions on a join point's dynamic state.

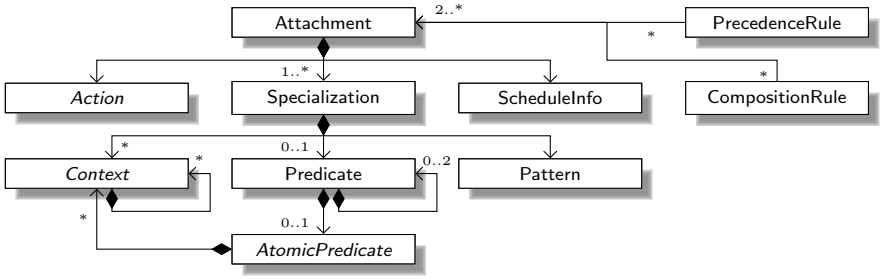



**Fig. 2.** Entities of the Language-Independent Advanced-dispatching Meta-Model.

As Fig. 2 shows, it is not only a Specialization that can refer to a Context to specify that this context value is exposed to an Action; Atomic Predicate and Context itself can also refer to Contexts. This means that the evaluation of Atomic Predicates and Contexts, respectively, depends on the exposure of further context values.[6] For example, a refinement of Context that realizes the reflective **thisJoinPoint** keyword of AspectJ declares its dependency on the individual context values that it composes, like argument values passed to the join point and whether the join point is a method call or field access.

Relations between Attachments are defined in terms of *Precedence Rules* and *Composition Rules*. Both kinds of rules govern the execution of Actions jointly applicable at the same join point. The former rules specify a partial order among the Actions and the latter rules specify which Actions must or must not be executed together. In all cases, a relation between Attachments

[5] The term, borrowed from AOP, refers to a specific execution of a dispatch site.
[6] Circular dependencies must be ruled out by the front-end.

carries over to the Actions contributed by the Attachments. The entities printed in italics in Fig. 2, i.e., Action, Atomic Predicate and Context, can be refined with the specific sub-constructs of a language being implemented in the ALIA4J approach. All other entities represent logical groupings of the refinable entities. They are fixed and used by FIAL to partially evaluate LIAM-based IR.

The listing below shows an AspectJ aspect with one pointcut-advice. This aspect will be compiled to a class with the name A and a method, say before_0(), containing the body of the **before** advice. The aspect's instantiation strategy is to create a singleton instance of A and always invoke the method thereon.

```
1 aspect A issingleton() {
2   before() : call(* *.m(..)) { /* advice body */ }
3 }
```

Figure 3 shows the LIAM-based IR for the pointcut-advice in this example. This example is minimalistic on purpose and does not use all of LIAM's features; section 4.1 discusses creating our IR from advanced-dispatch declarations in different languages, including AspectJ, in detail. At the moment, just note that AspectJ pointcuts are expressed by Specializations in LIAM. But Specializations also have additional purposes, for instance, they refer to a Context entity that realizes the aspect's instantiation strategy. In the example, PerTupleContext realizes the **issingleton** strategy. The Action maps to the advice functionality and the Schedule Info maps to the keyword **before**, **after** or **around**.

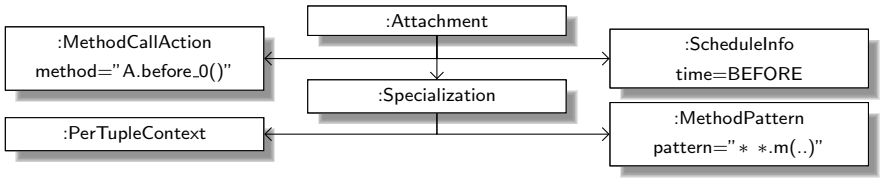



**Fig. 3.** Example of a LIAM-based advanced-dispatching IR.

### 3.3 FIAL and LIAM in Practice

The execution model of FIAL gives rise to both default compilation and interpretation strategies for dispatch sites. Either can be pursued by a FIAL instantiation. This facilitates a modular implementation of a LIAM entity's semantics in terms of a plain Java method, referred to as the entity's "compute" method.

