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

# Aspects for Stages: Cross Cutting Concerns for Metaprograms

Yannis Lilis[a] Anthony Savidis[ab]

a. Institute of Computer Science, FORTH

b. Department of Computer Science, University of Crete

Abstract In multi-stage languages the program code is finalized though a sequence of transformations defined in the program itself, a process known as staging, with stages also referred as metaprograms. Since stages are essentially programs, they may also require application of aspect-oriented methods to handle crosscutting concerns, something not considered or supported in existing aspect systems. We introduce aspect-oriented support for multi-stage languages by identifying three aspect types for the staging pipeline, namely *pre-*, *in-* and *post-* staging. We discuss their implementation in a language supporting compile-time metaprogramming, where aspects are realized as batches of AST transformation metaprograms, accompanied by an AOP-specific library. We also provide example scenarios where the proposed aspect types may be used in practice. Finally, we show how full-scale source-level aspect debugging is facilitated during the program compilation process.

Keywords Aspect-Oriented Programming; Multi-Stage Languages; Metaprogramming.

## 1 Introduction

Multi-stage languages (MSLs) [TS00, Tah04, She98] take the programming task of code generation and support it as a first-class language feature, realizing a sort of reification of the underlying language code generator. When code generation becomes a language construct, one may write generator code which produces other code that is integrated in the main program by substituting its generator. Thus the generator plays the role of a metaprogram, i.e. *program producing program*, while the process can be recursive with the generated code further producing extra code, causing staging to be nested. As a result, the practicing of multi-stage facilities is considered a metaprogramming method. The notion of staging is outlined under Figure 1.

As programming practices evolve and the concept of code generation becomes more and more mature, staging becomes a standard practice and the amount of staged code grows rapidly. For instance, comprehensive exception handling design patterns

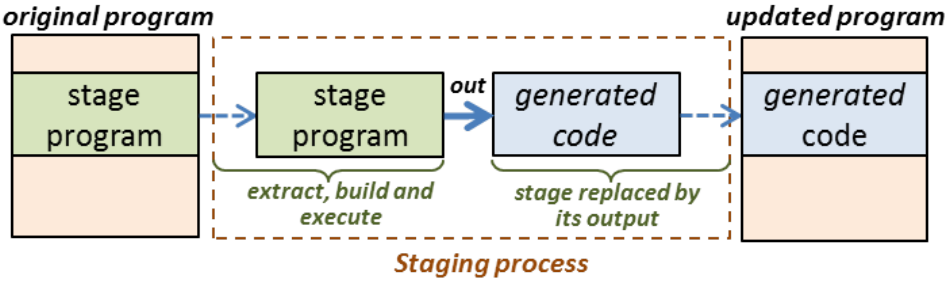


Figure 1 – General staging process in multi-stage languages

can be fully automated and composed through non-trivial stage programs [LS12a]. Despite this growth, most MSLs still treat staging as a special feature that is separated from the main language and is deployed with no resemblance to normal programs. Interestingly, exceptions to this rule are amongst the earlier MSLs like *Lisp* [McC62] and *Scheme* [Dyb09] whose macro systems offer the full power of the main language. These languages are actually aligned to our argument that stage metaprograms deserve the full range of programming techniques available to normal programs. *We argue that for stage metaprograms we should allow practicing of software engineering methods as with normal programs, including handling of crosscutting concerns through aspects.*

The latter relates to Aspect-Oriented Programming (AOP) [KLM+97], a methodology for modeling crosscutting concerns into modular units called *aspects*. Aspects contain information about the additional behavior, called *advice*, that will be added to the base program by the aspect as well as the program locations, called *join points*, where this additional behavior is to be inserted based on some matching criteria, called *pointcuts*. Aspects are typically expressed in separate languages and an *aspect weaver* combines the base program with the aspect program to form the final program.

Our work is motivated by the lack of support for AOP in thecontext of a MSL, and in particular not only for the normal programs but most importantly for *all stages* they might contain. Stage programs, besides their special mission being primarily generative to produce code, are essentially no different to normal programs. Thus, they deserve, and require, all typical programming techniques of normal programs, including aspects. For instance, *within stage code, one may deploy logging aspects to support tracing of method invocations, or apply exception handling aspects at appropriate call sites.* Apparently, there is no particular reason to forbid the application of such aspects in stage code. In fact, there are also various scenarios related to the generative role of stages. For example, stages handle code in the form of Abstract Syntax Trees (ASTs). In this context, we could define aspects for AST manipulation, such as decorating with extra code, validating according to criteria, or introducing custom iteration policies.

Without aspect support we simply limit the potential for developing stages using state of the art programming practices. For instance, consider *AspectJ* [KHH+01], a popular language for AOP, and *Mint* [WRI+10], a *Java* extension offering staging facilities. Staged code within a Mint program is actually Java code. However, it is not possible to use AspectJ to apply AOP on the staged code as it is never available in a form that can be manipulated by the aspect weaver[1]. In fact, the reason is more

[1]Here we only consider compile-time weaving or post-compile weaving. The alternative of load-time weaving that partly addresses this issue is discussed later on.

fundamental: *no interplay between the aspect weaver and a staging system has ever been considered or proposed.*

In current implementations for AOP the *language compiler* is ignorant of the *aspect weaver* and the transformation it performs on the original program. Overall, the aspect weaver is *never* in the compilation or execution loop. However, in MSLs, stages are composed and evaluated during either compilation or execution disabling any possibility for the aspect weaver to intervene. Additionally, the source or binary of a stage is transient during compilation or execution and cannot be available to the aspect weaver unless it becomes part of the loop. To resolve this, the respective source or binary files for stages must be created and supplied to the aspect weaver during the staging process. For the previous AspectJ and Mint example, the latter would involve explicitly writing aspects for staged code and weaving them into the binary along with the staged code so that the stage evaluation contains the advised functionality.

In this paper we propose the adoption of AOP in a MSL, introduce a methodology for aspect weaving in the entire staging pipeline and discuss an implementation[2] on an existing MSL. In this context, we do not introduce a separate aspect language for AOP, but we implement aspects as batches of AST transformation programs written in the same language. This approach fits well with typical multi-stage metaprogramming practices since programmers are already familiar with using and manipulating ASTs. Also, it allows exploiting features like reviewing, inspecting or debugging AST transformations that may already be offered by the MSL IDE. Overall, our main contributions are:

- Methodology for aspect-orientation in the entire staging pipeline.
- Deployment of aspects as AST transformations expressed in the same language.
- Implementation of source-level weaving in a language with compile-time staging.
- Integrated tool-chain to support debugging of aspect transformation programs.

The rest of the paper is organized as follows. Section 2 provides some background information related to ASTs, quasi-quotation and MSLs. Section 3 elaborates on the aspect weaving options for the various staging approaches, explains the rationale for introducing aspects into the staging pipeline and discusses the aspect categories applied in each step. Section 4 introduces the notion of treating aspects as AST transformation programs written in the same language and discusses how they can be integrated in the workspace management and build process within an integrated development environment. Section 5 presents various case studies deploying aspects in stages. Section 6 focuses on tool support to facilitate full-scale source-level debugging of aspect programs. Section 7 provides an analysis of related work while section 8 discusses additional issues about our approach and its deployment in a different context. Finally, section 9 draws key conclusions.

# 2 Background

## 2.1 ASTs and quasi-quotation

Metaprogramming involves generating, combining and transforming source code, so it is essential to provide a convenient way for expressing and manipulating source code

[2]For access to our system check the availability information at the end of the paper.

fragments. Expressing source code directly as text is impractical for code traversal and manipulation, while intermediate or even target code representations are too low-level to be deployed. Currently, the standard method for representing and manipulating code is based on ASTs, a notion originating from the *s-expressions* [McC62] introduced by Lisp. Although ASTs provide an effective method for manipulating source code fragments, manually creating them usually requires a large amount of expressions or statements, making it hard to identify the actually represented source code [WC93]. Thus, ways to directly convert source text to ASTs and easily compose ASTs into more comprehensive source fragments were required. Both requirements have been addressed by existing languages through a feature known as *quasi-quotation* [Baw99]. Normal quotation skips any evaluation, thus interpreting the original text as code. Quasi-quotation works on top of that, but instead of specifying the exact code structure, it essentially providing a source code template that can be filled with other code. To better illustrate this notion consider the following Lisp macro that generates the multiplication of the argument X by itself.

```
(defmacro square (X)
  '(* ,X ,X))
 (square 5) ; 25
```

Definitions after the *backquote* operator ' are not directly evaluated but are interpreted as a code fragment (i.e. an AST). The reverse of *backquote* is the *unquote* operator , that escapes the syntactic form and inserts its argument directly in the expression being created. This way, the invocation `(square 5)` creates the expression `(* 5 5)` that yields 25.

## 2.2 Multi-Stage Languages

MSLs extend the multi-level language [GJ95] notion of dividing a program into levels of evaluation by making them accessible to the programmer through special syntax called *staging annotations* [TS00]. Such annotations are introduced to explicitly specify the evaluation order of the various computations of the program. In this sense, a staged program is a conventional program that has been extended with the appropriate staging annotations. To illustrate the behavior of staging annotations consider the example written in *MetaML* [She98].

```
val code = <5>;
val square = <~code * ~code>;
val result = run square; (* 25 *)
```

*Brackets* `<_>` are used to create delayed computations thus constructing code fragments (i.e. ASTs). Then *escape* `~_` allows combining smaller delayed computations to construct larger ones by splicing its argument in the context of the surrounding *brackets* (i.e. performs AST combination). In this sense, the second assignment of the above code creates the delayed computation `<5 * 5>`. Finally, `run` evaluates the code specified by the delayed computation in the current stage (i.e. performs code generation based on the given AST), which in our example evaluates to 25. Notice that `run` essentially operate like a typed `eval` function [TS00] that receives an AST value instead of arbitrary source text.

Early research on MSLs like MetaML and *MetaOCaml* [CLT+01] targeted code generation during program execution (runtime staging or runtime metaprogramming - *RTMP*) and was focused in the domain of functional languages. Later research

also covered the application of program staging methods during program compilation (compile-time staging or compile-time metaprogramming - *CTMP*) as well as their adoption in the context of imperative languages (both statically or dynamically typed), thus offering various staging incarnations. For example, *Template Haskell* [SJ02] is a statically typed functional language that supports CTMP, *Converge* [Tra08] and *Metalua* [Fle07] are both dynamically typed imperative languages supporting CTMP, while *Metaphor* [NR04] and *Mint* [WRI+10] are both statically typed imperative languages that support RTMP.

Our work has been carried out in the dynamic object-oriented language *Delta* [Sav05, Sav10] and its *Sparrow* IDE [SBG07], extending them to supports aspect transformations in the entire staging pipeline. Before going into details specific for aspects, let's first brief the metaprogramming elements of our language as they are adopted throughout our discussion.

- Quasi-quotes (written `<<...>>`) are analogous to Lisp *backquote* or MetaML *brackets* and can be inserted around language elements to convert them to AST. Quasi-quotes can be nested at any depth (AST representing other ASTs) to allow forms for multiple levels of staging. For instance, `<< << x >> >>` represents the AST of `<< x >>`.

- Escape (written `~(expr)`) operates like Lisp *unquote* or MetaML *escape* and is used inside quasi-quotes to prevent converting *expr* to its AST form and evaluate it normally. It allows combining existing AST values in the AST being constructed by the quasi-quotes. Due to their special meaning within quasi-quotes, to create an AST value containing an escape we use a special form, called *delayed escape*, denoted as `<<~~expr>>`. A delayed escape essentially creates an AST with a placeholder for inserting code. As we discuss later, this can be used in the context of AOP to refer to the return value of a function or method that uses *after* advice or to emulate the *proceed* functionality of *around* advice.

- Inline (written `!(expr)`) operates like a Lisp macro invocation or MetaML *run* annotation by evaluating *expr* at translation time and inserting its value (that must be an AST) directly into the main AST, substituting itself and thus transforming the original source. Inline tags within quasi-quotes are allowed, but as all other quasi-quoted expressions they are just AST values and are not directly evaluated. For instance, `<<!(f())>>` represents the AST of inlining the result of function `f` and involves no staged evaluation. If however this is inlined in the source, generating the code `!(f())`, it will require evaluation, thus causing further staging.

