FOAL '08

Proceedings of the Seventh Workshop on Foundations of Aspect-Oriented Languages

held at the
Seventh International Conference on
Aspect-Oriented Software Development

April 1, Brussels Belgium

Workshop Organizers: Curtis Clifton, Shmuel Katz, Gary T. Leavens, and Mira Mezini

ACM International Conference Proceedings Series
ACM Press

The Association for Computing Machinery
1515 Broadway
New York New York 10036

ACM ISBN: 978-1-60558-110-1

Contents

Preface

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The seventh Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Seventh International Conference on Aspect-Oriented Software Development in Brussels Beligum, on April 1, 2008. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, and theory of aspect translation (compilation) and rewriting. The call for papers welcomed all theoretical and foundational studies of foundations of aspect-oriented languages.

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest within the programming language theory and types communities in aspect-oriented programming languages.
- Foster interest within the formal methods community in aspect-oriented programming and the problems of reasoning about aspect-oriented programs.

FOAL logos courtesy of Luca Cardelli

The workshop was organized by Curtis Clifton (Rose-Hulman Institute of Technology, USA), Shmuel Katz (Technion–Israel Institute of Technology, Israel), Gary T. Leavens (University of Central Florida, USA), and Mira Mezini (Darmstadt University of Technology, Germany). The program committee was chaired by Curtis Clifton.

We thank the organizers of AOSD 2008 for hosting the workshop, and in particular Workshops Chairperson Mario Sudholt for his help.

Message from the Program Committee Chair

The seventh FOAL workshop maintains the high bar for quality set by previous instances. FOAL is one of the primary forums for foundational work on aspect-oriented software development. As in the past, each paper was subjected to full review by at least three reviewers. Papers co-authored by program committee members or organizers were reviewed by four or five reviewers and were held to a higher standard. I am grateful to the program committee members for their dedication, insightful comments, attention to detail, and the service they provided to the community and the individual authors.

The members of the program committee were: Jonathan Aldrich (Carnegie Mellon University, USA), Lodewijk Bergmans (University of Twente, Netherlands), Curtis Clifton (Rose-Hulman Institute of Technology, USA), William Griswold (University of California, San Diego, USA), Günter Kniesel (University of Bonn, Germany), Shriram Krishnamurthi (Brown University, USA), Ralf Lämmel (Universität Koblenz-Landau, Germany), Karl Lieberherr (Northeastern University, USA), Hidehiko Masuhara (University of Tokyo, Japan), James Noble (Victoria University of Wellington, New Zealand), Klaus Ostermann (University of Aarhus, Denmark), Hridesh Rajan (Iowa State University, USA), and Damien Sereni (Oxford, UK). The sub-reviewers, whom I also thank, were: Bryan Chadwick, Robert Dyer, and Therapon Skotiniotis.

I am also grateful to the authors of submitted works. Ten papers were submitted for review this year. Of these, the program committee selected four for presentation at the workshop and publication in the proceedings. The committee judged that a fifth paper presenting preliminary work was intriguing and invited the authors to give a short presentation. An abstract outlining that work is also published herein.

The program was rounded out with an invited talk by William Harrison of Trinity College, Dublin, Ireland, and a panel discussion on the state of research in AOSD. The panelists were Mehmet Aksit (University of Twente, Netherlands), Oege de Moor (Oxford University, UK), Gary T. Leavens (University of Central Florida, USA), Klaus Ostermann (University of Aarhus, Denmark). I sincerely thank them all for their dedication and service.

Finally, I would like to thank the other members of the organizing committee of FOAL—Shmuel Katz, Gary Leavens, and Mira Mezini—for their work in guiding us toward another inspiring workshop.

Curt Clifton
FOAL ‘08 Program Chair
Rose-Hulman Institute of Technology
Terre Haute, Indiana, USA

Redundancy-free Residual Dispatch

Using Ordered Binary Decision Diagrams for Efficient Dispatch

Andreas Sewe Christoph Bockisch Mira Mezini

Technische Universität Darmstadt
Hochschulstr. 10, 64289 Darmstadt, Germany
{sewe, bockisch, mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

State-of-the-art implementations of common aspect-oriented languages weave residual dispatching logic for advice whose applicability cannot be determined at compile-time. But being derived from the residue's formula representation the woven code often implements an evaluation strategy which mandates redundant evaluations of atomic pointcuts. In order to improve upon the average-case run-time cost, this paper presents an alternative representation which enables efficient residual dispatch, namely ordered binary decision diagrams. In particular, this representation facilitates the complete elimination of redundant evaluations across all pointcuts sharing a join point shadow.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms

Languages, Performance

Keywords

Advice, aspect-oriented programming, dispatch functions, ordered binary decision diagrams, pointcuts, residual dispatch

1. INTRODUCTION

This paper presents an approach for optimizing dispatch in aspect-oriented languages of the pointcut-and-advice (PA) flavor [18]. In this flavor, of which the AspectJ language [17] is the most prominent example, aspects encompass two kinds of constructs: *pointcuts* and *advice*. While advice define the actions to be performed whenever the program is in a certain state,[1] called a *join point*, their associated pointcuts define predicates on join points, thereby associating states with the advice in question.

