AOP: towards a generic framework using program transformation and analysis

Pascal Fradet and Mario Südholt
IRISA / INRIA-Rennes
Campus de Beaulieu, 35042 Rennes cédex, France
fradet@irisa.fr

École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantex cédex 3, France
sudholt@emn.fr

Abstract

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? We address these questions for a particular but quite general class of aspects: aspects which can be described as static, source-to-source program transformations. We discuss the design of a generic framework to express aspects as syntactic transformations as well as a generic weaver. We also consider how to use semantic properties for the definition of aspects and how to implement these properties using static analysis techniques. As an application of the framework, we sketch how to describe and implement an aspect dealing with program robustness and exceptions.

1 Introduction

Aspect-oriented programming (AOP) [Kic+97a] can be seen as a new principle to structure software — but a principle which lacks a model[1]. Several convincing case studies of AOP have been presented [Kic+97b], which suggest that AOP is a principle with great potential. However, the questions raised in the 1st workshop on AOP [AOP97] indicate that models are needed: 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. We consider only aspects described as static, source-to-source program transformations. That is, an aspect is defined as a collection of program transformations acting on the syntax tree of the component program. Of course, this class rules out some aspects one might think of. However, there is a significant number of interesting aspects that perfectly fit this setting. For example, efficiency concerns expressed as program optimizations and program robustness belong to this class. 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 abstract syntax tree 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, however, 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.

Our mid-term goal is to provide a generic framework in which one can describe the syntax and semantics of component programs. Furthermore, the framework should provide aspect languages, a generic weaver and means to define analyzers in a standard form. One of our main concerns is the maintenance control over the resulting (woven) programs. In particular, weaving must be predictable. This raises several questions about the semantics (termination, convergence) of weaving.

In order to apply the framework, we study a specific aspect dealing with program robustness. Intuitively, such an aspect specifies invariants which must be verified by a program (e.g. "the

[1] We use these terms in the spirit of G. Berry [Ber98], that is to say: "a *principle* is a foundational statement in common language, easy to understand and with far-reaching consequences" whereas "a *model* is a formal framework in which a theory is developed according to some principles".

variable V must always be less than 5 before each loop"). After weaving, the program either respects the invariants or invokes an exception. 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. This problem can be easily described in terms of program transformations but requires static analysis techniques to produce efficient programs.

In section 2, we sketch the framework allowing one to express aspects as program transformations. Section 3 is devoted to the integration of semantic criteria into aspects and program analysis into the aspect weaver. We describe in section 4 the application of the framework to the aspect of program robustness. Due to space concerns and the fact that large parts of this material are still work in progress, the paper is mostly an informal discussion.

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. Transformations are formulated on the basis of a framework which 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 can then be seen and manipulated as a tree. Defining aspects for an imperative component language, for instance, can be done on the basis of the following abstract syntax definition (As common in transformational approaches, transformations and examples are represented using the concrete syntax.):

type *stmt*	::=	Seq *stmt stmt*	statement sequence
	|	Ass *var expr*	assignment
	|	If *expr stmt stmt*	conditional
	|	While *expr stmt*	loop

A transformation is just a function which takes the tree representing the component program and yields 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 that has been designed to define tree transformations using powerful patterns. A transformation in TrafoLa-H is of the form $pat \Longrightarrow \mathit{TreeExpr}$. 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.

Patterns can be non-linear, they can be combined with boolean operators (e.g. $pat \,\&\, pat$ matches a value iff the value matches both patterns), and there are wildcards matching each tree construct (e.g. assuming an imperative language syntax, there is a wildcard matching only statements and another one matching only expressions). A very powerful pattern operator is the extraction operator '↑'. When a tree v is matched against $pat_1 \uparrow pat_2$, v is cut in all possible ways into two values (v_1, v_2) such that each v_i, $i \in \{1, 2\}$, matches pat_i and the insertion of v_2 into v_1 (which has a hole) yields v. For example, $v \uparrow (x : (\mathrm{Ass}\, V\, e))$ will extract (and bind to x) all the assignments $V := e$ from the program it is matched against. We refer the reader to the TrafoLa-H reference manual [HS93] for further details.