When using the default code generation, the execution model is traversed depth-first until a LIAM entity is reached that does not depend on another one. For such a leaf, code is generated to invoke the "compute" method. In case of, e.g., a Context, this "compute" method returns the modeled value, which can then be passed to the "compute" method of the entity depending on the

Context, and so forth. Glue code is generated to ensure the correct evaluation of the dispatch function, depending on the result values of the Atomic Predicates.

A variation to this default compilation strategy is to delegate bytecode generation to the LIAM entity itself rather than just generating a call to its "compute" method. Because the bytecode-generation method is called individually for each dispatch site, its static context can be considered and the bytecode can be tailored to each site. Both strategies can be mixed freely; a LIAM entity must simply implement a "compute" method or one that directly emits Java bytecode.

As an example of a LIAM entity, consider the JoinPointKindContext presented below. It represents a string value describing the kind of the join point, accessible via **thisJoinPoint**.getKind() in AspectJ. The entity passes a signature Context to its super-constructor (line 3), thus stating that it depends on this Context, which returns the signature of the member associated with the current join point. As a consequence, a signature object is passed to the method getObjectValue,[7] the "compute" method, whenever the JoinPointKindContext is to be evaluated. In the example, this method picks one of the constant values defined in the JoinPoint class from the AspectJ runtime library appropriate to the signature (lines 6 ff.).

```
1 public class JoinPointKindContext extends Context {
2   public JoinPointKindContext() {
3     super(Collections.singletonList(ContextFactory.findOrCreateSignatureContext()));
4   }
5   public Object getObjectValue(Object liamSignature) { // "compute" method
6     if (liamSignature instanceof FieldReadSignature)
7       return JoinPoint.FIELD_GET;
8     else ...
9 } }
```

An alternative implementation declaring BytecodeSupport (line 1) is presented below. Its method emitting bytecode for a specific dispatch site (lines 5–10) inspects the signature of the associated member (line 6) and simply emits an instruction fetching the appropriate constant (line 7 ff.). Because the generated bytecode does not contain conditional control flow, it is more efficient than the "compute" method. No required Contexts have to be declared (line 3) as evaluation of this Context now does not depend on the signature Context.

```
1 public class JoinPointKindContext extends Context implements BytecodeSupport {
2   public JoinPointKindContext() {
3     super(Collections.<Context>emptySet());
4   }
5   public void build(BytecodeBuilder builder, GenericFunction site) {
6     if (site.getSignature() instanceof FieldReadSignature)
7       builder.appendGetstatic(JOIN_POINT_CLASS, "FIELD_GET",
8         TypeDescriptorConstants.STRING_CLASS);
9     else ...
10 } }
```

[7] The name, parameters and return type of a "compute" method must follow naming conventions that are ruled by methods not shown in this example.

The generation of bytecode for a LIAM entity may also depend on the actual execution strategy of the back-end. Therefore, ALIA4J uses Abstract Factories to create LIAM entities. A FIAL-based execution environment can override the factory methods for those entities for which back-end-specific bytecode can be generated; this is completely transparent to the front-end.

## 4 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.

### 4.1 Evaluation of LIAM

To validate our approach, we have refined LIAM with the concrete language subconstructs found in several languages. In the following, we will briefly discuss these refinements. For a full discussion of AspectJ, CaesarJ, Compose*, JPred, MultiJava, and ConSpec as well as the necessary LIAM refinements, we refer to our electronic appendix.[8] For the languages AspectJ and ConSpec we provide importers that automatically map source code to program-specific LIAM models.

