Generic IDE Support for Dispatch-Based Composition

Christoph Bockisch
Software Engineering group – University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands
c.m.bockisch@ewi.utwente.nl

Andreas Sewe
Software Technology group – Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
sewe@st.informatik.tu-darmstadt.de

ABSTRACT

Programming-language research produces a significant number of new programming styles to improve the composability of programs. This increases re-usability as well as other quality characteristics. But although they offer interesting composition concepts, new programming languages are rarely used because IDE support, which developers are used to, is missing. Examples of such IDE support are the visualization of call hierarchies or interactive debugging. While some languages, e.g., AspectJ, eventually reach a more mature level with elaborate IDE integration, not all language designers are able to invest this much effort towards IDE integration. Furthermore, the IDE integration of AspectJ also has its limitations; when debugging, the developer is confronted with synthetic code with no exact correspondence in the source code. As a result, the developer needs to understand the transformations performed by the compiler. Finally, some information invariably gets lost during weaving, e.g., the ability to map code evaluating pointcut designators to their definition in the source code.

In this paper, we propose to implement generic IDE tools for programming languages that provide advanced dispatching mechanisms. Such languages, including predicate dispatching and pointcut-advice languages, can be mapped to our execution model, called ALIA. The same execution model can then drive debugging functionality as well as static IDE services.

1. INTRODUCTION

In order to improve the modularity of source code, research strives to define new composition mechanisms, often in terms of new languages. Many such languages provide composition mechanisms by allowing to influence the dispatch of, e.g., method calls, like in multiple dispatching [9] or predicate dispatching [14]. But other composition styles can be mapped to a dispatching-based execution model as well, as we have shown [5] for pointcut-advice languages [15], Composition Filters [12], and a DSL for composing objects following the Decorator design pattern.

Usually, advanced dispatching mechanisms are provided as an extension of an existing programming language, the so-called base language, and the semantics of the advanced program features are realized by transforming them to the base language's imperative code. We have shown [5] that the dispatching mechanisms of all these languages share concepts from several broad categories: selection of call sites based on syntactic properties, access to the runtime state in which they are executed, evaluation of functions over the runtime state to select from alternative meanings, declaration of meaning in terms of actions on the runtime state, and description of relationships between applicable actions. Each language uses some extension of each core concept and the concrete concepts used in different languages often overlap.

Similarly, the requirements for IDE support of such languages overlap. Different kinds of support for the development in the investigated languages are recurring, but have to be implemented from scratch for each language. As a result, the IDE support for new languages is typically limited, as more effort is spent on the design of the language and the implementation of compilers than on the language's IDE integration. In the following, we discuss a few examples of IDE support from which all investigated languages can benefit.

Among the investigated languages, the aspect-oriented ones support implicit invocation. It thus is desirable to let the IDE visualize the places in the code at which other code may be (implicitly) the target of dispatch. To this end, the IDE support for the AspectJ language, the AspectJ Development Tools (AJDT), provides different ways of visualizing such relations. However, even for languages with only explicit invocation, similar IDE support is present. The Eclipse Java Development Tools (JDT) allow, e.g., to search for all call sites of a method, or to show the possible targets of a call site. It should be noted that, while calls must be explicit in Java, they can be virtual and multiple implementations may be applicable. The potential targets depend on the inheritance hierarchy, which may be too complex for the developer to grasp in its entirety. IDE support is therefore essential. The same observation holds for predicate-dispatching languages.

All investigated languages can be compiled to pure Java bytecode and can run on a standard JVM. Therefore, the default debugger of the IDE can be used to debug programs written in those languages. What is debugged, however, is the program *after* the transformation. Consequently, the developer is facing large amounts of infrastructural code that has been inserted by the compiler to realize the semantics and will often end up stepping through code for which no source code exists, which makes it even harder to understand. Another difficulty is that the Java debugger assumes that all code of a class was compiled from a single source file, but with new composition mechanisms this assumption may no longer hold: one class may be composed of multiple source files. Compilers merge all files into one; thus, the mapping from target code to the source code is lost and cannot be used by the debugger anymore.