- Execute (written `&stmt`) can be used to execute a statement at translation time. In Delta, function definitions are syntactically statements, so we can also write `&function f(){...}` to denote that `f` will be available only during compilation. Quasi-quoted execute tags are supported, but are again treated as ASTs.

## 3 Aspects in Staging

There are two approaches for weaving aspect code along with normal program code: *source-level weaving* and *binary-level weaving*. Source-level weaving involves applying the aspect on the original source to get the transformed version of the source that is

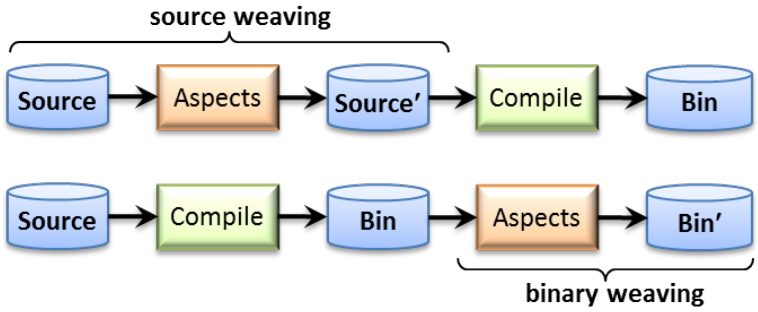


Figure 2 – The two alternative contexts for aspect weaving

then compiled to binary (Figure 2, top). On the other hand, in binary-level weaving the source is normally compiled to binary and then the aspects are applied to generate an updated binary version (Figure 2, bottom). However, in the context of existing MSL implementations, none of these approaches is sufficiently supported to facilitate AOP at a full scale. To explain why, we first consider the potential options for applying aspect weaving (either source- or binary-level) being *before*, *during* and *after* the staging process. Then we study the way such options can be supported under both CTMP and RTMP, the latter either for compiled implementations - $RTMP_C$, or interpreted ones - $RTMP_I$ (clearly, the distinction refers to the implementation method, not the language itself). As we discuss latter, the weaving options, and the way they can be applied, strongly depend on the MSL implementation approach. Although our system supports CTMP and all source-level weaving options, we discuss the rest two options to outline the important differences on weaving implementation, and because they concern a wide range of languages. In particular, RTMP concerns mainstream languages like *Java* and *C#*. Although not MSLs by default, they have MSL extensions, and also provide powerful reflection mechanisms that enable some degree of runtime metaprogramming [LS13]. Overall, the MSL implementation approach maps to different options of source or binary weaving, which display varying usability, expressiveness and efficiency properties when it comes to programming and applying aspects.

## 3.1 Weaving Options

In CTMP (Figure 3), the initial source is parsed into an AST form that is used to extract any stage metaprograms. For each stage, we extract its AST, optionally unparse it to get the stage source and then compile it to get the stage binary. The stage program is then run to transform the initial source by updating the program AST. This process continues until we have no more staging. At that point we can unparse the AST to get the final source and then compile it to get the final program binary or just translate the AST directly into binary. As already mentioned, examples of languages supporting this staging model are Converge, Metalua and Delta (support of a model is not necessarily exclusive; in fact all above languages also support RTMP).

In $RTMP_C$ (Figure 4), the main source is directly compiled into binary that is then loaded by the execution system. The runtime version of the main program contains stages that when invoked will generate code either as dynamic source code or directly in binary format. The dynamic source code has to be further compiled in binary code and will eventually update the instructions of the currently executing program.

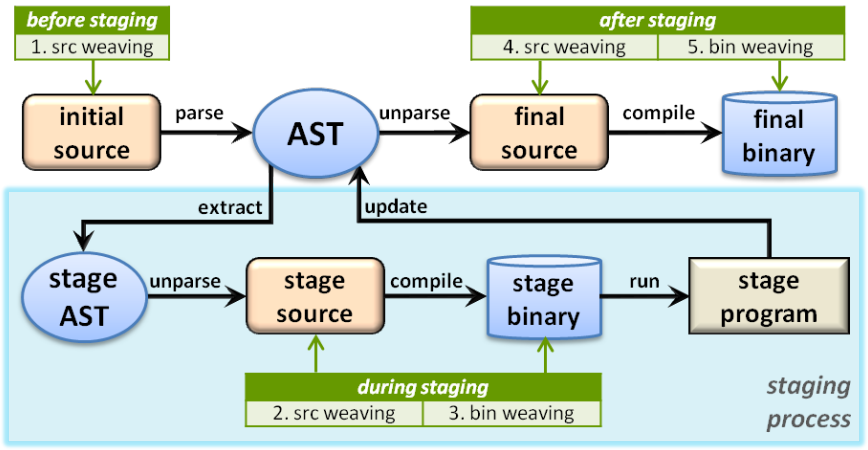


Figure 3 – Compile-time staging and aspect weaving options

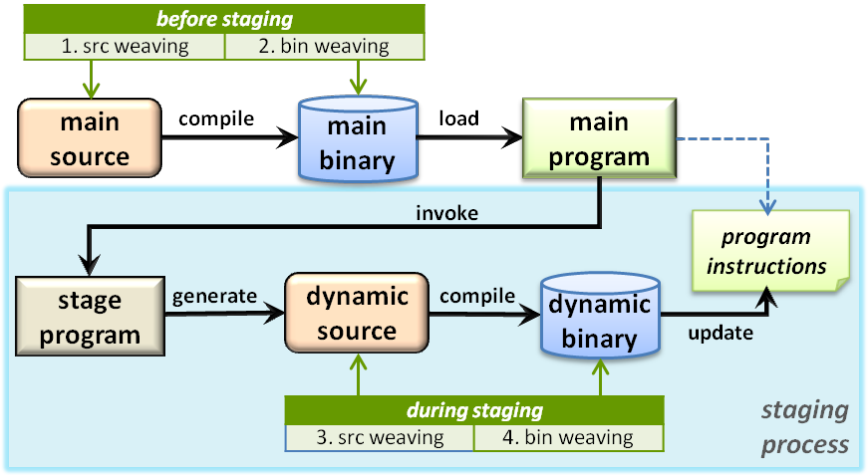


Figure 4 – Runtime staging (compiled language case) and aspect weaving options

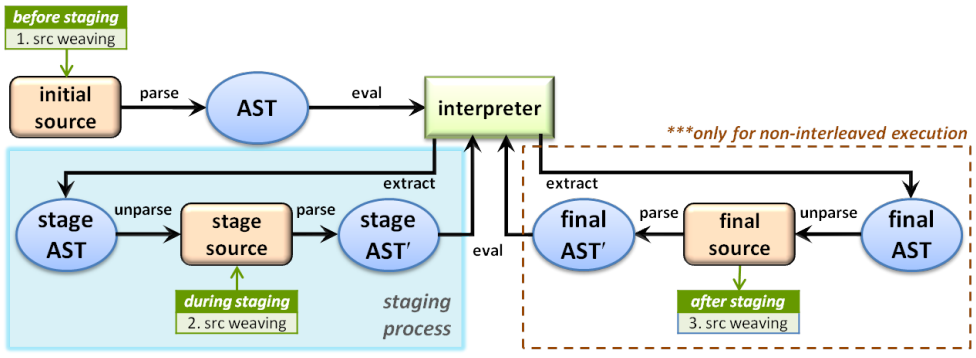


Figure 5 – Runtime staging (interpreted language case) and aspect weaving options

Such compilation may produce directly native code, or byte code for languages with virtual machine runtimes. In the latter case, once byte code is loaded and executed, typical JIT compilation may be applied, a process with no particular interference or relationship to the weaving or staging process. The same process is repeated for all meta-instructions encountered in the program execution. Languages supporting this staging model include MetaOCaml, Metaphor and Mint.

In $RTMP_I$ (Figure 5), the initial source is parsed into an AST form and then evaluated by the interpreter. Any stage metaprogram is also extracted in AST form and then evaluated by a recursive invocation of the interpreter. In case we want to have the stage source available, after extracting the stage AST we can unparse it to get the respective stage source, optionally transform it, and then parse it to AST for it to be evaluated. The extra steps taken to allow manipulating the stage source can be used as an entry point for AST transformations on the stage metaprogram. Languages supporting this staging model include MetaML, Lisp and Scheme.

Apart from the processing diagrams, Figures 3-5 also highlight the potential options for applying aspect weaving in a MSL. These options are not mutually exclusive and can be combined to achieve aspect orientation in multiple steps of the compilation or execution process. For instance, in CTMP we could apply source-weaving on the initial source (Figure 3: 1), apply source- or binary-weaving on the stage source or binary of the stage metaprogram respectively (Figure 3: 2-3), or finally we could apply source- or binary-weaving on the final version of the code, as transformed after evaluating all stages (Figure 3: 4-5). Similarly, in $RTMP_C$ we could apply source- or binary-weaving on the main code (Figure 4: 1-2) and then at runtime apply source- or binary-weaving on the code generated by the metaprogram (Figure 4: 3-4).

Finally, in $RTMP_I$, and following the common interpreted evaluation order for stages, we could apply source-level weaving on the initial source (Figure 5: 1) or the stage source that has to be unparsed for this purpose (Figure 5: 2). To offer the weaving option *after* the evaluation of stages (Figure 5: 3) a small modification on the way stages are actually interpreted is required. In particular, stages are commonly evaluated as part of the main program execution and as soon as they are met within program definitions. Thus, staging evaluation interleaves main program evaluation. In general, once staging completes, part of the main program is already executed, rendering meaningless to apply aspects on a program that is already partially evaluated. The reason is simple to explain. Consider we allow such weaving to take place, and imagine a function that is affected by weaving and which has already been invoked many times. Then, the semantics of such a function can vary during execution, with the version before weaving being different to the one another after weaving. Now, this sort of inconsistency appears only due to interleaving of stage evaluation with the main program. To adopt an interpreted evaluation that disables interleaving would be trivial in $RTMP_I$ implementations and with no discount on stage expressiveness. In particular, it suffices to apply stage evaluation first, and then, once no staging remains, proceed evaluating the main program. In fact, this type of ordering is similar to CTMP implementations, while setting *after* staging weaving a well-defined option.

Notice that in all previous cases, the initial source file contains both normal program code together with stage code, while aspect code is considered to be in separate files. Thus, a weaving process may in fact apply aspects to any of them. We further elaborate on what functionality can be addressed by the potential aspect applications in the following section.

The difference between applying AOP in a normal language and a meta-language

| Weaving Context | Weaving Subject | Compile-time Staging | Runtime Staging | |
|---|---|---|---|---|
| | | | Compiled language | Interpreted language |
| Source Code | Initial source | ✓ | ✓ | ✓ |
| | Stage sources | ✓ | ✓ | ✓ |
| | Final source | ✓ | N/A | ✓ |
| Binary Code | Initial binary | N/A | ✓ | N/A |
| | Stage binaries | ✓ | ✓ | N/A |
| | Final binary | ✓ | N/A | N/A |

Table 1 – Ability to implement aspects under different categories of MSLs and for the different possible weaving contexts and subjects

is that in the former case, there is a single source or binary for transformation, while in the latter case there are multiple sources or binaries for transformation, involved in different parts of the process. Table 1 summarizes the options for applying AOP with different combinations of source and binary aspect weaving for each staging approach.

Trying to apply the current AOP practices without interfering with the staging pipeline, means essentially operating as a source code pre-processor or binary code post-processor, thus limiting the potentials for aspect weaving. For CTMP we are limited to weaving options 1, 4 and 5, for $RTMP_C$ we are limited to weaving options 1 and 2, while for $RTMP_I$ we are limited to the single option 1. These however cannot fully express aspect transformations in the staging pipeline. For $RTMP_C$, this should be clear, as the dynamic code is generated at runtime with no way to be updated. For $RTMP_I$, it would be possible for the normal or staged code present in the initial source but there is no way to handle any code introduced by staging. For CTMP, the only supported scenario relates to a two-stage language, where we have only one stage of metaprogramming and the entire meta-code is available within the original source. In this case, it is possible to apply source-level weaving to transform the existing meta-code, while also applying binary-level weaving right after compilation to transform the code generated by the metaprogram. An example for this scenario would be *C++*, where a pre-compilation source-level weaving could transform the template code (i.e. the stage-code) and a post-compilation binary-level weaving transform the template instantiations (i.e. the generated code). The previous method cannot be applied for languages with more than two stages (i.e. more than one nested metaprograms). The reason is that the initial source-level weaving can only transform the meta-code that already exists in the original source, but not the meta-code that is introduced as a result of a previous stage. Other than that, any binary-level weaving would operate on the final program source after all stage metaprograms have been executed, and of course cannot transform their functionality.