The *pointcut language*, which is an integral part of any PA flavor language, is typically based on propositional logic or some extension thereof. Its *atomic pointcuts* designate, e.g., a **call** to a specific method and are subsequently composed by propositional operators to form more complex pointcuts. Furthermore, they generally are parameterized; such atomic pointcuts are satisfied if and only if the program's state allows for a satisfying parameter binding [23]. On the whole, these constructs are the foundation more elaborate pointcut languages based on, e.g., temporal logics can rest upon [10].

1.1 Residues and Dispatch Functions

The semantics outlined above are typically implemented in two steps [16]: *join point shadow matching* and *weaving*. First, the matching step identifies all parts of the program, called *join point shadows*, whose execution may result in a join point satisfying a given pointcut. Then, the weaving step inserts code at the join point shadows performing the actions defined by each advice. But as compilers may be unable to statically determine whether an atomic pointcut is satisfied or not at a join point shadow [3], residual dispatching logic has to be woven into the program's code. This *residue* is the result of the pointcut's partial evaluation [19].

In this setting, residual dispatch at a join point shadow can be viewed as the evaluation of a finitary Boolean function $f_\phi : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, the so-called *dispatch function* [9]; whether an advice is applicable depends solely on the prior evaluation of the n atomic pointcuts occurring in ϕ, the residue in question. Residual dispatch thus bears close resemblance to predicate dispatch [12], even though the function's range is restricted to two outcomes: an advice is either applicable or not. But this restriction is not inherent in residual dispatch; in fact, this paper extends the notion of dispatch functions to the simultaneous evaluation of all residues $\phi \in \Phi$ sharing a join point shadow. The dispatch function hence becomes $f_\Phi : \mathbb{B}^{\tilde{n}} \to \mathbb{B}^m$ and characterizes the subset of advice applicable at the join point; which combination of the $m = |\Phi|$ advice is executed depends on the state of the $\tilde{n}$ atomic pointcuts jointly occurring in Φ.

Both cases are illustrated by Figure 1, which depicts the residual dispatching logic woven for the following two advice.

[1] For uniformity of presentation, events in the program's execution, as identified, e.g., by AspectJ's **call** atomic pointcut, are treated as part of the program's state.

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

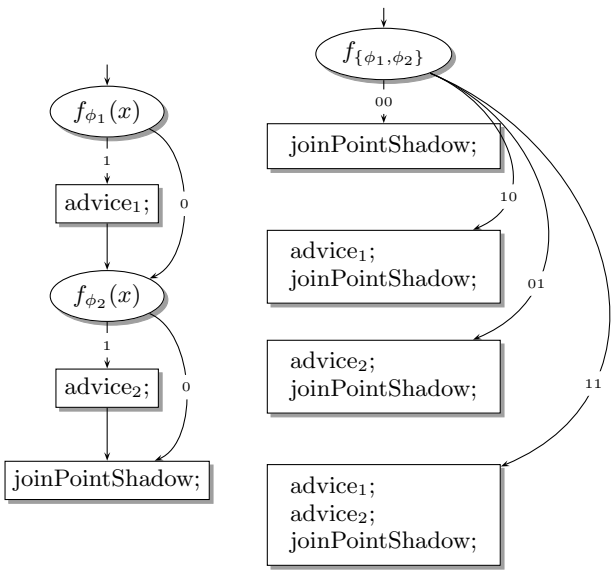

Figure 1: Two implementation schemes employing dispatch functions for residual dispatch at a join point shadow.

Hereby, the two associated pointcuts match the same join point shadow but give rise to two residues, namely ϕ_1 and ϕ_2, which are—in general—different, although there may of be identical atomic pointcuts occurring in both.

before() : joinPointShadow && ϕ_1 { advice$_1$; }
before() : joinPointShadow && ϕ_2 { advice$_2$; }

The first of the above implementation schemes resorts to two distinct dispatch functions, f_{ϕ_1} and f_{ϕ_2}, which is the approach followed by current weavers, whereas the second employs just a single dispatch function: $f_{\{\phi_1,\phi_2\}}$. Here, in a state x and based on the value of $f_{\{\phi_1,\phi_2\}}(x) \in \mathbb{B}^2$, some combination of advice is executed. But although the above suggests that the weaver generates code for each combination of advice, this need not be the case in practice (cf. Sections 4, 5.1).