We have provided an architecture for implementing advanced-dispatching languages in a way that they can share the implementation of overlapping concepts [5]; it is called the *Advanced-dispatching Language-Implementation Architecture* (ALIA) and consists of a lan-

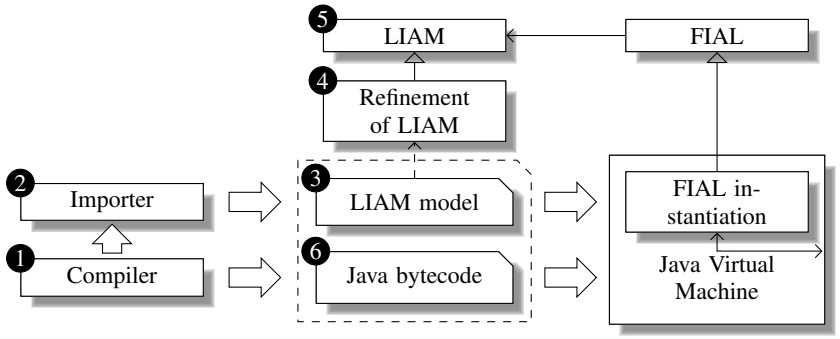

Figure 1: Overview of the application life cycle in ALIA4J-based language implementations.

guage-independent meta-model of advanced dispatching concepts and any number of execution environments that process models conforming to this single meta-model. For languages extending Java we have implemented this architecture, called ALIA for Java (ALIA4J), which furthermore provides a framework factoring out shared components of such execution environments.

In this position paper, we will discuss how ALIA's meta-model, more specifically its implementation in ALIA4J, and the framework for execution environments can be used to provide a generic infrastructure for IDE support of advanced-dispatching languages.

2. THE ALIA ARCHITECTURE FOR JAVA

In ALIA4J, the meta-model stipulated by ALIA is embodied in the *Language-Independent Advanced-dispatching Meta-model* (LIAM). LIAM hereby acts as the form of intermediate representation for advanced dispatching in programs. The actual intermediate representation, in turn, is a model conforming this meta-model (the so-called *LIAM model*). Code of the program not using advanced dispatching mechanisms is represented in its conventional Java bytecode form. The *Framework for Implementing Advanced-dispatching Languages* (FIAL) implements common components and work flows required to implement execution environments based on a JVM for executing LIAM models. A brief overview, of the approach can be found in [7][1].

Figure 1 shows an overview of the ALIA4J approach. Concretely, the flow of compiling and executing applications in this approach is shown. The compiler ❶ starts processing the source code; a dedicated importer component ❷ adapts the compiler's output to a model for the advanced dispatch declarations in the program ❸ based on the refined subclasses ❹ of the LIAM meta-entities ❺. Furthermore, the compiler produces an intermediate representation of those parts of the program that are expressible in the base language ❻ alone.

The nine meta-entities of LIAM capture the core concepts underlying the various dispatching mechanisms, but at a finer granularity than the concrete concepts found in high-level languages; one concrete concept often maps to a combination of LIAM's core concepts. Figure 2 shows the meta-entities in LIAM, which are implemented as abstract classes. Attachment, specialization, and predicate are an exception to this rule, i.e., they are concrete classes, as they provide logical groupings of entities of the meta-model and cannot be refined. The meta-entities are discussed in detail in [5, Chapter 3.2][2].

In short, an attachment corresponds to a unit of dispatch description. In terms of aspect-orientation (AO), this is a pointcut-advice

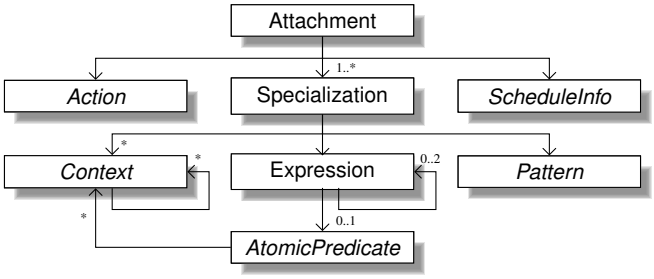

Figure 2: Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

pair, in terms of predicate dispatching, an attachment corresponds to a predicate method. Action specifies an action to which the dispatch may lead (e.g., an advice or the predicate-method body). Specialization defines static and dynamic properties affecting dispatch: patterns specify syntactic properties of call sites which are affected by the declared dispatch; predicate and atomic predicate entities model dynamic properties a dispatch depends on (dynamic pointcut designators in AO terminology). Context entities model access to values in the context of a dispatch, like the calling object or argument values. Finally, the schedule information models constraints between multiple actions applicable at the same generic-function call. This includes the order of their execution, as well as relations like mutual exclusion.