Another possible weaving approach, still operating on binary level, would be to insert the extra functionality upon loading of the binary. This can be achieved by extending the loader with hooks that will perform the weaving, being the way load-time weaving is actually supported in AspectJ. Disregarding any performance penalties about the computations taking place at loading time, binary loading occurs for both normal and stage programs, so this approach could potentially be used to weave functionality in both of them without interfering with the staging pipeline.

However, this method does not allow differentiating between normal and stage

programs, meaning they cannot be supported with different aspects. The only way enabling different aspects, following our proposition, is for the MSL to uniquely name and separate the produced stage classes for all stage code fragments. The later would allow load-time weaving approaches by selectively intervening only on stage classes, once adopting the class name patterns of the MSL compiler. However, the latter requires two important changes. Firstly, we should guarantee that the MSL generates separate classes for stage code snippets, something not currently supported by known runtime MSLs for Java. Secondly, the naming patterns for stage classes should become a documented feature of MSLs so that load-time weavers can exploit them. Essentially, these two extensions serve no other purpose than allow bringing a load-time weaver into the staging loop. The later repeats our earlier argument that *no stage-level weaving is possible without the MSL actually setting the ground.*

Additionally, load-time weaving is applicable only for languages compiled to bytecode, like Java or C#, when run directly by respective virtual machines. However, it is not appropriate when Ahead-Of-Time compilation (AOT) is applied on such languages. Clearly, it is not applicable for languages that directly generate native code, like C or C++. Overall, we consider load-time weaving to be insufficient for full-scale aspect deployment within a MSL and do not further include it in our discussion, although it could achieve functionality similar to some of the case studies discussed later on.

In conclusion, in order to effectively support aspects for stages in a MSL, aspect weaving should be necessarily introduced as part of the staging process.

## 3.2 Aspect Categories

In a MSL, the original program $p_0$ also contains the stage metaprograms $s_1, ..., s_n$. With the execution of these stages, the original program $p_0$ is transformed sequentially to $p_1, ..., p_n$, the last being the final program version. In AOP, we typically have the original program $p$ that is advised by an aspect program $a$. Introducing AOP in a MSL requires considering the various interaction points: (i) program $p_0$ is advised by aspect program $a$; (ii) stage metaprograms $s_1, ..., s_n$ are advised by aspect program $a$; and (iii) intermediate program transformations $p_1, ..., p_n$ are advised by aspect program $a$.

Considering the first interaction point, the program $p_0$ contains both normal program code and staged code, meaning that the aspect $a$ could advise any of them. However, none of them have their final form yet; normal code may be transformed by stage code, while code of a particular stage may be transformed by higher stage code. This means that applying aspect $a$ to advise normal program code or stage code may cause inconsistencies and thus should be avoided. For example, consider a scenario where we advise the normal code to insert logging functionality for all functions it contains. With this taking place before the staging process, any functions generated due to staging will not contain the logging functionality, eventually resulting into final code where only some of the functions are actually advised. Nevertheless, applying an aspect on the original program can be useful. It can introduce additional code for a specific stage or even introduce extra stages. Such an aspect can be seen as a higher-order metaprogramming facility that allows the transformation logic to be entirely decoupled from the main source code. For example, this allows turning normal code to stage code to perform some computations during compilation and improve performance (sort of partial evaluation) or introduce stage code that performs static analysis on specific parts of the original source. Such aspects are always executed before the staging process, so we call them ***pre-staging aspects***.

In the second interaction point, we have each stage metaprogram $s_i$ being advised by

the aspect program $a$. Each stage contains code from both the original program along with code generated by stages directly embedded in it (higher-order). Thus, applying the aspect right before its evaluation, guarantees that the stage has its final form and that the advice functionality is consistent. The reason for using such aspects relates to crosscutting functionality typically found in stage code. With stages involving code generation, the manipulation of ASTs is very common, typically involving scenarios of structural validation, decoration with extra functionality or attributes, and custom iterators. Apart from their special purpose as code generators, stages are also programs that may involve crosscutting functionality typically found in normal programs, like synchronization, logging and monitoring. For instance, a common scenario may involve adding logging calls to trace meta-function invocations. For the weaver to deploy such aspects, it needs to have access to the source or binary code of each stage. This means that the compiler (in CTMP) or the runtime system (in RTMP) should not treat stages as private transient programs, but should somehow supply produced source or binary files to the aspect weaver to operate on. Additionally, interplay between the weaver and the compiler or runtime is required following the actual weaving process. More specifically, in source-level weaving, the compiler (or interpreter) generates the stage source, gives it to the weaver and gets back the advised version that it then compiles (or interprets). In binary-level weaving (only in compiled-languages), the compiler first compiles the stage source to binary, gives it to the weaver and gets back the advised binary version. We call such aspects ***in-staging aspects***.

For the third interaction point, we notice that the intermediate program transformations $p_1, ..., p_{n-1}$ are in fact intermediate forms. This means that any aspect application in them occurs on an incomplete program and may thus cause inconsistencies. The case of applying an aspect on $p_n$ in particular requires that all stage evaluations have been performed and the final program version has no more staging.

It should be noted that in RTMP, either interpreted or compiled, it is generally undecidable to judge if no further staging process can take place after a certain runtime point. The reason is that use of reflection mechanisms, dynamic loading or *eval* can generate implicit staged code, not visible in the currently executing program instructions. Moreover, in either RTMP or CMTP, aspects applied *after* staging could also introduce further staging. Consequently, there is no way to impose just a single staging process. As a result, we define as *final* a program containing no more staged code. Clearly, if implicit staging is introduced by the evaluation of the *final* program itself, or by aspects applied *after* staging, then further aspect weaving following the proposed approach reapplies. With such a scenario, additional staging rounds occur, leading to another final program at the end. In this sense, the term *final* just denotes the program resulting from a staging process, not by all staging processes. Overall, aspects on $p_n$ play the same role as aspects on normal programs: there is a program that needs to be advised involving no staging. They are applicable to both CMTP and $RTMP_I$, for the latter assuming a non-interleaved execution. They do not apply to $RTMP_C$, since, when the runtime staging process completes, part of the program has already been executed. Such aspects are always applied after the staging process, so we call them ***post-staging aspects***.

Table 2 gives an overview of each discussed aspect category, highlighting its purpose and deployment options for each staging approach.

For a complete combination of stages and aspects one may introduce multi-stage programming in the context of an aspect program. In fact, MSLs fully support nested stages, being metaprograms that generate the code of enclosing metaprograms.

| | Purpose | Compile-time Staging | Runtime Staging | |
|---|---|---|---|---|
| | | | Compiled language | Interpreted language |
| **Pre Staging** | Introduce or update staging on the original program | Before source compilation (*source weaving*) | Before main compilation (*source weaving*) or after main compilation (*binary weaving*) | Before main interpretation (*source weaving*) |
| **In Staging** | Update stages | Before stage compilation (*source weaving*) or after stage compilation (*binary weaving*) | Before dynamic source compilation (*source weaving*) or after dynamic source compilation (*binary weaving*) | Before stage interpretation (*source weaving*) |
| **Post Staging** | Update the final program | Before final compilation (*source weaving*) or just after compilation completes (*binary weaving*) | N/A (main program is already executing) | After non-interleaved interpretation of all stages (*source weaving*) |

Table 2 – Overview of stage aspect categories and their application context for different MSL implementation approaches

Similarly, one could consider chained aspects, being aspects applying cross-cutting concerns on the logic of other aspects. Thus, their combination is theoretically unlimited. Regarding the blending of stages and aspects, an aspect program $a_0$ may itself contain stage metaprograms $s_1, ..., s_n$ that transform it sequentially into $a_1, ..., a_n$, the last being the final version of the aspect program. Such a combination is meaningful, enabling aspect properties like pointcuts and advice to be generated through metaprogramming. However, it requires the aspect language to be extended with multi-stage constructs. If we consider an aspect to be applied to a client program through a binary executable form, any staging during aspect compilation is transparent to all of its clients, so its deployment remains the same despite staging. Effectively, the two sides of the combination between MSLs and AOP are orthogonal and can be adopted independently of each other. In this paper we primarily focus in the first direction, i.e. introducing AOP in a MSL. However, as discussed in the following section, we provide aspects as transformation programs written directly in our MSL, meaning our aspects can be fully staged.

## 4 Aspects without Dedicated Languages

A spin-off outcome of our work is an alternative way to apply aspect transformations. Essentially, we treat aspects as AST transformation programs written in the same language and deploying an aspect library working on ASTs. We elaborate on this notion and discuss how such transformation programs can be integrated in the workspace management and build process of the integrated development environment. We do

not argue that this is the ultimate approach towards supporting AOP; we present it as a viable alternative and discuss its advantages when deployed in an existing MSL.

## 4.1 Aspects as Transformation Batches

To test aspects for stages we started thinking of crafting a prototype aspect engine for our staged language. In this context, we observed that the language offers quasi-quotes, escaping, and a comprehensive AST library, all of which are not staged but can be used as part of a normal program. Now, the latter are essentially everything one needs to algorithmically perform source code transformations. Practically, aspects are a restricted form of algorithmic cross-cutting transformations, currently offered with distinct languages with automations in expressing pointcuts and advice. Regarding pointcuts, one might directly offer a library set to search AST nodes against criteria defined as predicate functions. A similar library set can also be offered for defining and applying advice.

The prototype implementation of our approach offers only static aspect transformations, being analogous to the static model offered by AspectJ. However, treating aspects as transformation batches is not limited to static weaving as such. Batches may be implemented in a runtime preprocessing process to perform binary or load-time weaving, thus operating in a way similar to the dynamic aspect weaving model. We focused on a compile-time static model only for practical reasons: (i) it is more efficient, as it introduces no runtime overhead; and (ii) it leads to smaller executable images, since the aspect program is not linked with the affected program.

Along these lines, it became clear that all aspect features may be directly realized via a respective aspect library working on ASTs. This led us to the idea of turning aspect programs to normal language programs taking as input the AST of another program while deploying the aspect library to apply pointcuts and advice directly in the main language. In particular, aspect programs contain a main function, conventionally called *transform*, which takes a single AST argument, transforms it as needed and returns the updated version. To apply a series of aspect programs on a source file we use a special weaver program. The weaver initially takes the source file and parses it into an AST. Then, for each of the given aspect programs, it invokes the *transform* function passing as argument the current AST version which it then updates based on the function's return value. The same process continues until all aspect programs have been applied and the source has been transformed to its final form, encompassing the source code as advised by all the aspects. Essentially, aspect weaving is a batch process of AST transformations (see Figure 6).

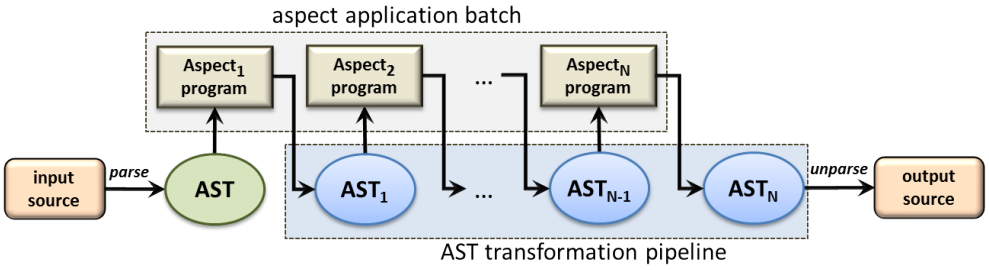


Figure 6 – Source-level aspect weaving as a batch of AST transformation programs.

