# Towards a Generic Framework for AOP

Pascal Fradet[1] and Mario Südholt[2]

[1] IRISA/INRIA, Campus de Beaulieu, 35042 Rennes cedex, fradet@irisa.fr
[2] École des Mines de Nantes, 4 rue A. Kastler, 44307 Nantex cedex 3, sudholt@emn.fr

## 1 Introduction

During the 1st workshop on AOP [AOP97] several fundamental questions were raised: What exactly are aspects? How to weave? What are the join points used to anchor aspects into the component program? Is there a general purpose aspect language? In this position paper, we address these questions for a particular class of aspects: aspects expressible as static, source-to-source program transformations. An aspect is defined as a collection of program transformations acting on the abstract syntax tree of the component program. We discuss the design of a generic framework to express these transformations as well as a generic weaver. The coupling of component and aspect definitions can be defined formally using operators matching subtrees of the component program. The aspect weaver is simply a fixpoint operator taking as parameters the component program and a set of program transformations. In many cases, program transformations based solely on syntactic criteria are not satisfactory and one would like to be able to use semantic criteria in aspect definitions. We show how this can be done using properties expressed on the semantics of the component program and implemented using static analysis techniques. One of our main concerns is to keep weaving predictable. This raises several questions about the semantics (termination, convergence) of weaving.

## 2 Aspects and aspect definitions

*Component language and program transformations.* We advocate using a single powerful and flexible transformation language for the definition of aspects. First, our framework should be generic with respect to the component language. To this aim, the abstract syntax of the component language is described by a tree data type. The component program is seen and manipulated as a tree. Then, defining aspects for a specific component language can be done on the basis of the abstract syntax definition. A transformation is just a function which maps the tree representing the component program to a new tree. Any programming language could be used; there exist, however, powerful and executable specialized languages which permit to express concisely such transformations. These languages are based on patterns and tree matching operators. TrafoLa-H [HS93] is such a language where transformations are of the form $pat \implies TreeExpr$.

Applied to a source program, it transforms a subtree matching *pat* into the result of the evaluation of *TreeExpr*. The variables occurring in *pat* are bound to subtrees and *TreeExpr* is a functional expression which is evaluated with these bindings.

*Aspects.* The features of TrafoLa-H make it easy to specify join points, both generic ones or join points which are specific to a particular program. For example, assuming an imperative component language, patterns matching "each program point", "each assignment containing a division", or "all calls to the function $f$" can be described succinctly. In this setting, an aspect is simply a set of transformations specifying how code should be transformed at join points. The order of declaration of transformations is not relevant and transformations can be applied in any order. A fundamental question is whether the transformations are semantics-preserving or not. We believe that restricting ourselves to semantics-preserving transformations would be too strong a limitation. The class of expressible aspects would boil down to optimization aspects. On the other hand, transformations which are not semantics-preserving may be much too general because it is absolutely crucial to keep control over the semantics of woven programs. Each aspect language has to include appropriate restrictions on the transformations.

*Generic weaving.* The generic aspect weaver is defined in this setting using repeated applications of program transformations to the component program until a fixpoint is reached. The weaver is therefore parameterized with a component program $\mathcal{P}$ and a set of transformations $\mathcal{T}$:

$$\text{Weaver}(\mathcal{P}, \mathcal{T}) \;=\; \text{if } \exists \tau \in \mathcal{T} \,:\, \tau(\mathcal{P}) \not\equiv \mathcal{P} \text{ then Weaver}(\tau(\mathcal{P}), \mathcal{T}) \text{ else } \mathcal{P} \quad (1)$$

This definition raises several interesting issues. First, in general, this definition does not describe a terminating algorithm because of the fixpoint computation. So, one has to make sure that the rewriting system specified by the program transformations is terminating. While this problem is undecidable in general, it is often trivial to solve for practically relevant transformations. A second problem arises from transformations which are not semantics-preserving. Since no application order is specified, two different weavings of the same component program and aspects may lead to programs whose semantics differ. In some cases, this might be acceptable. Otherwise, one would also have to make sure that the rewriting system is confluent.