**AspectJ** The AspectJ compiler creates a class for each aspect, with a virtual method for each advice. The aspect's instantiation strategy, defined in the "per-clause", specifies whether a new instance of this class must be created at a join point or an existing instance is to be used. In either case, the virtual methods compiled from the advice are invoked on this instance. When mapped to LIAM, an aspect's instantiation strategy is represented by a Context: The **pertarget**, **perthis** and **issingleton** strategies are mapped to a PerTupleContext, which associates a tuple of input values with a lazily created instance of the aspect class; for the former two a 1-ary tuple containing a CalleeContext or a CallerContext is used, for the latter a 0-ary tuple. The **percflow** and **percflowbelow** strategies are mapped to a PerCFlowContext and PerCFlowBelowContext, respectively. Each Specialization refers to the Context representing the instantiation strategy as its first exposed Context. All pointcuts defined in an aspect are replaced by their conjunction with the pointcut by which the aspect's per-clause is parameterized.

For each pointcut-advice pair in the aspect body, one Attachment is created, its Action being a MethodCallAction that refers to the method the compiler created for the advice. The Schedule Info trivially mirrors the keyword **before**, **after**, or **around**. Each pointcut is mapped to a set of Specializations. The mapping of individual pointcut designators to LIAM is best illustrated by a representative example: The **args** pointcut designator can be parameterized by an identifier corresponding to a pointcut parameter. This imposes a dynamic constraint on an

[8] See `http://www.alia4j.org/alia4j-languages/mappings`.

argument's type and exposes the argument's value to the advice. The restriction is mapped to an InstanceofPredicate with an associated ArgumentContext. For the value exposition, an ArgumentContext is associated with the Specialization.

When precedence is defined between aspects in terms of **declare precedence**, for each pair of Attachments from the referred aspects one Precedence Rule is created. Named pointcuts, abstract aspects and pointcuts, and inter-type *member* declarations [16] can also be realized with the ALIA4J approach, but we omit their discussion for the sake of brevity. Inter-type declarations that modify the type hierarchy (**declare parents**) or emit errors and warnings during compilation (**declare error**, **declare warning**) are naturally out of scope for ALIA4J.

**CaesarJ** While CaesarJ's pointcut-advice language features are the same as AspectJ's, a CaesarJ class can also be deployed and undeployed dynamically using **deploy**, **undeploy**, or a dedicated API. In this case, the program specifies an actual instance of the class which is to be deployed, i.e., an ObjectConstantContext parameterized with this object is used as the first context of all Specializations. Dynamic deployment also can add a scope, i.e., the class's pointcut-advice may be active only within in a single thread or while a specified object is executing. This scope is modeled as an Atomic Predicate and the Predicates of all Specializations are replaced by a conjunction with this Atomic Predicate.

**Compose*** In Compose*, filter modules are superimposed (deployed) on so-called *inner* objects and contain filters that react upon methods invoked either on (**inputfilters**) or by (**outputfilters**) the inner object. Data fields in a filter module can be defined, e.g., as **internals** that have a distinct value for each inner object.

For each of a module's filters, consisting of filter type, condition part, matching part, and substitution part, an Attachment is created. Hereby, the filter type and the substitution part are together mapped to an Action, the former determining the kind of Action and the latter its parameterization. Filter types like the **Exception** filter are predefined and are mapped to dedicated Action entities. Filter types provide a specification of their effects: Whether they are active in the calling or returning flow is captured by a Schedule Info entity; whether the message flow continues after the Action or whether subsequent filters are skipped is captured by Composition Rules. Conditions are implemented as methods in Compose* and represented by LIAM's MethodPredicate. Access to internal data fields is represented as PerTupleContext configured with a 1-ary tuple exposing the CalleeContext, when accessed from an input filter, or the CallerContext, when accessed from an output filter.

Filter modules have to be explicitly superimposed; the corresponding Attachments are not deployed by default. Superimposition acts on a set of classes on whose instances a filter module is to be superimposed. This is modeled by a conjunction of the affected Attachments' Predicates with an ExactTypePredicate Atomic Predicates (configured with either CalleeContext or CallerContext for input and output filters, respectively). Further constraints between filter modules

specified in Compose* can be represented using LIAM's Precedence Rules and Composition Rules.