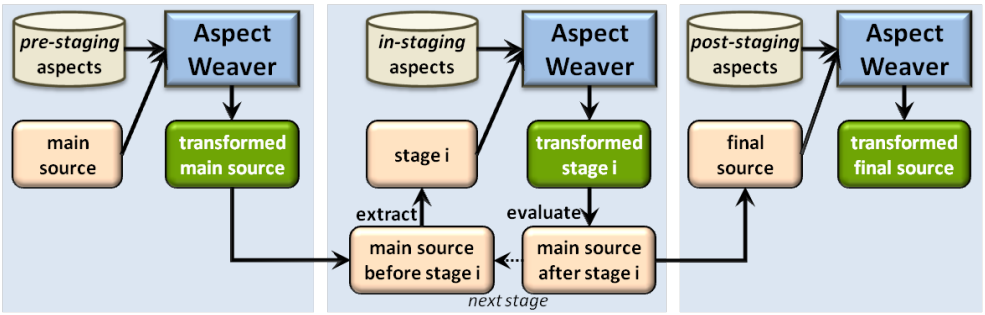


Figure 7 – Overview of all aspect transformation batches occurring during compilation.

The above transformation process applies to all discussed aspect categories. As shown in Figure 7, we have three batches (chains) of transformation programs as follows: (i) one for the original program - *pre-staging aspects*; (ii) one for each stage - *in-staging aspects*; and (iii) one after the staging process - *post-staging aspects*. Once the last batch is applied, the final program version is then compiled to binary form.

From a deployment perspective, we try to minimize the coupling between the compiler and the aspect weaver and achieve a uniform invocation style. In fact, they never communicate explicitly, but are coordinated by the build system. To this end, the aspect weaver always receives a batch and an affected source file as input, and produces a source file as output. In this sense, the weaver is unaware of the previously mentioned aspect categories. It simply applies the current transformation batch to the input source file. Along these lines, the compiler also receives a source file as input by the build system. With aspects present, the transformed source version is supplied to the compiler; otherwise the original source file is directly supplied. The reason for unparsing the result of every aspect transformation batch, and not maintaining it in the form of an AST, is for simplicity, since this way, we retain the original compiler accepting directly source text.

This approach has many advantages compared to custom aspect languages. Firstly, no second language and translator are deployed. Secondly, by turning aspects to normal programs they can be directly hosted in the language IDE and be normally debugged. Thirdly, aspects become first-class IDE citizens and thus can be managed under the same umbrella with the programs they actually transform. Finally, their software engineering directly reuses the techniques and constructs of normal programs, not requiring reinventing the wheel as with aspects languages (e.g. aspect inheritance is essentially reintroduced).

Overall, we realized that the AST manipulation elements are not merely some utility elements for staged languages, but could play a fundamental role in all cases where source code needs to be manipulated. In particular, if supported with the required IDE extensions, they can substitute transformation languages by a combination of processes and libraries.

## 4.2 Aspect Transformation Library

As previously discussed, the metaprogramming elements of the language are sufficient for any AST transformation and thus for introducing cross-cutting functionality. However, we further facilitate the development of aspect programs by providing an AST transformation library with functionality that resembles the typical AOP style.

Since we target AST transformations, joinpoints essentially match specific AST node types to which advice functionality can be added. For example, we support the typical joinpoints like the call and execution of a function or method, the execution of an object constructor, the getting or setting of an object field and the execution of an exception handler. Each of them correspond to specific AST nodes; the function execution corresponds to the AST of the body of the matched function while the execution of an exception handler corresponds to the AST of the matched exception handler's body. Pointcuts are expressed as string literals and are matched against AST nodes using a custom pattern matching language. For example, the pointcut `method m(*)` will match nodes corresponding to method definitions with name m and any number of arguments. We support the typical pointcuts covering the basic joinpoints, pointcut combinators (i.e. `and`, `or`, `not`) for composition as well as some pointcuts specific for AST manipulation. For instance, the `ast(type)` pointcut matches all AST nodes that have the given type, the `parent(pattern, [childId])` matches parent nodes whose child at index *childId* (or any child if not specified) satisfies the given pattern while the `descendant(pattern)` matches the nodes that are part of a sub-tree whose root node is of the given type. Such pointcuts allow specifying fine-grained aspect transformations on a target AST. For example if we want to advise the `break` statements of a `for` loop within some method `m` we can use the following pointcut: `"ast(break)and descendant(for)and descendant(method m(*))"`. In the same sense, the pointcut `"ast(assign)and parent(id(x), lvalue)"` will match all assignments whose child with index *lvalue*, i.e. whose left value, is an identifier $x$, for instance `x = 1`, `x = f()`, etc.

The main function of our library is `aspect(target: ast, pointcut: string, advice_type: enumerated, advice: ast)` that given a target AST and the pointcut to match will insert the advice AST as specified by the advice type. The target argument may specify either the entire program AST or any of its sub-trees that may have been obtained through custom AST traversal or prior node matching against some criteria. Regarding the advice type, we support *before*, *after* and *around* advice, meaning that the given code may be inserted respectively before, after or around the matched joinpoint. The exact way that advice code is inserted depends on the joinpoint and the matched AST node; for example, when we match the execution of a function, *before* advice inserts the given code at the beginning of the matched function body, while *after* advice inserts its code at all exit paths of the matched function body. For *after* advice applied on function or method execution in particular, we can use the delayed escape `<< ... ~~retval ... >>` that will carry the original return value of the function. Another delayed escape, specifically `<< ... ~~proceed ... >>`, is also typically used in *around* advice. The advice is applied by firstly substituting the AST being advised with the given advice AST and then by replacing the delayed escape (`~~proceed`) with the original AST value. For example, when applying the around advice `<<print("before"); ~~proceed; print("after")>>` on the AST of a function call `<< f()>>` the result will be `<<print("before"); f(); print("after")>>`. This construct can also be combined with ASTs that contain staging annotations. For example, the advice `<<!(~~proceed)>>` can transform the expression `f()` into `!(f())`, while the advice `<<~~(~~proceed)>>` can transform `x` into `~x` (the first delayed escape represents a single `~` while the `~~proceed` is typically replaced by the target AST, here `x`). Essentially, when using around advice, the last argument to the aspect function can be seen as a process that takes the matched AST and transforms it as described by the target AST.

| **Library functions** | ***aspect***(target:*ast*, pointcut:*string*, advicetype:*enumerated*, advice:*ast*):*void*<br>***match***(target:*ast*, pointcut:*string*) : *list<ast>*<br>***advise***(target:*ast*, advicetype:*enumerated*, advice:*ast*) : *void* | | |
|---|---|---|---|
| **Advice type** | *BEFORE* | *AFTER* | *AROUND* |
| **Basic Pointcuts** | **execution**(*pattern*)<br>**exception**(*pattern*) | **call**(*pattern*)<br>**class**(*field*) | **setter**(*pattern*)<br>**getter**(*field*) |
| **AST Pointcuts** | **child**(*pattern* [, *childId*])<br>**parent**(*pattern* [, *childId*]) | **descendant**(*pattern*)<br>**ascendant**(*pattern*) | **ast**(*type*)<br>**construction**(*pattern*) |
| **Pointcut Combinators** | *pointcut* **and** *pointcut* | *pointcut* **or** *pointcut* | **not** *pointcut* |

Table 3 – Overview of the basic elements offered by our AOP library

To allow explicit transformation logic while still relying on pattern matching we also provide two additional functions: `match(target: ast, pointcut: string)`, that will find and return all nodes within the target AST that match the given pointcut and `advise(target: ast, advice_type: enumerated, advice: ast)` that will insert the advice AST in the target as specified by the advice type. A summary of the basic elements offered by our AOP library is provided in Table 3.

## 4.3 Aspects in the Workspace Manager

In a system supporting binary-level weaving, the aspect sources are typically placed along with the normal program sources in the workspace management. For instance, in the *AJDT* [Fou] Eclipse plugin for AspectJ, there are aspect-enabled projects that can host both normal Java and AspectJ source files whose generated code is woven together after compilation. In a system with source-level weaving, the aspect transformation has to be in executable form while a normal program is still in source code waiting to be transformed before its compilation. This means that aspect sources and normal program sources are compiled at different times and thus should be properly distinguished in the workspace management. Particularly in our system, where aspects are implemented as typical programs within the same language and their separation with normal programs relies only on their different deployment, supporting such a distinction is even more critical.

Our system supports this distinction by introducing the notion of *aspect sources* that are organized in *aspect projects*. An aspect source contains all typical source information required for its build and deployment (e.g. compilation flags, dependencies, runtime libraries, etc.) as well as information about its transformation purpose, i.e. if it is a pre-staging, an in-staging or a post-staging aspect. This information is explicitly provided by the programmer who specifies the transformation category for each aspect source. In fact, for a single aspect it is possible to specify more than one category, for instance both in-staging and post-staging. The reason for this is that stages may involve computations typically found in a normal program, so they may also require similar crosscutting functionality. Thus, a single aspect is allowed to address both stages and normal programs.

Aspect projects are used for grouping aspect sources and allow specifying the ordering of multiple aspect sources of the same type. For each project or source file

within the workspace, the IDE allows specifying the aspect projects that will be used to advise it. As aspect programs are also programs, aspect sources inherit all properties of normal sources and they can be advised as well. This means that it is possible to use an aspect transformation to manipulate the code of another aspect transformation (but not of itself, as that would require a pre-existing binary of its code).

## 4.4 Aspects in the Build Process

Aspect transformation may be part of the compilation loop; however the aspect weaver need not be tightly coupled with the compiler. Actually, they may both be unaware of the existence of the other and let the build system orchestrate their interoperation.

The aspect weaver just takes an input source, transforms it one or more times and gives as output the resulting output source, thus naturally involving no additional interoperation with either the compiler or the build system. On the other hand, the compiler receives as input a source file and gives as output a binary file; however it requires interoperating with the build system to handle the build process of any stages involved in the process. In this sense, interaction between the build system, compiler and aspect weaver, illustrated in Figure 8, is as follows. When a source is to be built, the build system invokes the weaver with that source as input (step 1), applies the associated pre-staging aspects, receives its output (step 2) and then uses that as input to the compiler (step 3). Then, during the staging pipeline, the meta-compiler assembles the stage source (step 4) and asks the build system to build it (step 5) and provide its binary code; that code will then be execute to update the AST of the initial program being compiled. After receiving the stage source, the build system can invoke the aspect weaver to apply the in-staging aspects (step 6), get the transformed stage source (step 7) and send it for compilation on a new compiler instance (step 8). The nested compilation will provide the stage binary (step 9) that the build system can then supply to the original compiler (step 10) to continue its stage execution. After the current stage execution, if there are still additional stages the same process is repeated (step 11). Eventually, there will be no more staging and the source code resulting from the staging process is ready to be built (step 12). This

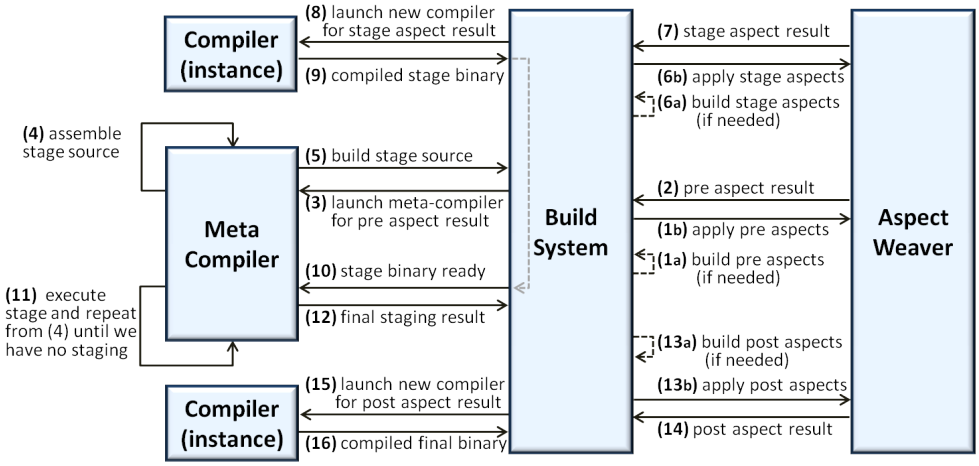


Figure 8 – Interaction diagram between the build system, compiler and aspect weaver.

final source is then propagated to the weaver for applying the post-staging aspects (step 13), and the result is sent to yet another compiler instance (step 15) that will generate the final binary code (step 16).