## 3 Integrating program analyses

*Property-based aspects.* In many cases, purely syntactic criteria are not completely satisfactory to define aspects. As an illustration, let us consider a specific aspect dealing with program robustness. Intuitively, such an aspect specifies invariants which must be verified by a program. After weaving, the program either respects the invariants or invokes an exception. For instance, if the invariant to

check is $V \leq 5$ a naive solution would be to insert the statement **if** $V > 5$ **then** `error` after all assignments to $V$. But there is no point in generating such a test after the assignment $V := V - 1$. We would like to check the invariant only when it may be violated. This means, we need a way to define and use semantic criteria in aspects. This is achieved by extending the syntax of aspect-defining transformations as follows

$$pat \implies \text{if } Prop \text{ then } T_1 \text{ else } T_2 \tag{2}$$

where *Prop* is a property of the component program defined using its standard semantics. Intuitively, this can be read "for each part of the component program matching *pat*, if *Prop* can be proven then produce the tree $T_1$ else produce $T_2$". Assuming an axiomatic semantics, an example of a transformation (which can be implemented using a local analysis) is

$$pat \implies \text{if } \{V \leq 5\}\, V := E\, \{V \leq 5\} \text{ then } V := E \text{ else } V := E;\ \textbf{if } V > 5 \textbf{ then } \texttt{error;}$$

which avoids inserting tests when the invariant holds after the assignment assuming that it holds before. Note that we could achieve even better results with a global analysis. In this case, inserted tests augment the precision of the analysis because it proceeds on the transformed programs.

Since we consider only static and automatic weaving, the properties occurring in aspects are meant to be inferred by a static analyzer. Thus, we can only expect safe approximations of these properties. Furthermore, one is not supposed to have any knowledge about the precision of the analyses. In order to have control on the semantics of the produced programs, it is important to enforce that each transformation of the form (2) satisfies the following semantic equality

$$Prop \ \Rightarrow\ [\![T_1]\!] = [\![T_2]\!]$$

In the case of semantics-preserving transformations this condition trivially holds. Otherwise, the condition ensures that the precision of the analyzer cannot have any impact on the meaning of the woven program. Indeed, if *Prop* does not hold then the analyzer will not be able to infer it (it infers only safe approximations) and $T_2$ will be produced otherwise $T_1$ and $T_2$ are semantically equivalent and the result of the analyzer does not semantically matter. Thus, the properties are best seen as filters to optimize weaving. In our previous example, it is clear that

$$\{V \leq 5\}\, V := E\, \{V \leq 5\} \ \Rightarrow\ [\![V := E]\!] \ =\ [\![V := E;\ \textbf{if } V > 5 \textbf{ then } \texttt{error;}]\!]$$

*Generic weaving of property-based aspects.* Since we are interested in a generic description of aspects and the aspect weaver, we need a framework allowing the definition of the component language semantics, the description of properties and the derivation of static analyzers. The automatic derivation of an analyzer from a semantics and a property is still an open research issue. At the moment, we are only working on a common formulation for different analyzers. The weaver remains essentially the same as defined in (1) but each application

of a transformation may require a program analysis to be performed. In general, properties or transformations can be global so the component program must be re-analyzed after each transformation. In the common case of local properties and transformations, a one-pass analyzer can be integrated into the weaver.

*Hypotheses.* The usability of the approach as described hitherto may depend too much on the analyses. For example, the aspect of robustness described below would not be realistic without program analysis. This does not quite fit the spirit of AOP (i.e. "no smart compilers"). We address this problem by extending the language of aspects with so-called hypotheses. An hypothesis is of the form $pat \Longrightarrow! \mathrm{Prop}$. It is not checked by the analyzer but integrated as a new piece of information. Through hypotheses, the user can help and control the analyzer. Of course, false hypotheses may lead to unexpected results but they are at least documented and the user has explicitly acknowledged her or his responsibility.

## 4 Conclusion

Until now, aspects have always been described and implemented in a rather ad hoc way. Here, we have sketched a generic framework based on program transformation and analysis which accommodates a large class of aspects. It is generic with respect to the component programming language: different languages can be incorporated by changing the abstract syntax. Once the syntax is described, the framework provides a pattern-based language to describe aspects and a generic weaver. Aspects can refer to semantic properties of the component program. In order to implement property-based aspects the framework provides a common format to express static analyzers.

At the moment, the main weakness of the framework is semantic. When transformations which are not semantics-preserving are to be taken into account, the framework does not provide much help to reason about the semantics of weaving. For this reason, the framework does not come close to a theoretical foundation at the moment. However, it does provide useful tools as well as simple answers to the questions asked in the introduction.

In the near future, we intend to complete the description and formalization of the framework. We see robustness and exceptions as a paradigmatic example of an aspect. They are largely independent from the component program but their introduction crosscuts large parts of it. We are designing a comprehensive aspect of robustness and plan to implement it for a small imperative language.

## References


[AOP97] K. Mens, C. Lopes, B. Tekinerdogan, G. Kiczales. "*Aspect-Oriented Programming Workshop Report*", 1st Int. Workshop on AOP, ECOOP, 1997.

[HS93] R. Heckmann, G. Sander: *"TrafoLa-H Reference Manual"*, LNCS 680, ch. 8, 1993.

[Kic+97b] G. Kiczales et al.: "*Aspect-Oriented Programming"*, collection of technical reports no. SPL-97-007 – 010, Xerox Palo Alto Research Center, 1997.