At runtime, FIAL derives a dispatch model for each dispatch site in the program from all attachments that have been defined. Thereby, FIAL solves the constraints specified as schedule information and derives a single dispatch function per call site from the predicates of all specializations. This function is represented as a binary decision diagram (BDD) [8], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with the actions to be executed. For each possible result of dispatch, the BDD has one leaf node. Figure 3 shows an example of such a dispatch model with the atomic predicates x_1 and x_2 and the actions y_1 and y_2. For a detailed explanation of this model, we refer the reader to [18].

The dispatch model is defined in such a way that an execution strategy can immediately be derived from it. The default execution strategy requires that each concrete entity implementation provides a Java method implementing its semantics. It is possible to override the default strategy and implement an optimization strategy on a per atomic predicate basis, in a modular way. These strategies are extensively discussed in [5].

The approach allows to implement new concrete concepts modularly by refining the abstract class of a meta-entity. We have already shown how to map the languages AspectJ, JAsCo, Compose*, Cae-

[1]Some details presented in [7] are outdated, but it may nevertheless act as an introduction to the basic concepts.

[2]There, some meta-entities are named differently, but the structure of the meta-entities is the same. Therefore, the interested reader will be able to map the discussion to the new names.

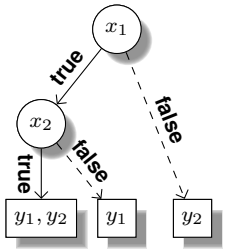

Figure 3: A dispatch function's evaluation strategy.

sarJ, and a simple domain-specific language to our meta-model [5]. By now, we have also developed mappings for the languages MultiJava [10], JPred [16], and ConSpec [4], which are, however, still unpublished work.

Important for the present paper is the fact that all concrete concepts participating in a dispatch are expressed in a declarative and fine-grained model. This model can thus be used to derive information relevant to the different services of an IDE. Furthermore, the model stays first-class during the execution of a program and can therefore easily support dynamic features of an IDE, e.g., debugging, profiling, or testing.

3. ALIA4J-BASED IDE SUPPORT

So far, we have implemented a limited IDE integration for the ALIA4J mapping of AspectJ language. However, we aim at making this support more general and support other languages, too. Moreover, we aim at filling the gaps in our IDE support.

3.1 Cross References for AspectJ

For the AspectJ language, we have implemented a nearly complete integration with our architecture. All necessary LIAM entities are implemented and we have developed an automatic importer component which allows to execute AspectJ programs on an FIAL-based execution environment while developing it in the standard AJDT. The benefit of this integration is that some FIAL-based execution environments perform sophisticated dynamic optimizations which make AspectJ programs execute faster than the product of the standard compiler [6].

Because we bypass the weaving phase of the AspectJ compiler in this approach, pointcuts are not evaluated at compile time anymore. Thus, the compiler also cannot determine the crosscutting structure of the aspects which would normally be used by the IDE to show, e.g., in the "Cross References" view, or which would be used to facilitate navigation between advised join point shadows and their advice, as depicted in Figure 4. To restore the accustomed functionality, we have developed an extension to the AJDT that provides the crosscutting structure when compiling AspectJ applications for execution on a FIAL-based execution environment, i.e., without compile-time weaving. This comprises an instantiation of the FIAL framework which is, however, not integrated in a Java Virtual Machine like a full-fledged FIAL-based execution environment. Instead of providing FIAL with dynamic information about generic-function calls, it provides static approximations of call sites. Our framework then evaluates all patterns in the LIAM models of the AspectJ project and builds the dispatch model for each call site. Afterwards, for dispatch models which are not trivial, i.e., where no advice is attached, the links are established in AJDT's abstract structure model.

To support this work, the LIAM meta-entities are extended to also store the location in the source code where they are defined. This is similar to the debug information present in Java bytecode. This debug information facilitates recovery of the file name and line number

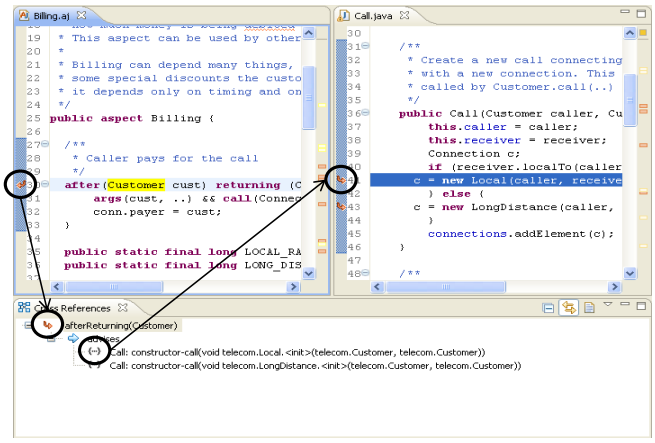

Figure 4: Linking of pointcut-advice and advised locations in AJDT.

whose compilation has lead to a bytecode instruction, respectively in the case of LIAM to a model entity. The builder uses this information to establish links between source locations, as is stipulated by the AJDT abstract structure model.