**JPred** In JPred, methods may have a predicate in a **when** clause. A class can contain multiple methods with the same name and formal parameters but with different **when** clauses. When the method is invoked, the implementation with the most specific, satisfied predicate is executed; an implementation whose predicate implies the predicate of another implementation overrides it. Methods defined in a super-class are also overridden. The JPred compiler statically checks that for each method-call site exactly one implementation will be applicable.

For all these predicate methods, the compiler generates a plain Java method with a unique name. For each predicate method an Attachment is created with a MethodCallAction configured to execute this method. The Pattern of the Attachment selects invocations of the method according to the predicate-method name and the Predicate corresponds to the predicate specified by the **when** clause. As only a single predicate method must ever be executed, even if multiple predicates may be satisfied, the overriding relations are mapped to Composition Rules.

**ConSpec** Unlike the above languages, ConSpec [1] is not a general-purpose language but used only to express security policies. Regardless, it shares a number of characteristics with aspect-oriented languages: Its notion of events and guards is akin to AOP's pointcuts whereas its notion of updates is akin to advice, the key difference being a constrained set of possible actions; updates can only affect a limited set of state variables in limited ways. These state variables can moreover exist in several scopes, which allows them to be associated with particular objects (**OBJECT**) or persisted across program runs (**MULTISESSION**). In either case, LIAM can express scopes using an appropriate PerTupleContext; in the latter case, e.g., the lazily created instance is initialized with the persisted state.

**New, Domain-Specific Languages** The ALIA4J approach was used in the course "Advanced Programming Concepts" (2009/10) taught at the University of Twente to illustrate the execution semantics of advanced-dispatching languages and to perform practical assignments. During this course, groups of two or three students developed prototypes of domain-specific languages (DSLs), covering domains as diverse as (1) the declarative definition of debugging activities, (2) annotation-defined method-level transactions, (3) asynchronous Future-based inter-thread communication, (4) runtime model checking, (5) authentication and authorization, and (6) the automatic enforcement of the Decorator design pattern. All language prototypes except the sixth could be implemented by re-using the already existing LIAM entity implementations. This shows that our approach is well suited for the implementation of domain-specific languages.

**Summary and Lessons Learned** Table 1 shows the different concrete entities we implemented while mapping the languages AspectJ, CaesarJ, JPred, MultiJava, Compose*, and ConSpec to LIAM, as well as their usage in the different

| | AspectJ/ CaesarJ | Com- pose* | JPred/ MultiJava | Con- Spec | | AspectJ/ CaesarJ | Com- pose* | JPred/ MultiJava | Con- Spec |
|---|---|---|---|---|---|---|---|---|---|
| Context | | | | | Pattern | | | | |
| Argument | ✓ | ✓ | ✓ | ✓ | Method | ✓ | ✓ | ✓ | ✓ |
| Callee | ✓ | ✓ | ✓ | ✓ | Constructor | ✓ | | ✓ | ✓ |
| Caller | ✓ | ✓ | | | StaticInit. | ✓ | | | |
| Result | ✓ | ✓ | | ✓ | FieldRead | ✓ | | | |
| Arguments | ✓ | ✓ | | | FieldWrite | ✓ | | | |
| DebugInfo | ✓ | | | | AtomicPredicate | | | | |
| Signature | ✓ | ✓ | | | Instanceof | ✓ | ✓ | ✓ | |
| PerTuple | ✓ | ✓ | | ✓ | Method | ✓ | ✓ | ✓ | ✓ |
| PerCFlow | ✓ | | | | ExactType | ✓ | ✓ | ✓ | |
| PerCFlowBelow | ✓ | | | | CFlow | ✓ | | | |
| ObjectConstant | ✓ | | ✓ | | CFlowBelow | ✓ | | | |
| AspectJSignature | ✓* | | | | Bin.Relation | | ✓ | ✓ | ✓ |
| JoinPointKind | ✓* | | | | Action | | | | |
| SourceLocation | ✓* | | | | FieldRead | (✓) | (✓) | (✓) | (✓) |
| ThisJoinPoint | ✓* | | | | FieldWrite | (✓) | (✓) | (✓) | (✓) |
| Thread | ✓(CaesarJ) | | | | MethodCall | ✓ | ✓ | ✓ | ✓ |
| Constant | | | ✓ | ✓ | CFlowEnter | ✓ | | | |
| Field | | | ✓ | ✓ | CFlowExit | ✓ | | | |
| ArrayElement | | | ✓ | ✓ | NoOp | | ✓ | | ✓ |
| BinaryOperation | | | ✓ | ✓ | Throw | | ✓ | | ✓ |
| UnaryOperation | | | ✓ | ✓ | | | | | |
| MethodResult | | ✓ | ✓ | ✓ | | | | | |
| ReifiedMessage | | ✓* | | | | | | | |