This description assumes that the aspect transformations are already available in binary form. In general, this may not be the case, so before applying any aspect transformations, the build system may first have to build them to get their binary form (Figure 8, steps 1a, 6a, 13a). Such build steps are automatically perform by the system if needed and may involve additional meta-compilation, in case the target aspect source contains meta-code, or even recursive aspect transformations, if the target aspect is also advised by another aspect. Essentially, a single build request for the initial source may trigger multiple nested build requests for metaprograms or aspect programs involved directly or indirectly in the process. For instance, consider a pre-staging aspect source that contains meta-code and a normal source with no meta-code that will be advised by the aspect source. When trying to build the normal source we require the binary of the aspect source, meaning we have to build it first (normal source, step 1a). Since the pre-staging aspect involves no aspect transformations of its own, it is directly sent to the meta-compiler that handles the staging process and returns a binary for it (aspect source, steps 3-5, 8-12 and 15-16). Then we can continue with the weaving of the normal source and the subsequent meta-compilation process that will result in the final binary (normal source, steps 1b-5, 8-12 and 15-16). Of course, if either the aspect source or the original source were advised by additional in-staging or post-staging aspects, there would be further aspect weaver invocations (i.e. steps 6-7 and 13-14) and possibly additional nested build requests. Overall, the build process is a recursive process that relies on the following principles:

- Building a source advised by specific aspect projects requires recursively building all aspect sources of these projects, invoking the aspect weaver to transform the initial source and then recursively building the last transformation result.
- Building a source with no aspect transformations, but containing meta-code requires assembling each stage, building it recursively, executing its binary code to update the main program AST and finally recursively building the final source when no more meta-code is present.
- Building a source with no aspect transformation or meta-code requires recursively building any module dependencies for the initial source and then finally typically translating its source code into binary.

# 5 Case Studies

To show that the proposed approach is not just a theoretical concept but also has practical applications, we provide example scenarios focusing on pre-staging and in-staging aspects. For post-staging aspects we do not provide explicit scenarios as any typical AOP aspect is also a valid post-staging aspect.

## 5.1 Aspects to Insert Staging

In [Tah04], Taha describes a methodology for taking conventional programs and turning them into multi-stage programs thus reducing potential runtime overhead and improving performance. For instance consider the classic *power* example.

```
function power (x, n) { // original power version
    if (n == 0) return 1;
    else return x * power(x, n - 1);
}
a = 2; print(power(a, 4)); // recursive invocation at runtime
function spower (x, n) {   // staged power version
    if (n == 0) return <<1>>;
    else return <<~x * ~(spower(x, n - 1))>>;
}
a = 2; print(!(spower(<<a>>, 4))); // final code is: print(a*a*a*a*1);
```

Typically, the staged version has to be explicitly written by the programmer. However, it is possible to turn the methodology into an algorithm allowing the automation of this process (i.e. a methodology for transforming function `power` into `spower` automatically). Implementing such an algorithm requires analyzing the AST of the target program to locate potential for deploying staging and then transforming it appropriately to introduce the necessary staging annotations. In this sense, the application of the algorithm can be seen as an aspect-oriented transformation that weaves the advice functionality, i.e. staging annotations, at the desired pointcuts, i.e. source locations with staging potential. In particular, this is a *pre-staging aspect*; it will transform the initial source, originally containing only normal code, to enrich it with staging annotations. The aspect is shown below with `transform` being its entry point. Notice the use of the previously described AST-wise pointcuts like *descendent*, *child*, *ast*, etc. used for fine-grained AST matching.

```
toAST = << <<~~proceed>> >>; //transforms x into <<x>>, used below
function InFunc(name) { return "descendant(function "+ name +"(*))"; }
function MatchCall(name) { return "call(" + name + ")"; }
function StageDefinition(func) { //stage the function body
  local addEscape = <<~~(~~proceed)>>; //transforms x to ~x
  local recursiveCall = MatchCall(func.GetName());
  aspect(func,recursiveCall,AROUND,addEscape);//escape recursive calls
  local exprs = match(func, "child(return)");//begin with return exprs
  while(not exprs.empty()) { //until handling all exprs for the result
    local dependencies = listnew(); //holds deps for the current exprs
    foreach(local expr, exprs) {
      advise(expr, AROUND, toAST); //turn expr into AST form
      ids=match(expr,"id(*) and not descendant("+recursiveCall+")");
      foreach(local id, ids) { //for matched ids (args&locals) in expr
        dependencies.pushback(id.GetName());//mark id as a dep
        advise(id, AROUND, addEscape); //escape the id
      }
    }
    exprs.clear();
    foreach(local dep, dependencies) { //check for assignments to deps
     assigns=match(func,"ast(assign) and parent(id("+dep+"),lvalue)");
     foreach(local assign, assigns) //recursively for matched assigns
       exprs.pushback(assign.GetChild("rvalue"));//check the rvalues
    }
  }
}
```

```
function StageCalls(ast, funcName) {
 calls = match(ast, MatchCall(funcName)+"and not "+InFunc(funcName));
 foreach(local call, calls) { //for each matched non-recursive call
  foreach(local actual, call.GetActuals()) //iterate over actuals
   if(not actual.IsConst()) advise(actual,AROUND,toAST); //stage args
   advise(call, AROUND, << !(~~proceed) >>); //stage entire call
 }
}
function transform (ast) { //ast holds the code to be transformed
  foreach(local func, match(ast, "function *(*)"))//find all functions
    //can the result be expressed as a math function over input args?
    if (CanBeStaged(func)) {
      StageDefinition(func);       //stage the function definition
      StageCalls(ast, func.GetName()); //stage calls in the entire ast
    }
  return ast; //the transformed ast is the aspect weaving result
}
```

In particular, the aspect will first try to find functions that have potential for staging. Without going into details, this process essentially looks for functions whose result can be expressed in a mathematic expression over their input arguments. Power, as well as other mathematical functions like factorial, fibonacci, etc., fit the above description and will be matched by the aspect. For each of the matched functions, we need to stage both definition and invocations. For the definition, we have to stage all items relating to the function result. In this sense, we begin by properly staging the return expressions of the function while marking any argument or local variables involved in their computation. We then repeat the same process targeting any assignment to the previously marked variables. We properly stage the right hand side of each assignment and mark any additional arguments or local variables involved in its computation. This process continues iteratively until all involved variables have been handled. We should also note that any recursive function invocations are by default considered to be involved in the final function result so they are staged up-front and then excluded from the remaining process (hence the recursive call pointcut). For the power example in particular, this process will transform `return 1;` to `return <<1>>;` and `return x * power(x, n-1);` to `return <<~x * ~(power(x, n-1))>>;`. This is achieved by applying around advice and specifying `toAST = << <<~~proceed>> >>` and `addEscape = <<~~(~~proceed)>>` as advice targets. The former essentially turns `1` into `<<1>>` and `x * power(x, n-1)` into `<<x * power(x, n-1)>>` while the latter further transforms `<<x * power(x, n-1)>>` into `<<~x * ~(power(x, n-1))>>`.

Then, for each invocation, the aspect will introduce the *inline* operator and turn any non-constant argument to its corresponding AST form. In the power example, this process will transform the invocation `power(a, 4)`; `toAST = << <<~~proceed>> >>` turns `a` into `<<a>>`, while `<<!(~~proceed)>>` further transforms `power(<<a>>,4)` into `!(power(<<a>>, 4))`. The result is essentially the automatic staging of all relevant function invocations that achieves the desired performance gain.This would not have been possible without the pre-stage aspect, as the original program contained no staged code and its compilation would yield binary code where all functions and their invocations maintained their original form.

This may not be a representative AOP example, but it shows how a pre-staging aspect should operate, i.e. updating or changing the staging of a program, and

illustrates a scenario were such functionality is useful. In fact, this example relates to partial evaluation and would be typically handled by a partial evaluator without requiring the extra aspect specification. However, the binding-time analysis involved in partial evaluation is not complete and can only approximate the knowledge of the programmer, meaning that explicitly specifying how the code should be staged may yield better results. Additionally, implementing the aspect involves mainly AST manipulation that a programmer is familiar with, while effective use of a partial evaluator involves a steep learning curve [JGS93].

Considering the specific power example, writing and applying the aspect is of course more complicated than staging the code manually. However, the aspect is generic enough that it can be used for a variety of other mathematical functions without the need for manually staging each of them. Additionally, the aspect automatically locates and stages all function invocations; without that, the programmer would have to locate all such invocations (probably multiple ones, scattered in the source code) and stage them manually.

## 5.2 Aspects for Custom Static Analysis

During the compilation process, a compiler typically performs a series of static analysis checks to the program being compiled. However, a programmer is typically not aware of the checks being performed while also being unable to customize their behavior. The latter can be achieved by placing staged code at specific locations within the original source so that their execution performs the desired static analysis checks. In this direction, we can use a *pre-staging aspect* to introduce the custom analysis code along with its deployment. For example, the aspect below introduces staged code to analyze all functions definitions.

```
function transform (ast) {
  local allFunctions = match(ast, "ast(function)")
  foreach(local func, allFunctions) { //iterate over matched functions
    local name = func.GetName();
    //insert a staged call to analyze the matched function
    advise(func,AFTER,<<&analyze(compiler::GetFunctionAst(~name));>>);
  }
  advise(ast,BEFORE,<<&function analyze(f){...}>>);//insert staged def
  return ast;
}
```

## 5.3 Aspects to Introduce Memoization in Stages

To improve runtime performance for mathematical functions involving intense computations a common technique is to generate for them constant tables, i.e. tables that will map specific function arguments directly to a constant value. Such tables can be generated by metaprograms; for example, consider the following code that generates a constant table for a range of Fibonacci numbers. As indicated by the `&` annotation, functions `fibonacci` and `GenerateFibonnaciTable` are staged; in particular, the latter uses the former to calculate the required values and merge them into a constant table that is inlined in the program code (`!(GenerateFibonnaciTable(20))` invocation). At runtime, function fib will provide the result directly by accessing the generated constant table.

```
&function fibonacci(n){//compile time version using normal computation
  if (n == 0 or n == 1) return 1;
  else return fibonacci(n - 1) + fibonacci(n - 2);
}
&function GenerateFibonacciTable(upperBound) {
  local numbers = nil;
  for (local i = 0; i < upperBound; ++i)
    numbers = <<~numbers, ~(fibonacci(i))>>; //merge computed values
  return <<[~numbers]>>; //generate const table with these values
}
function fib(n) { //runtime version using the generated const table
  static table=!(GenerateFibonacciTable(20));//inline const table here
  return table[n];
}
print("fib(15) = ", fib(15));  //call involves no runtime overhead
```

While the above technique improves *runtime performance*, during compilation the metaprogram still has to compute the required values, something that may take a long time. To improve *compile-time performance* (metaprogram execution), we can use *memoization*, i.e. caching the result of a function to avoid recalculating it with the same arguments. This functionality is not coupled to a specific metaprogram but would apply to any metaprogram with similar functionality. As such, it can be expressed as an *in-staging aspect* that will be used for each such metaprogram advising its function invocations with memoization. The *fibonacci* example above can be advised with memoization by the following aspect code:

```
function transform (ast) {
  local pointcut = "execution(function fibonacci(n))";
  local beforeAdvice = <<static memoizer = [];    //memoization cache
                         if(memoizer[n] != nil) return memoizer[n];>>;
  aspect(ast, pointcut, BEFORE, beforeAdvice);
  //memoize the result:~~retval will carry the original return value
  aspect(ast, pointcut, AFTER, <<return memoizer[n]=~~retval;>>);
  return ast;
}
```

## 5.4 Aspects for Locking Shared Objects in Stages

Since stages are normal programs, their execution may involve multiple threads of execution that share various resources. This raises the issue of protecting stage code from possible race conditions by introducing typical synchronization constructs like mutexes. This can be achieved using a locking aspect as shown in the following code.

```
function transform (ast) {
  local inClass = "class(SharedObject)"; //class for synchronization
  local pointcut = inClass + " and execution(method *(*))";
  aspect(ast, pointcut, BEFORE,<<self.mutex.lock();>>);
  aspect(ast, pointcut, AFTER,<<self.mutex.unlock();>>);
  aspect(ast,inClass,BEFORE,<<@mutex:mutexnew()>>); //add mutex member
  return ast;
}
```

## 5.5 Aspects for Tracing Diagnostics in Stages

Stages may contain code that was never part of the original program and thus it may not be easy to trace their execution when they don't behave as expected. In such cases, unless the IDE provides support for debugging metaprograms, the only option is to manually insert logging calls within functions of the stage program to trace their execution. However, logging is a well-known crosscutting concern that can be addressed through AOP. In this sense, using the following code as an in-staging aspect achieves the desired functionality.

```
function transform (ast) {
  local funcs = match(ast, "execution(function *(*))");
  foreach(local f, funcs){ //iterate over matched function definitions
    local name = f.GetName();
    advise(f, BEFORE, <<print("Entering " + ~name);>>);
    advise(f, AFTER,  <<print("Exiting "  + ~name);>>);
  }
  return ast;
}
```

Note that since we use an *in-staging aspect*, the `ast` argument passed to the `transform` function will contain stage code. As such, the tracing functionality is only introduced in functions available in stages, and not the functions of the final program.