In the current version, input is hard-wired to the AspectJ compiler's output. But since our architecture already provides a plug-in mechanism to provide input in different formats, a straight-forward extension is to use this mechanism. Then, the same support can be provided for any language that can be mapped to our approach.

While the above AJDT extension shows the feasibility of building static tool support based on our ALIA approach, we do not aim to extend the AJDT in our future research work. Instead, we will re-implement similar support, including an AJDT-like structure model and related views, by directly extending the Java Development Tools (JDT). This is necessary because the AJDT and the structure model are hard-linked to the AspectJ compiler, a dependency that we would rather avoid. Furthermore, we have already outlined that there are commonalities between the cross references view and, e.g., the call hierarchy of methods explicitly called. Since both concepts, explicit and implicit calls, are unified in our architecture, we like to provide IDE support for both in the same way. The developer will benefit from such unified tools, because he will see all contributors to a call at the same time.

3.2 Debugging Support

The debugging support we envision will be based on the availability of our declarative dispatch model at runtime. For example, this makes it possible to visualize the dispatch model for a call at a breakpoint. The dispatch model is complete in the sense that it specifies on which runtime values the dispatch depends and which predicates are evaluated on these values. While the model, naturally, only specifies the role of values that are used (e.g., "the first argument value"), in a debugger, also the value can be shown. This is already done by modern debuggers, e.g., in the "Variables" view of the Eclipse debugger. In contrast to general-purpose debuggers, our debugger for advanced dispatch will only show values relevant for the dispatch, and associate them to their role names.

Another contribution that results from using the ALIA approach to enable debugging is that dispatch declarations defined in different languages can be combined. Since all dispatch declarations (pointcut-advice, multi-methods, etc.) are mapped to the same meta-model, i.e., to LIAM, the actions resulting from these declarations can be

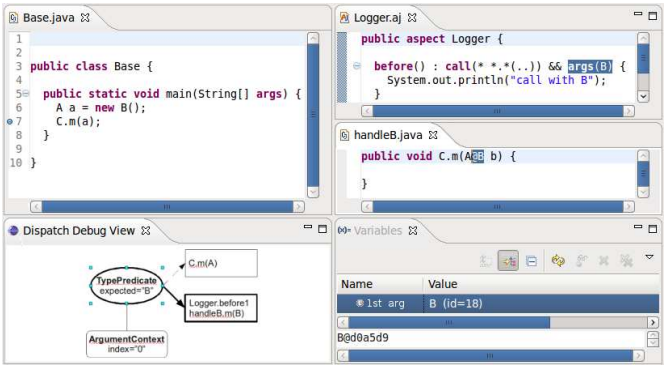

Figure 5: An idea of the GUI for generic debugging support for multiple advanced-dispatching languages.

executed alongside. That means that, e.g., calls to multi-methods can be advised.

Figure 5 illustrates the envisioned visualization in debugger. At the top of the figure, three editors are shown. The editor at the left-hand side shows Java code calling the method C.m(A) in line 7; at this line, a breakpoint is set. At the right-hand side, the top editor shows an AspectJ pointcut-advice and the bottom editor shows a multi-method defined in MultiJava. Both dispatch declarations define a dynamic constraint on the first argument: Only when this is of type B, the advice is to be executed, respectively, the multi-method applies. In this case, the multi-method overrides the Java method definition.

The bottom part of Figure 5 shows a possible visualization of the (simplified) dispatch model for the call at the breakpoint. The dispatch function is simple and only contains one atomic predicate, which tests the type of a context value, in the example that of the first argument. The bold elements show the path which the evaluation actually has followed. The bold solid arrow emerging from the predicate indicates that it has been satisfied, therefore, the actions in the bold box are to be executed as the dispatch's result, i.e., the actions Logger.before1 and handleB.m(B). If the predicate was satisfied, the action C.m(A) would be executed.