**Table 1.** Usage of LIAM entities in different languages. ✓: non-trivial entity directly used in language mapping; ✓*: trivial context adapting interface of value; (✓): non-trivial entity used indirectly.

language mappings. CaesarJ shares the column with AspectJ, as the pointcut-advice part of the language largely overlap with AspectJ; JPred and MultiJava share a column because the former subsumes the latter.

### 4.2 Evaluation of FIAL

We have developed various FIAL-based back-ends (Steamloom$^{ALIA}$, SiRIn, and NOIRIn) using different execution strategies reaching from interpretation over bytecode generation to direct generation of machine code. Experiments have shown that native machine-code generation for LIAM entities of simple language concepts does not improve performance significantly. Thus, we will not discuss the implementation of Steamloom$^{ALIA}$ and its use of modularly implemented machine-code generation strategies here. Nevertheless, this support is useful for more complex VM-integrated optimizations, e.g., for **cflow** [6].

**SiRIn** SiRIn, the Site-based Reference Implementation, wraps every dispatch site into a special method and generates bytecode for these "reified" dispatch sites using the ASM bytecode engineering library.[9] Each wrapper method contains code derived from the dispatch function. 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 itself is a Java 6 agent; it does not require a native component and is thus fully portable.

**NOIRIn** NOIRIn, the Non-Optimizing Interpreter-based Reference Implementation, refrains from code generation and interprets the execution model produced by FIAL. Based on NOIRIn, implementing generic IDE support for debugging FIAL's execution models is straight-forward [28, 9]. Because NOIRIn does not generate bytecode for dispatch sites, it can only handle LIAM entities which implement a "compute" method. This is not a restriction because it can be expected that for each LIAM refinement a "compute" method is implemented at first, eventually supplanted by an optimizing bytecode generation. Like SiRIn, NOIRIn integrates with any standard Java 6 VM.

**Integration Testing** We provide an extensive suite of integration tests, which use the FIAL framework to define and deploy LIAM-based dispatch representations, execute an affected dispatch site, and verify the correct execution. The suite is independent of any concrete FIAL instantiation and, thus, also acts as compatibility test. It contains one JUnit test case per provided LIAM entity and several test cases for FIAL's services like dynamic deployment or ordering actions at shared join points. Each test case contains up to 512 tests using the tested entity or service in different ways and executing dispatch sites with different characteristics. Nearly all of the 4,045 tests are systematically generated to cover all relevant variations of dispatch sites: execution in a static or virtual context; dispatch of a method call, field read or write; etc.

## 5 Conclusions and Future Work

In this paper, we have presented the ALIA4J approach to implementing language extensions. Phrasing them in terms of advanced-dispatching enables us to implement numerous languages, ranging 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 of language sub-constructs is possible even across language families.

The re-use of implementation facilitated by ALIA4J allows programming-language researchers and designers of domain-specific languages to focus on their immediate task: developing source languages for solving certain problems. Already established language sub-constructs do not have to be implemented anew.

[9] See `http://asm.ow2.org/`.

ALIA4J's back-end-independent execution model and the possibility to modularly implement bytecode generation for language constructs make optimizations developed in back-ends immediately available to all languages implemented with our approach using the affected construct. We believe that this can improve the quality of language prototypes, but this is subject to future studies.