## 5.6 Aspects for Exception Handling in Stages

As already discussed, the code of a stage metaprogram may be sophisticated and involve multiple scenarios where errors can occur. In this context it is a typical practice to use exception handling to separate the normal execution from the error handling code. Exceptions can be seen as a crosscutting concern allowing them to be modularized as aspects [KLM+97]. In this sense, stage code could utilize an *in-staging aspect* to be advised with the error handling logic. For example, the following aspect can be used to specify different exception handling policies for a variety of use cases.

```
function AllMethodsInClass(class)//func to create pointcut expressions
 { return "execution(method *(*)) and descendant(class("+class+"))"; }

function transform (ast) {
  aspect(ast, AllMethodsInClass("RemoteObject"), AROUND,
    <<try { ~~proceed; } catch Exception { log(Exception); }>>
  ); //log and ignore any exception from remote object invocations
  aspect(ast, AllMethodsInClass("StackWithDbyC"), AROUND,
    <<try { ~~proceed; } catch ContractException { assert false; }>>
  ); //no contract exceptions allowed with using Design by Contract
  aspect(ast, AllMethodsInClass("ConfigurationManager"), AROUND,
    <<try { ~~proceed; } catch IOException
       { throw [@class:"ConfigException", @source:IOException]; } >>
  ); //hide low level IOExceptions and raise higher level ones
  return ast;
}
```

## 5.7 Aspects for Decorating Classes in Stages

Stages are mainly code generators and thus they make extensive use of AST creation and manipulation. Even if the AST library offered by the language facilitates AST traversal and manipulation, programmers may still want to decorate AST values with custom functionality. To do so, one would have to implement an additional library and manually decorate AST creation occurrences in the code. The latter can be seen as a crosscutting concern that can be addressed through the following *in-staging aspect*. In particular, the aspect will locate all quasi-quotes nodes (i.e. AST creations) and apply the desired decoration based on the language element they contain. Of course, any *inlines* and *escapes* have to be advised as well to retrieve the original AST value from the decorated object.

```
function transform (ast) {
  local quotes = match(ast,"ast(quasiquote)");//find all AST creations
  foreach(local quote, quotes) {
    if (quote.GetChild().GetType() == "class")
      advise(quote, AROUND, <<[ //AST creations replaced with objects
        @ast : ~~proceed,       //original AST stored as normal data
        method GetMethods   (){...},  //custom methods added
        method GetAttributes(){...},
        method BaseClasses  (){...}
      ]>>);
    else if (quote.GetChild().GetType() == "function")
      advise(quote, AROUND, <<[ //AST creations replaced with objects
        @ast : ~~proceed,       //original AST stored as normal data
        method GetName  () {...},  //custom methods added
        method GetActual(n){...},
        method GetLocals() {...}
      ]>>);
    else //perform similar handling for other quoted language elements
  }
  //replace escape and inline targets with original AST
  aspect(ast, "child(escape)", AROUND, <<~~proceed.ast>>);
  aspect(ast, "child(inline)", AROUND, <<~~proceed.ast>>);
  return ast;
}
```

## 5.8 Aspects for Custom AST Iteration in Stages

It is typical for staged code to traverse the tree structure of an AST value. In Delta, this is achieved through an AST visitor, where node types are associated with handler functions. For example, the following code will traverse the AST shown and invoke the associated handler (i.e. the anonymous function) for every function node contained within the AST.

```
ast = << function f() { return << function g() {} >>; } >>;
visitor = astvisitornew();
visitor.sethandler("function", function(node, id, entering){ ... });
ast.acceptpreorder(visitor);
```

The visitor does not differentiate between functions being directly within the traversed AST or inside any nested quasi-quotes it contains, meaning that in this example the handler will be triggered by both functions `f` and `g`. However, it is very common for the traversal to target only functions directly within the AST and not nested ones. To achieve this, we have to introduce additional handlers to keeps track of the stage nesting and modify existing ones to utilize this information. This can be modeled with the following *in-staging aspect*:

```
function transform(ast) {
 visitors = "ast(assign) and parent(call(astvisitornew()),rvalue)";
 foreach(local visitor, match(ast, visitors)) {
  local handlers = "execution(function (node, id, entering)) and
   descendant(call("+id.GetName()+".sethandler(*))"; //match handlers
  local advice = <<if (nesting==0)~~proceed;>>;//consider func nesting
  aspect(visitor.GetEnclosingBlock(), handlers, AROUND, advice);
  local id = visitor.GetChild("lvalue").copy();
  advise(visitor,AFTER,<< //introduce nesting and increase/decrease it
    local nesting = 0;  //as needed in quasi-quote and escape handlers
    ~id.sethandler("quasiquotes", function(node, id, entering)
      { if (entering) ++nesting; else --nesting; });
    ~id.sethandler("escape", function(node, id, entering)
      { if (entering) --nesting; else ++nesting; });
    >>);
  }
  return ast;
}
```

Notice that we first update existing handlers and then introduce the new ones to avoid advising them or having to specify a more complex pointcut that excludes them.

## 5.9 Aspects for AST Validation in Stages

ASTs are usually constructed through *quasi-quotes*, however they cannot express structures depending on some computation, for example having an `if` statement with a variable number of `else if` clauses. To allow generating such code patterns, metalanguages typically provide some extra facility, like a library for explicit AST creation and manipulation. ASTs created using either the library or through quasi-quotes should interoperate; however while ASTs created by quasi-quotes are well-formed, ASTs created through the library may be incomplete or even ill-formed. In this context, a programmer could insert custom validation code at specific source locations, ensuring that any AST is well-formed and that any manually constructed erroneous AST is reported as early as possible. This functionality can be achieved through an *in-staging aspect* by introducing a `validate` function weaving invocations to it at locations where a manually constructed AST is combined with an AST created through quasi-quotes, along with any staged function that operates on an AST.

```
function transform(ast) {
  local validator = << function validate(ast) { ... return ast; } >>;
  advise(ast, BEFORE, validator); //introduce the validate function
  //turn <<... ~x ...>> into validate(<<... ~(validate(x)) ...>>)
  aspect(ast,"ast(quasiquotes)", AROUND, <<validate(~~proceed)>>);
```

```
  aspect(ast,"child(escape)", AROUND, <<validate(~~proceed)>>);
  //turn !(...) into !(validate(...))
  aspect(ast,"child(inline)", AROUND, <<validate(~~proceed)>>);
  aspect(ast,"execution(function *(ast,*)) and not execution(function
    validate(*)",BEFORE,<<validate(ast);>>); //validate AST arguments
}
```

# 6 Debugging Aspects

Utilizing aspects or metaprograms in a development process is a challenging task on its own. Trying to combine the two presents an inherently increased level of complexity, requiring the IDE to provide advanced tool support for writing and debugging programs in order to help programmers in this demanding task. In this direction, we extend our previous work on tool support for metaprogramming [LS12b], to also support aspect-oriented transformations. Since we build upon our implementation for aspect support, the discussion is focused on source-level weaving. Nevertheless, the feature implementation could also utilize binary-level weaving, while the rationale for their support is still valid in both cases.

## 6.1 Reviewing Woven Code

When aspect code is woven together with normal program code, either through source- or binary-level weaving, the result is a transformed version of the code that is not available to the programmer. This may not be an issue when the resulting code behaves as expected or the aspect is simple enough to verify its functionality in a few execution sessions. However, if the resulting code does not behave as expected or the aspect involves some complex pointcuts, information about the final version of the code can be invaluable to programmers, allowing them to see how the aspect application transformed the code and figure out the reason of the erroneous behavior. Reviewing the results of aspect weaving can provide helpful information even when the resulting code executes correctly, as it enables programmers to move from an abstract representation of the final code to a concrete visualization, increasing their understanding of the transformation that takes place.

This is a similar requirement with the reviewing of the updated version of the main program after having evaluated some stage metaprogram [LS12b]. As such, it is addressed in a similar way by *unparsing* the AST produced by the aspect transformation into source text, storing that text into a source file, and finally notifying the IDE to insert it in the workspace, properly associating it with the original source. Sparrow already supports this functionality for metaprogram results, so the only addition required involves the aspect weaver. In case of multiple aspects, the weaver generates an updated version of the source code after applying each separate transformation, thus providing a full trajectory of the transformation process. Figure 9 illustrates a sample workspace involving files generated by both staging and aspect transformations.

## 6.2 Providing Accurate Compile Errors

When the source code of a program that has passed through multiple transformation steps contains errors, it is not clear whether the error was present in the original source or if it was introduced as a result of one or more of the transformations [Tra08]. In this

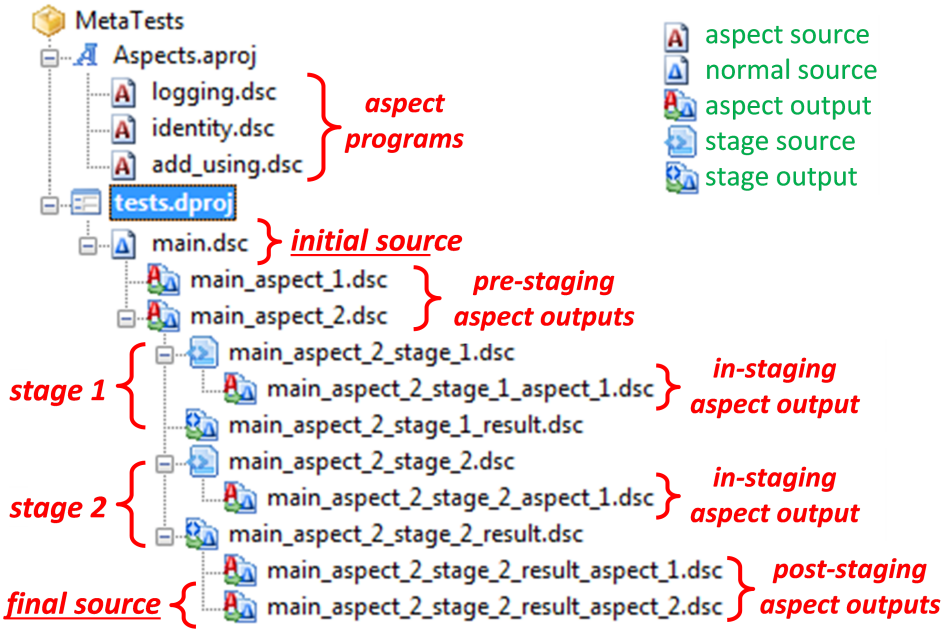


Figure 9 – Sample workspace with sources generated by staging and aspect transformations

sense, instead of a single error report specifying the final error location, the compiler should be able to track down and provide the first introduction of the erroneous code as well as the complete transformation chain that led to its final form. To achieve this functionality it is possible to associate any generated source location with the source location it originated from before the transformation took place. This way, we can create a list of source references that can track the error across all source files involved in the compilation.