In a graphical debugger as proposed here, the user can select and introspect entities that participate in the dispatch at which the virtual machine is currently suspended. In the figure, the TypePredicate is selected. The selection in the editors showing the AspectJ and MultiJava code highlights the code which has led to this predicate in the dispatch function. The "Variables" view shows the runtime values on which the current selection depends, i.e., the first argument value. As can be seen, this is an instance of B and therefore, the predicate is satisfied.

As outlined above, the entities in the dispatch model can be linked to multiple source locations. The result of single atomic predicates in the dispatch function can be presented, which explains the result of the dispatch. Potentially, it will be advantageous not to completely evaluate the dispatch function and let the developer view the result afterward, but to allow a step-wise evaluation of the dispatch function. We will investigate both approaches.

3.3 Additional Ideas

The AJDT provides the developer with more detailed information than just "these advice apply to this join-point shadow". It already includes additional information by specifying whether the join-point shadow is always affected by an advice or only sometimes because there is a dynamic pointcut designator in the matching pointcut. Also, when showing the applicable advice, the AJDT orders them according to their precedence.

Nevertheless, we envision to increase the provided information in several ways. First, it is interesting to specify not only *that an advice is conditional* but also, *what the condition is*. Next, presenting a sequential list of advice is too limited because some languages support more complex relations between advice at a join-point shadow. AspectJ, e.g., already provides "around" advice which can be nested; thus, a tree would be more suitable to present this information. Other languages allow to define more complex relationships between advice at a shared join-point shadow. Examples are mutual exclusion or conditional execution in Compose*, or overriding in JPred and MultiJava.

The dispatch model, explained in some detail in the previous subsection, can also be made available before runtime. A visualization of the cross references can, thus, take all information in the dispatch model into account. This includes the exact specification of the condition under which an action is applicable at a call, dispatch declarations sharing the call site and relationships (order, execution constraints, etc.) among them.

Since the implementation of our architecture, i.e., FIAL and LIAM, is very modular, it is also easily possible to make part of their implementation interactive. A possible use is making pattern matching interactive in order to debug patterns. The AJDT shows the developer in which places pointcuts match, but in some cases, developers of pointcut-advice may wonder *why* a specific pointcut (respectively the pattern used in a pointcut) does or does not match. Since the definition of specializations (the equivalent to pointcuts in AspectJ) and the call sites are available first-class in FIAL, it is possible to perform the evaluation, e.g., for a specific call site, and show the developer the different steps in the evaluation. This is similar to the debugging support for dispatch functions, but can be performed before runtime.

4. RELATED WORK AND FUTURE WORK

Eaddy et al. [13] have identified several requirements for debugging aspect-oriented programs. They support source-level debugging by deferring the weaving to runtime, as in our approach. It is thus possible to view the definition of pointcut-advice that have lead to the execution of a specific statement. In contrast to our approach, the dispatch function is not represented in a structured declarative way, but only by the imperative code resulting from the pointcut-advice definitions. Thus, the dynamic program state that has lead to executing or not executing an advice is more difficult to determine for the developer. Furthermore, the original definition of an aspect (or dispatch declaration) is not presented. Therefore, constraints among advice sharing this join-point shadow are not easily visible, and, thus, cannot be easily debugged.

Pothier et al. [17] discuss a retrospective debugging approach for aspect-oriented programs. They record a complete execution trace that can be inspected after the execution. While this is not the kind of debugging that we will support, we will nevertheless take inspiration from their work in order to present AO-specific visualization of debugging information.

De Borger at al. [11] define an architecture for implementing debuggers for aspect-oriented languages. This architecture is based on a structurally reflective model of aspect definitions. For each aspect that is active during the program's execution, its structure can be queried by means of this model. It is possible to determine the executions of advice, which are *caused* by a pointcut, including executions in the past and in the future. Their model is meant to be an API used by a debugger front-end and offers some infrastructure required by debuggers, e.g., to enable aspect-specific breakpoints.

Our underlying model is more fine-grained and provides more information: constraints among aspects like precedence are not available through the reflective API. Nevertheless, we plan to investigate whether their work can be used as an interface for our approach. It

may be possible that our back-end, i.e., a FIAL-based execution environment, can be used as an implementation of their API. Should we follow this path, we aim to contribute additional functionality to the API which can be provided by means of our back-end. Similarly, the IDE integration of debugging that we envision, may be implementable with their API as back-end.