Aspects. These features 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", "the first assignment of the variable x" or "all calls to the function f", can be described succinctely. 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; this would rule out, in particular, the aspect of robustness. 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. A possible way to tackle this problem is to define an order on programs and to ensure that each transformation respects this order. We leave this difficult question as a future research topic.

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 $\mathcal{T}$:

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

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 very difficult (actually undecidable) in general, it is often trivial to solve for practically relevant transformations. For many aspects (e.g. aspects for optimizations), it is usually easy to devise transformations which enforce termination because they cannot be applied repeatedly to the same piece of code.

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

Using properties in aspects. In many cases, purely syntactic criteria are not completely satisfactory to define aspects. The aspect of robustness is a good illustration of this point. 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. In other terms, 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

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

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. For example, assuming an axiomatic (Hoare-like) semantics, an example of a transformation is

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

which avoids inserting tests when the invariant holds after the assignment assuming that it holds before. This is a purely local analysis and we will see below that we can do better.

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

$$\mathit{Prop} \ \Rightarrow \ [\![T_1]\!] = [\![T_2]\!]$$

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 (because only safe analyses are used) 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 the transformations. 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. Abstract interpretation [CC92] provides a useful methodology to this end. The automatic derivation of an analyzer from a semantics and a property is still an open research issue. At the moment, we are working only on designing a common formulation for different analyzers. One possibility is to represent any analyzer as an abstract machine whose rules are of the form

$$(P, S) \triangleright (P', S')$$

where P and P' are properties and S and S' are stacks of trees. Initially, the stack S contains a single tree which represents the component program and P is the precondition (e.g. the smallest element in the lattice of properties). The analyzer specifies how to scan the tree and infer post-conditions.

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. For sophisticated analyses, this process is likely to be much too costly. In the common case of local properties and transformations, the analyzer can be integrated into the weaver. The idea is to apply a transformation at each step of the analyzer. An analyzer state is of the form (P, $E : S$) and indicates that P is a precondition for the subtree E. So, if there exist a transformation of the form (2) where E matches *pat* then E is transformed into T_1 if $P \Rightarrow$ *Prop* or in T_2 otherwise. When no transformation can be applied anymore, the analyzer proceeds with the transformed sub-tree (e.g. T_1 or T_2). Depending on the specific aspect under consideration, several analysis passes may be needed. There are also cases where a single pass if sufficient: for instance, when the transformations operate only on the leaves.

Making 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

$$\mathit{pat} \implies ! \mathit{Prop}$$

An hypothesis 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 Aspect of Robustness

A robust program has a standard domain (of input values) where it terminates normally (and satisfies its specifications) and an exceptional domain where it terminates by an error or exception. Usually, writing robust programs involves writing many tests scattered throughout the code. In practice, inserting the tests is difficult and results in tangled code. Hence, many programs work

only for the (unspecified) standard domain and do not meet their specifications on the exceptional domain. We believe that AOP can provide a solution to this problem.

In our approach, the component program is not robust in the sense that it satisfies its specification for the standard domain but not for the exceptional one. The aspect can be seen as the formal description of the standard (or, equivalently, the exceptional) domain. The weaving process produces a robust program equivalent to the component program for the standard domain but which results in an exception for the exceptional domain.

The most declarative approach for the definition of this notion of robustness for imperative programs is to describe the standard domain by invariants on variables. Let us consider again our small imperative component language; in this case an invariant is a boolean expression of the language involving program variables. Some examples of such constraints for integer variables are $V \leq 5$, $X = Y + Z$, $X = Y \vee X = Z$. The user specifies the standard domain as a constraint in conjunctive normal form $P_1 \wedge \ldots \wedge P_n$. This can then be translated automatically in our framework into an aspect of the form

$$pp : pat \implies \text{if } |P_i| \text{ then } pp \text{ else } (\{ \textbf{ if } !P_i \textbf{ then } \texttt{error}\}; pp) \qquad \text{for } 1 \leq i \leq n$$

where *pat* matches any program point, the match is bound to *pp* and $|P_i|$ denotes the constraint expressed in the semantic domain.