A similar error tracking scenario is involved in the typical metaprogramming process requiring the creation and maintenance of a list of source references [LS12b]. However, in that case the entire process takes place within the compiler that simply provides the IDE with all relevant source reference information upon generating the error report. With the introduction of aspect transformations, we have a separate aspect weaver process responsible simply for source transformations and unaware of the source references maintained by the compiler. Since the aspect weaver and the compiler have to be as loosely-coupled as possible, the infrastructure for the source references is moved within the IDE, stored as metadata accompanying each generated source file. This way, whenever a generated source is created by a transformation process, either due to aspects or metaprogramming, we also provide the associated source references to the IDE. With this information, the IDE can then track down all sources involved in the generation or transformation of an error and form the entire transformation chain, properly associating it with any issued error report. This functionality is illustrated in Figure 10, where the given error report refers to all source files involved in the generation of the erroneous code. This allows the programmer to navigate across the various source files versions (as discussed they are available in the workspace), reviewing the transformations performed from one version to the next and eventually understanding which transformation introduced the error.

```
Error, file 'main_aspect_1_stage_2_result_aspect_2.dsc', line 19:
Expression '(g)' not a callable value (its type is 'Undefined').
See file 'main_aspect_1_stage_2_result_aspect1.dsc', line 18.
See file 'main_aspect_1_stage_2_result.dsc', line 12.
See file 'main_aspect_1_stage_2_aspect1.dsc', line 12.
See file 'main_aspect_1_stage_2.dsc', line 12.
See file 'main_aspect_1_stage_1_result.dsc' line 12.
See file 'main_aspect_1_stage_1_aspect1.dsc', line 32.
See file 'main_aspect_1_stage_1.dsc', line 32.
See file 'main_aspect_1.dsc', line 32.
See file 'main.dsc', line 11.
Finished compiling 'main_aspect_1_stage_2_result_aspect_2.dsc',
1 ERROR detected.
```

Figure 10 – Tracing compile errors in the source transformation pipeline involving staging and aspects; source names refer to the workspace of Figure 9

## 6.3 Tracing the Evaluation of Aspects

Being able to view the result of an aspect transformation is a step closer to debugging aspect applications; however it lacks the information about *how* the code reached its final form. Such information involves tracing the entire control flow of the transformation logic as well as inspecting the transformation data. Essentially, the requirement is to provide full-fledged source-level debugging of the aspect program that is invoked to perform the transformation. Any aspect program is executed during the compilation process and performs AST modifications, so it resembles the execution of a normal compile-time metaprogram. In this sense, we can reuse the IDE debugger front-end functionality for compile-time debugging [LS12b] and instrument the aspect weaver with a debugger back-end that will handle the execution of the aspect programs. This way we can support debugging of the aspect transformation logic without practically making any changes to the existing infrastructure. An example of such a debugging session is illustrated in Figure 11.

As we mentioned earlier, in-staging aspects transform stage metaprograms before they are evaluated, so the two execute sequentially. From a debugger perspective, this means that the front-end has to be able to support multiple different back-ends. Additionally, all stage executions take place within the compiler, meaning they are served by a single debugger back-end, while the aspect transformations for different stages are executed by different aspect weavers, meaning they are served by multiple debugger back-ends. Essentially this means that the debug session of the stage metaprogram can be interleaved with the debug sessions of the in-staging aspects.

For example consider a program with two stages of metaprograms where each of these stages is subject to an aspect transformation. When compilation begins, the debugger back-end within the compiler will connect with the IDE front-end. After the first stage is composed it will be sent to the aspect weaver for transformation. Upon launching the weaver, its debugger back-end will also connect to the IDE front-end overriding the previous connection. The weaver will then proceed with the execution of the aspect transformation that is the first program that will be debugged. After finishing the transformation, the debugger back-end of the weaver disconnects from the IDE front-end, restoring the compiler debugger back-end as the active one. The

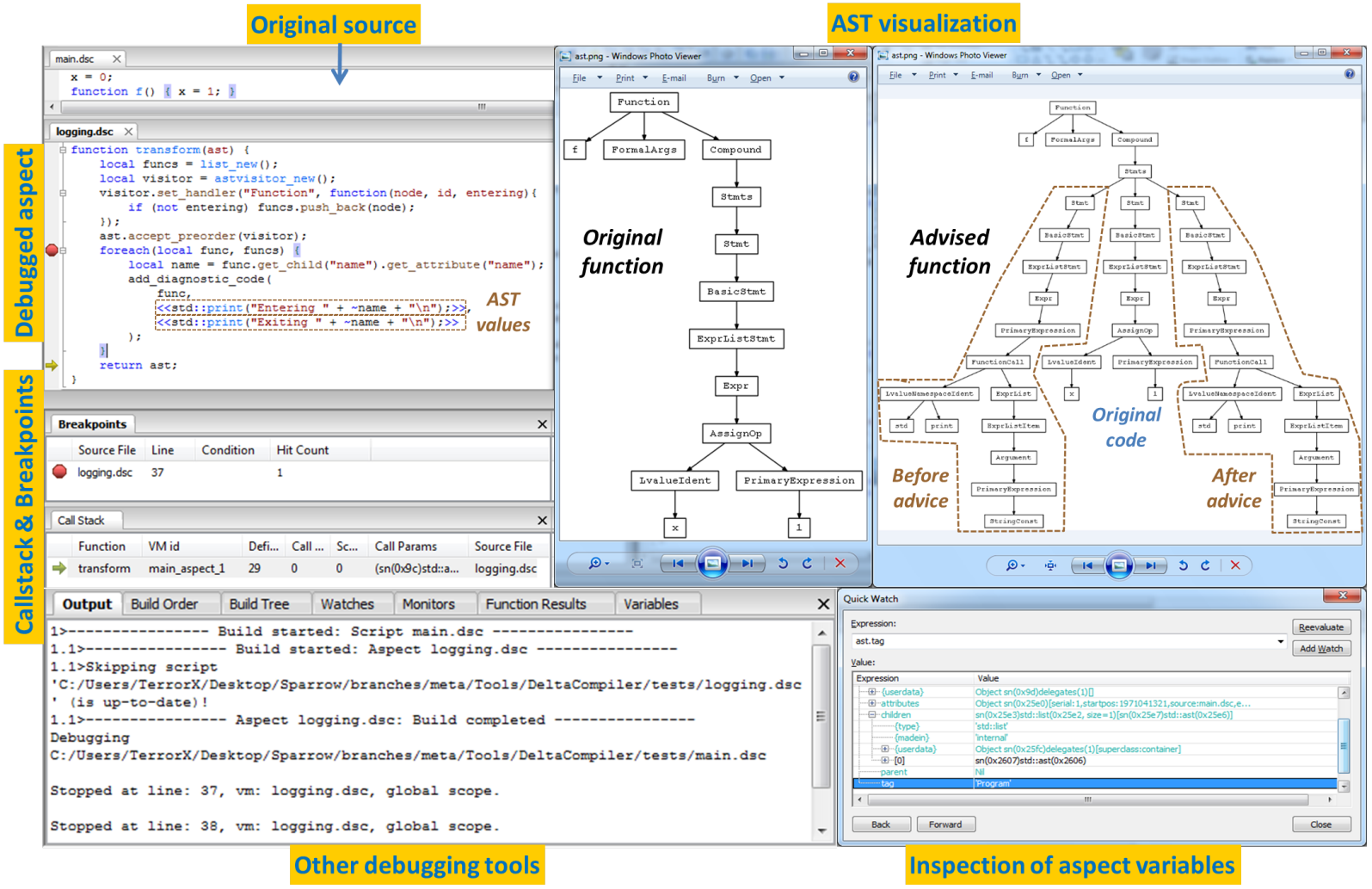


Figure 11 – Full-scale source-level debugging of aspect programs when they are actually applied *during compilation*.

Yannis Lilis, Anthony Savidis. *Aspects for Stages: Cross Cutting Concerns for Metaprograms*. In Journal of Object Technology, vol. 13, no. 1, 2014, pages 1:1–36. doi:10.5381/jot.2014.13.1.a1

compiler then proceeds with the execution of the first stage that is the second program that is debugged. When the stage execution is finished, the initial program AST is transformed and ready to compose the second stage. The same process is repeated, with the launch of the aspect weaver creating a new active debugging connection for the aspect transformation of the second stage and its termination restoring the compiler's debugging connection as the active one, to finally debug second stage execution.

Offering the functionality described above requires extensions in the infrastructure; in particular allowing the debugger front-end to handle multiple back-ends. However, considering the execution order of the systems involved in the process, namely the compiler and the instances of the aspect weaver, it is clear that no two systems may run in parallel. This means that the only required extension is to support adding and removing debugger connections, while typically serving the one added most recently.

# 7 Related Work

To our knowledge, this is the first work with a systematic proposition towards supporting aspects for stage programs in the context of MSLs. However, we consider our work to be closely related to the attempts of using metaprogramming features for achieving aspect-orientation. For example, *AOP++* [YZC05] is a generic AOP framework in C++ that utilized the metaprogramming constructs of the language, i.e. templates, to express pointcut expressions and match joinpoints at compile-time. *Nemerle* [SMO04] facilitates metaprogramming through its macro system and can support AOP features by applying annotation based macro invocations on program classes. *AspectR* [BF02] is a library for Ruby that utilizes metaprogramming techniques to implement AOP by wrapping code around existing methods in classes. *Groovy AOP* [KG08] is an AOP system for Groovy that provides a hybrid dynamic AOP implementation based on both metaprogramming and byte-code transformation. Aspects, pointcuts and advice are specified at compile-time based on a Groovy based domain-specific language while the advice is woven into byte-code at runtime using dynamic compilation.

Languages like Lisp or Scheme have a built-in notion of stages, while they also facilitate AOP through library support, for example using *AspectL* [Cos04] or *AspectScheme* [DTK06] respectively, thus allowing potentially combining stages and aspects. However, these libraries target generic AOP and do not provide explicit support for introducing aspects in staged code. This means that while from an expressiveness point of view it is possible to specify aspects for stages, from a software engineering point of view it requires introducing additional sophisticated macros, something difficult even for advanced users. The latter could be easily addressed with a dedicated AOP library for stages offering such macros out of the box and thus facilitating the adoption of AOP practices in staged code. It also shows that even languages with both concepts require a more systematic approach for their combined deployment.

*MorphJ* [HS11] is a language that introduces a form of meta-programming by enabling the specification of general classes that are produced by iterating over members of other classes. In this sense, it can also be used to achieve AOP functionality by advising structural program features (e.g. method before-, after-, and around-advice). As a program generation or transformation approach, MorphJ only allows enhancement of classes through subtyping or delegation. On the contrary, our system allows arbitrary code generation or transformation making it more expressive. As an AOP approach, MorphJ allows advising normal program code but cannot support advising its reflective transformation functionality, i.e. the metaprogram specifying the

general class generation. A fundamental point of our proposition is that metaprograms may also require AOP functionality, so we support all stages of a multi-stage program to be subject to AOP. The advantage of MorphJ over our system (or other AOP tools) is the guarantee of modular type safety enabling the general classes to be type-checked independently of their uses. Indeed, in our system it is possible for an aspect program to be valid on its own, but cause errors upon its deployment. However, such an error is still reported during compilation, while the offered error reporting facility discussed in section 6.2 allows it to be easily identified and thus resolved.

There are also systems that provide dynamic AOP support through meta-object protocols or byte-code modification at load- or run-time. Examples include but are not limited to *JAC* [PDFS01], *Handi-Wrap* [BH02] and *Spring* [Joh11] for Java, *AspectS* [Hir02] for Smalltalk and *AspectLua* [CBF05] for Lua. This approach is orthogonal to our work that focuses on systems with static AOP (like AspectJ).

Existing tools for debugging code involving AOP are also relevant to our work. For example, [EAH+07, YBA12] offer support for debugging the final woven code while properly associating execution with the original source code or the aspect source code. Our system relies on source-level weaving and keeps the results of each aspect transformation, so source-level debugging of the woven code is straightforward by using the result of the final aspect transformation. Instead we focus on providing source-level debugging support for the *transformation logic*, allowing programmers view normal and aspect related code as ASTs and trace the entire weaving process.

The application of aspects has been applied in Model-Driven Engineering (MDE) as well, introducing cross-cutting concerns directly at the modeling domain. The latter requires custom MDE languages to represent aspects, in particular to express pointcuts and advice, and respective weavers operating on models. Examples of such work are discussed in [SHU02], [CvdBE07] and [FS07]. In our context, such techniques could be applied assuming a notion such as *staged models* is defined and supported in the MDE arena. In other words, *staged models* could be considered as models resulting from the evaluation of *generative models*, something analogous to source code staging. Now, while meta-modeling is widely used in MDE, it has a connotation quite different to the genuine generative behavior of stages in MSLs. Overall, once staged models are somehow supported, aspect categories like the ones proposed may directly apply.