The IDE Meta-tooling Platform (IMP) [2] is an Eclipse project aiming at providing meta-implementations of typical IDE tools. Examples are a re-usable infrastructure for syntax highlighting, refactoring support, semantic or static analyses, execution and debugging. Their focus is on providing an infrastructure for the IDE integration and the graphical user interface, but not on providing an infrastructure for the runtime part of actual debugger implementations. Nevertheless, we will consider to integrate our work with this project. Potentially, the LIAM meta-model can act as re-usable abstract syntax tree for dispatch declarations in the IMP. We hope to be able to re-use components for the more static IDE support like the visualization of implicit and explicit calls.

There are other Eclipse projects into which we may integrate our envisioned work. The first option is the Dynamic Languages Toolkit (DLTK) [1] which is a collection of frameworks to minimize the effort of developing IDEs for dynamic languages. The second option is the Textual Modeling Framework (Xtext) [3] which is a framework for generating full-fledged Eclipse text editors from grammars for domain-specific languages, including an abstract source code model.

5. CONCLUSION

In the suggested research work, we aim at providing a generic implementation of IDE support, most importantly containing debugging support, for advanced-dispatching programming languages. We will build this support on the FIAL framework and the LIAM meta-model (part of the ALIA architecture for Java), which provide a first-class, declarative model of all dispatches in a program. We have mapped the aspect-oriented languages AspectJ, CaesarJ, Compose*, JAsCo, the predicate-dispatching languages JPred and MultiJava, and other languages to this model. All mapped languages will thus be able to directly benefit from the IDE support we aim to provide.

The IDE support will primarily consist of a navigable visualization of explicit as well as implicit calls (the former are used in predicate dispatching, the latter in pointcut-advice languages), and of debugging support. Both kinds of IDE integration will be driven by the declarative, first-class dispatch model available in ALIA. Since ALIA facilitates the execution of dispatch declarations written in different languages, all such dispatch declarations can be executed in one program run alongside; similarly, the debugging support we envision will be able to debug all such declarations at the same time. It will facilitate to jump to the source code defining the dispatch declaration, and it will to show all execution steps leading to a specific dispatching result. We will investigate similar support for reasoning about the evaluation of patterns used in pointcut-advice, respectively for the composition of actions applicable at the same call site.

Providing such IDE support that will work "out of the box" will increase the acceptance of new programming languages which offer sophisticated composition mechanisms by means of dispatch declarations. The envisioned IDE support will make the effects of applying advanced composition mechanisms to a program more obvious to developers, which will help them to learn such new mechanisms. ALIA's ability to execute programs written in different languages with different composition primitives and the resulting IDE support, will give developers the free choice of combining different languages and benefit from all their features. We would also like to note that many composition mechanisms which do not obviously map to a dispatching problem can still be handled by our architecture. For example, we successfully mapped AspectJ's inter-type member declarations to our approach; in fact, the example used in section Section 3.2 uses the open classes feature of MultiJava, which is equivalent to inter-type member declarations.

6. ACKNOWLEDGMENTS

We would like to thank all contributors to the ALIA4J project. In particular, our thanks go to Jannik Jochem, who developed the integration of ALIA4J with the AJDT.

7. REFERENCES

[1] The Dynamic Language Toolkit. `http://eclipse.org/dltk/`, 2010.

[2] The IDE Meta-tooling Platform. `http://eclipse.org/imp/`, 2010.

[3] The Textual Modeling Framework. `http://eclipse.org/xtext/`, 2010.

[4] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of REM*. Elsevier Science Publishers B. V., 2008.

[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*. ACM Press, 2006.

[7] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of VMIL*, New York, NY, USA, 2007. ACM.

[8] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35, 1986.

[9] C. Chambers. Object-oriented multi-methods in cecil. In *Proceedings of ECOOP*. Springer Verlag, 1992.

[10] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *Proceedings of OOPSLA*. ACM, 1999.

[11] W. De Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of AOSD*. ACM, 2009.

[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] M. Eaddy, A. V. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Software Composition*. Springer, 2007.

[14] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP*. Springer Verlag, 1998.

[15] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*. Springer Verlag, 2003.

[16] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 31(2), 2009.

[17] G. Pothier and E. Tanter. Extending omniscient debugging to support aspect-oriented programming. In *In Proceedings of SAC*. ACM, 2008.

[18] A. Sewe, C. Bockisch, and M. Mezini. Redundancy-free residual dispatch. In *Proceedings of FOAL*. ACM, 2008.