Language extensions developed using ALIA4J all build on the same language-independent meta-model: LIAM. This gives rise to the possibility of combining, e.g., AspectJ and JPred within a single program without unwanted interferences caused by low-level code transformations. But such a detailed study of the high-level interactions of different language implementations has yet to be done.

We also plan to re-implement several past research results uniformly within the ALIA4J approach. An optimized implementation of control-flow-based Atomic Predicates [6] in STEAMLOOM$^{\text{ALIA}}$, e.g., will benefit everyone using this platform-dependent back-end. As the LIAM-based intermediate representation is independent of a specific execution strategy, the same code is still executable on a less optimizing but platform-independent back-end. We also plan to map additional languages to our approach to further strengthen our claim of its generality.

Research is currently going on in developing new optimizations of language sub-constructs and making them available through the interface of LIAM. Furthermore, we are investigating extensions to LIAM and FIAL to make them more suitable to support tasks like debugging or profiling advanced-dispatching programs [9, 28]. Other research focuses on optimizing the generic service implementations in FIAL like the evaluation of Patterns [4], which will benefit all FIAL-based back-ends and thus all languages implemented in our approach.

## 6 Acknowledgements

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, Haihan Yin, and Martin Zandberg. We would also like to thank Eric Bodden and Jan Sinschek for their comments on earlier drafts of this paper. This work was supported by CASED (`www.cased.de`).

## References

1. I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of REM*, 2008.
2. I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. In *TAOSD*, volume 388 of *LNCS*, pages 135–173. 2006.
3. P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc : An extensible AspectJ compiler. In *TAOSD*, volume 3880 of *LNCS*, pages 293–334. 2006.

4. R. Bijker, C. Bockisch, and A. Sewe. Optimizing the evaluation of patterns in pointcuts. In *Proceedings of VMIL*, 2010.
5. C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages*. PhD thesis, Technische Universität Darmstadt, 2009.
6. C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient control flow quantification. In *Proceedings of OOPSLA*, 2006.
7. C. Bockisch, S. Malakuti, M. Aksit, and S. Katz. Making aspects natural: Events and composition. In *Proceedings of AOSD*, 2011.
8. C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of VMIL*, 2007.
9. C. Bockisch and A. Sewe. Generic IDE support for dispatch-based composition. In *Proceedings of Composition*, 2010.
10. C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of ECOOP*, 1992.
11. C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM TOPLAS*, 28(3), 2006.
12. A. de Roo, M. Hendriks, W. Havinga, P. Dürr, and L. Bergmans. Compose*: a language- and platform-independent aspect compiler for composition filters. In *Proceedings of WASDeTT*, 2008.
13. R. Dyer and H. Rajan. Supporting dynamic aspect-oriented features. *ACM TOSEM*, 20(2), 2010.
14. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of OOPSLA*, 2007.
15. M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP*, 1998.
16. P. Flach. *Implementierung von AspectJ-Intertypdeklarationen*. Bachelor's Thesis, Technische Universität Darmstadt, 2008. (In German).
17. W. Havinga, L. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of ECOOP*, 2008.
18. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.
19. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP*, 2001.
20. H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*, 2003.
21. T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM TOPLAS*, 31(2):1–54, 2009.
22. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC*, 2003.
23. W. Royce. Improving software economics-top 10 principles of achieving agility at scale. White paper, IBM Rational, May 2009.
24. B. G. Ryder, M. L. Soffa, and M. Burnett. The impact of software engineering research on modern progamming languages. *ACM TOSEM*, 14, 2005.
25. H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *Proceedings of OOPSLA*, 2008.
26. A. Sewe, C. Bockisch, and M. Mezini. Redundancy-free residual dispatch. In *Proceedings of FOAL*, 2008.
27. É. Tanter. An extensible kernel language for AOP. In *Proceedings of the Workshop on Open and Dynamic Aspect Languages*, 2006.
28. H. Yin and C. Bockisch. Developing a generic debugger for advanced-dispatching languages. In *Proceedings of WASDeTT*, 2010.