# 8 Discussion

We continue by discussing elements that may differ in other languages and provide an overview for deploying our approach using a mainstream AOP language like AspectJ.

***Scope extrusion*** In the Delta language, variables within quasi-quotes are resolved with their name in the context where the quoted code will actually be inserted, i.e. are lexically scoped at the insertion point. In this sense, there are no guaranties regarding name bindings and as such no scope extrusion issue. However, our proposition towards AOP for stages is orthogonal to such an issue. In a language where symbols within quasi-quotes bind to specific variables via lexical scoping, the same language facilities that are used to guarantee the name binding for normal program compilation can be extended to also apply for any aspect transformations. For example, Template Haskell [SJ02] and Converge both use the notion of *original names* to bind quasi-quoted symbols to top-level definitions within a module. In particular, for a top-level function `f` within a module `M`, any reference to `f` used within quasi-quotes is directly translated to `M:f` (`M.f` for the Converge version), uniquely referring to the particular name. In

the same sense, any quasi-quote of an aspect transformation could also refer to the same function `f` using an extension for original names. Since the name of the module is not directly available during the compilation of the aspect program, we could instead use a special delayed escape `~~module` that will be replaced with the name of the module when it becomes available. This way, we could directly use original names within quasi-quotes, for instance writing `<<~~module:f>>`.

***Interaction and Commutation Among Aspects*** Our implementation realizes aspects as separate AST transformation programs that are applied sequentially. In this sequence, any aspect being applied operates on the result of earlier aspects, so naturally the application order is important; in fact, applying aspect programs with a different order may yield different results meaning that this is not a commutative process. For example, consider an aspect for introducing additional members to a class and another one for automatically generating accessor functions for the class members. If the former aspect is applied first the resulting class will have accessor functions for all members while if it is applied second any newly introduced members will have no accessor functions. Apart from the ordering issue, aspect transformations are applied once and for all, without the ability to be triggered again by other aspects. Essentially, an aspect may inspect changes introduced by earlier aspects, but not vice versa, effectively disallowing any bidirectional interaction between two aspects. Both limitations arise from the particular aspect implementation as transformation programs and are not inherent issues of our proposition for aspects in stages. In this sense, utilizing a more traditional AOP approach, with a separate aspect language and collective weaving of the aspect code along with the normal program or metaprogram code, aspects of the same category can interact with each other, while their commutation is the same as with normal aspects.

***AOP for stages using Mint and AspectJ*** To apply our methodology using Mint and AspectJ, AspectJ first has to be extended to support the staging extensions of Mint. The latter is required so as to allow the aspect code contain staging annotations. Additionally, the stage binaries produced by Mint need to be available before they are executed so as to be advised by the aspect weaver. Essentially, the translation-execution loop required for the staging process has to provide an entry point allowing updating the original stage binary with the advised one. With these extensions, we can then follow the binary weaving shown in Figure 4. Initially, the original program is compiled to binary and is advised by the pre-staging aspects. As such, the program execution that follows uses the advised version of the binary. At runtime, whenever a stage binary is produced, the aspect weaver can intercept it, advise it with the in-staging aspects and then send it for execution. This way, the stage execution contains both original and aspect functionality. Regarding post-staging aspects, Mint uses runtime metaprogramming so, as previously discussed, they cannot be supported.

# 9 Conclusion

In this paper we focused on the application of AOP in the context of staging and introduced aspect-orientation in the entire processing pipeline of a MSL. Towards this direction we identified three separate categories of aspects that should be supported for expressing transformations for either normal or staged code, either present in the original program or the result of code generation. Each aspect category targets a specific step of the staging pipeline and has a specific goal. Specifically, *pre-staging aspects* target the original code and can be used to introduce staging or transform

existing stages. Then *in-staging aspects* concern stages and apply AOP to affect stage code. Finally, *post-staging aspects* (applicable only in CTMP) aim to affect the final source resulting from the staging process and apply typical AOP transformations.

To realize the proposed methodology we carried out the implementation on an existing language with compile-time multi-stage metaprogramming. Our system provides full support for aspect-orientation in a MSL, relies on source-level weaving and implements aspects as batches of AST transformation programs. This implementation approach seemed to fit well with typical multi-stage metaprogramming practices since programmers are already familiar in using and manipulating ASTs. Additionally, it allows exploiting features like reviewing, inspecting or debugging AST transformations that may already be offered by the language IDE.

Overall, our work aims to setup a discipline for applying AOP in MSLs and hopefully stimulate further work towards combined practices amongst the two worlds.

## References


[Baw99] Alan Bawden. Quasiquotation in Lisp. In *PERM*, pages 4–12, 1999. Available from: `http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf`.

[BF02] Avi Bryant and Robert Feldt. AspectR - simple aspect-oriented programming in Ruby, 2002. Available from: `http://aspectr.sourceforge.net/`.

[BH02] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02*, pages 86–95, New York, NY, USA, 2002. ACM. `doi:10.1145/508386.508396`.

[CBF05] Néélio Cacho, Thaís Batista, and Fabrício Fernandes. A dynamic aop approach. *Journal of Universal Computer Science*, 11(7):1177–1197, 2005. `doi:10.3217/jucs-011-07-1177`.

[CLT+01] Cristiano Calcagno, Queen Mary London, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, 2001. Available from: `http://www.cs.rice.edu/~taha/publications/preprints/pldi02-pre.pdf`.

[Cos04] Pascal Costanza. A short overview of AspectL. In *EIWAS'04*, pages 23–24, 2004. Available from: `http://www.p-cos.net/documents/aspectl-short-final.pdf`.

[CvdBE07] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola WEAVR: Aspect orientation and model-driven engineering. *Journal of Object Technology (JOT), Special Issue on Aspect-Oriented Modelling*, 6(7):51–88, 2007. `doi:10.5381/jot.2007.6.7.a3`.

[DTK06] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, 2006. `doi:10.1016/j.scico.2006.01.003`.

[Dyb09] R. K. Dybvig. *The Scheme Programming Language*. The MIT Press, fourth edition, February 2009.

[EAH+07] Marc Eaddy, Alfred Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In *SC'07*, pages 200–215. Springer, 2007. doi:10.1007/978-3-540-77351-1_17.

[Fle07] Fabien Fleutot. Metalua manual, 2007. Available from: http://metalua.luaforge.net/metalua-manual.html.

[Fou] The Eclipse Foundation. AJDT: AspectJ development tools. Available from: http://www.eclipse.org/ajdt/.

[FS07] Lidia Fuentes and Pablo Sánchez. Designing and weaving aspect-oriented executable uml models. *Journal of Object Technology (JOT), Special Issue on Aspect-Oriented Modelling*, 6(7):109–136, 2007. doi:10.5381/jot.2007.6.7.a5.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *PLILPS '95*, volume 982 of *LNCS*, pages 259–278. Springer, 1995. doi:10.1007/BFb0026825.

[Hir02] Robert Hirschfeld. AspectS: Aspect-oriented programming with Squeak. In *NODe '02*, volume 2591 of *LNCS*, pages 216–232. Springer, 2002. doi:10.1007/3-540-36557-5_17.

[HS11] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33(2):6:1–44, February 2011. doi:10.1145/1890028.1890029.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Joh11] R. Johnson. The spring framework - reference documentation, 2011. Available from: http://static.springsource.org/spring/docs/current/spring-framework-reference/html/aop.html.

[KG08] Chanwit Kaewkasi and John R. Gurd. Groovy aop: a dynamic aop system for a jvm-based language. In *SPLAT '08*, pages 3:1–6, New York, NY, USA, 2008. ACM. doi:10.1145/1408647.1408650.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01*, volume 2072 of *LNCS*, pages 327–354. Springer, 2001. doi:10.1007/3-540-45337-7_18.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997. doi:10.1007/BFb0053381.

[LS12a] Yannis Lilis and Anthony Savidis. Implementing reusable exception handling patterns with compile-time metaprogramming. In *SERENE 2012*, volume 7527 of *LNCS*, pages 1–15. Springer, 2012. doi:10.1007/978-3-642-33176-3_1.

[LS12b] Yannis Lilis and Anthony Savidis. Supporting compile-time debugging and precise error reporting in meta-programs. In *TOOLS 2012*, volume 7304 of *LNCS*, pages 155–170. Springer, 2012. doi:10.1007/978-3-642-30561-0_12.

[LS13] Yannis Lilis and Anthony Savidis. An integrated approach to source level debugging and compile error reporting in metaprograms. *Journal of Object Technology (JOT)*, 12(3):1–26, 2013. doi:doi:10.5381/jot.2013.12.3.a2.

[McC62] J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[NR04] Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In *GPCE '04*, volume 3286 of *LNCS*, pages 168–185. Springer, 2004. doi:10.1007/978-3-540-30175-2_9.

[PDFS01] Renaud Pawlak, Laurence Duchien, Gérard Florin, and Lionel Seinturier. JAC: A flexible solution for aspect-oriented programming in Java. In *REFLECTION 2001*, volume 2192 of *LNCS*, pages 1–24. Springer, 2001. doi:10.1007/3-540-45429-2_1.

[Sav05] Anthony Savidis. Dynamic imperative languages for runtime extensible semantics and polymorphic meta-programming. In *RISE 2005*, volume 3943 of *LNCS*, pages 113–128. Springer, 2005. doi:10.1007/11751113_9.

[Sav10] Antony Savidis. The Delta Programming Language, 2010. Available from: http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html.

[SBG07] Antony Savidis, Themistoklis Bourdenas, and Yannis Georgalis. An adaptable circular meta-ide for a dynamic programming language. In *RISE 2007*, pages 99–114, 2007. Available from: http://www.ics.forth.gr/hci/files/plang/sparrow.pdf.

[She98] Tim Sheard. Using MetaML: A staged programming language. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 207–239. Springer, 1998. doi:10.1007/10704973_5.

[SHU02] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A UML-based aspect-oriented design notation for AspectJ. In *AOSD '02*, pages 106–112, New York, NY, USA, 2002. ACM. doi:10.1145/508386.508399.

[SJ02] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002. doi:10.1145/636517.636528.

[SMO04] Kamil Skalski, Michal Moskal, and Pawel Olszta. Metaprogramming in Nemerle, 2004. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.8265&rep=rep1&type=pdf.

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 30–50. Springer, 2004. doi:10.1007/978-3-540-25935-0_3.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008. doi:10.1145/1391956.1391958.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000. doi:10.1016/S0304-3975(00)00053-0.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *PLDI '93*, pages 156–165, New York, NY, USA, 1993. ACM. doi:10.1145/155090.155105.

[WRI+10] Edwin M. Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *PLDI '10*, pages 400–411, New York, NY, USA, 2010. ACM. doi:10.1145/1806596.1806642.

[YBA12] Haihan Yin, Christoph Bockisch, and Mehmet Aksit. A fine-grained debugger for aspect-oriented programming. In *AOSD '12*, pages 59–70, New York, NY, USA, 2012. ACM. doi:10.1145/2162049.2162057.

[YZC05] Zhen Yao, Qi-long Zheng, and Guo-liang Chen. AOP++: a generic aspect-oriented programming framework in C++. In *GPCE '05*, volume 3676 of *LNCS*, pages 94–108. Springer, 2005. doi:10.1007/11561347_8.

## Availability information

The source code of the aspect weaver, AOP library and IDE extensions is available for download through our Subversion repository `https://139.91.186.186/svn/sparrow/` using a guest account (username: guest and empty password). All case studies are running examples present in the Aspects workspace that is included in the Delta distribution available from `http://www.ics.forth.gr/hci/files/plang/sparrow-setup.exe`.

## About the authors

**Yannis Lilis** owns an MSc from the Department of Computer Science, University of Crete, being currently a PhD student collaborating with the Institute of Computer Science - FORTH.
His e-mail address is `lilis@ics.forth.gr`.

**Anthony Savidis** is a Professor of Programming Languages and Software Engineering at the Department of Computer Science, University of Crete, and an Affiliated Researcher at the Institute of Computer Science, FORTH.
His e-mail address is `as@ics.forth.gr`.