The user may also want to specify properties for specific parts of the component program. In this case, (s)he must write them directly as transformations. For example, in order to check divisions by zero, one can write

$$pp : pat(e_1/e_2) \implies \text{if } |e_2 \,!= 0| \text{ then } pp \text{ else } (\ \{\textbf{if } e_2 = 0 \textbf{ then } \texttt{error} \text{ “division by zero”}\}; pp\ )$$

Such transformations are not semantics-preserving. On equal inputs the source program may terminate normally whereas the transformed program reports an error. The semantics of the woven program is well defined and clear, however. The domain for which the two programs differ is explicitly and clearly described as part of the aspect definition. It would certainly be more difficult to reason directly on the tangled code.

It is easy to ensure that a set of such program transformations is terminating and that a single pass is sufficient to perform the required program analysis. One only has to make sure that inserted tests are not considered as join points by patterns. After one pass, the properties specified in the aspect will hold at each program point (the program point is either preceded by tests ensuring the properties or they have been proved to hold already at this point). Note that the rewriting system is not confluent. Depending on the transformation strategy, the resulting programs may include different tests in different order. However, all these syntactically different results are semantically equivalent.

For constraints on numerical variables, a standard analysis [CH78] can be used. This analysis infers approximate linear relations between variables using an abstract domain based on polyhedras. It is strictly more precise than interval-based analyses. The extension to boolean variables is straightforward. Pointer variables introduce new problems but we can integrate them into the same generic framework using other analysis techniques such as those described in [FGL96, SRW97].

It is important to note that we are in a different situation than traditional static analysis which sometimes fails to yield any useful information at all. Here, when an invariant cannot be proved to hold, the program is transformed by including a test. But the analysis proceeds on the transformed program, which will be analyzed on the basis of the information that the invariant holds after the newly introduced test. That is, each inserted test provides new information to the analyzer. Hence, we believe that many spurious tests will be filtered out by the analyzers without having recourse to hypotheses.

For now, we have only considered the insertion of basic exceptions which terminate the program immediately. We plan to look at more sophisticated techniques of exception handling (e.g. exceptions with retries). This must be done with care, however, because the integration of exceptions must be controlled tightly. In general, exceptions can be used as a mechanism to arbitrarily change the control flow of programs. The impact of general exceptions on the functionality of the component program after weaving can therefore quickly become uncontrollable.

5 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 define 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 any help (laws , tools, ...) to reason about the semantics of weaving. We have not taken any clear-cut decision about these semantic issues, yet. It is clear that transformations which are not semantics-preserving must be strongly restricted. Otherwise, one would quickly lose control because the generic weaver applies transformations in an indeterminate order. For these reasons, the framework does not come close to a model or a theoretical foundation at the moment. However, it does give simple answers to the questions asked in the introduction. Aspects are expressed as a set of program transformations in a general purpose language. Join points are nodes of an abstract syntax tree defined by explicit selection using patterns as part of aspects. The weaver, finally, is a fixpoint operator applying the transformations to the component program.

In the near future, we intend to complete the description and formalization of the framework. The aspect of robustness is a non-trivial example, which is interesting in itself. We plan to implement both for a small imperative language.

References

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

[Ber98] G. Berry. "*>From principles to programming languages*", invited talk at POPL, 1998.

[CC92] P. Cousot, R. Cousot: *"Abstract Interpretation and Logic Programming"*, Journal of Functional Programming, 13(2-3), 1992.

[CH78] P. Cousot, N. Halbwachs: "*Automatic discovery of linear restraints among variables of a program*", POPL, 1978.

[FGL96] P. Fradet, R. Gaugne, D. Le Métayer. " *Static detection of pointers error: an axiomatisation and a checking algorithm*", LNCS 1058, pp. 125-140, ESOP, 1996.

[HS93] R. Heckmann, G. Sander: *"TrafoLa-H Reference Manual"*, in: B. Hoffmann, B. Krieg-Brückner (eds.): *"Program Development by Specification and Transformation. The PROSPECTRA Methodology, Language Family, and System"*, LNCS 680, ch. 8, 1993.

[Kic+97a] G. Kiczales et al.: "*Aspect-Oriented Programming*", Special Issues in Object-Oriented Programming, Max Muehlhaeuser (editor) et al, dpunkt Heidelberg, 1997.

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

[SRW97] S. Sagiv, T. Reps, R. Wilhelm. "*Solving shape-analysis problems in languages with destructive updating*", ACM Transactions of Programming Languages and Systems, 1997.