1.2 Efficient Evaluation Strategies

In either case dispatch functions open up the possibility for *redundancy-free* residual dispatch. In particular, each of the n arguments of a dispatch function f_ϕ need not be evaluated more than once. By extension, any of the $\tilde{n}$ arguments of f_Φ, each of which may be shared by several residues, has to be evaluated at most once. This possibility hinges on a few assumptions on residual dispatch, though. But these assumptions typically hold for PA flavor languages in general and AspectJ in particular (cf. Section 2).

As each atomic pointcut's evaluation incurs a certain run-time cost, which varies depending on the kind of pointcut [3], it is of interest to find an *evaluation strategy* minimizing the overall cost of evaluation. Hereby, the average-case cost of evaluating $f_\Phi(x)$ for all states $x \in \mathbb{B}^{\tilde{n}}$ is most relevant to efficient dispatch, as it determines the run-time cost incurred in the long run. In contrast, the worst-case cost with respect to all states merely determines an upper bound.

Yet, regardless of the precise notion of optimality employed, the problem of finding an optimal strategy is NP-hard, as the Boolean satisfiability problem can be reduced to it.[2] Consequently, this optimization problem is usually approached heuristically. It is, e.g., often advantageous to evaluate those atomic pointcuts first whose evaluation incurs the least run-time cost. In addition to such heuristics there is a fundamental method which ensures improvement with respect to run-time costs: the elimination of redundant evaluations—ideally across residues.

1.3 Contributions and Structure of this Paper

This paper presents an approach which performs complete redundancy elimination across all pointcuts sharing a join point shadow. It furthermore makes use of the aforementioned heuristic by incorporating an ordering based on the cost of the atomic pointcuts' evaluation. To enable these optimizations, the approach employs an alternative representation of Boolean functions, namely ordered binary decision diagrams [8].

The remainder of this paper is organized as follows. First, Section 2 states the assumptions made on advice dispatch to enable the simultaneous evaluation of multiple residues. Then, Section 3 discusses advantages and disadvantages of the functions' traditional formula representation. Section 4 presents ordered binary decision diagrams as an alternative representation. Thereafter, Section 5 assesses both representations based on implementation experience and experimental results. Finally, Section 6 discusses related work, while Section 7 concludes this paper and suggests areas for future work.

2. ASSUMPTIONS ON ADVICE DISPATCH

The simultaneous evaluation of multiple residues is made possible by the three assumptions on advice dispatch stated below.

- The evaluation of an atomic pointcut is side-effect free.
- The binding of parameters is no side-effect of an atomic pointcut's evaluation.
- The execution of an advice does not affect the evaluation of a pointcut associated with another advice.

The first and second assumption make it possible to evaluate the atomic pointcuts occurring in a single residue in any order. The third assumption extends this possibility to multiple residues. Together, these assumptions allow for a clean separation between the residues' evaluation on the one hand and the advice's execution on the other hand.

The above assumptions impose only moderate restrictions on PA programs. In particular, they hold for most—if not all—programs written in AspectJ. Violations of the first assumption, although prohibited by neither *ajc* [16, 17] nor *abc* [4], the two principal compilers for the AspectJ language, are strongly advised against in the language's *Programming Guide* [2], since the order of evaluation of atomic pointcuts is implementation-specific. The second assumption is even actively enforced by *ajc* as the AspectJ language disallows ambiguous parameter bindings [23]. It should be noted, however, that *abc* resolves these ambiguities consistently rather

[2]If a non-tautological formula ϕ were satisfiable, i.e., if $\exists x \in \mathbb{B}^n . f_\phi(x) = 1$, then at least one argument of f_ϕ would have to be evaluated by an optimal strategy.

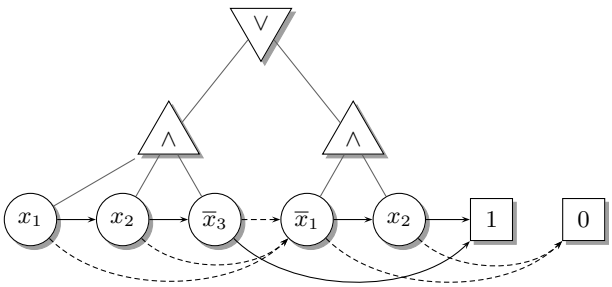

Figure 2: A formula (in DNF) and an evaluation strategy for $(x_1 \wedge x_2 \wedge \overline{x}_3) \vee (\overline{x}_1 \wedge x_2)$.

than disallowing them outright [4]; here, parameter binding becomes a side-effect of evaluation.

In contrast to the first two assumptions, which carry over from predicate dispatch [12], the third assumption is specific to advice dispatch and may indeed be violated by legal AspectJ programs: a pointcut associated with one advice can observe, by means of an **if** atomic pointcut, some state affected by another advice. The latter thus becomes what is known as narrowing advice [21]; whether the former is executed depends on the execution of the latter. But this advice-on-advice interaction is subtle, error-prone, and presumably not often used. Still, if it is used, the weaver has to resort to multiple dispatch functions instead of a joint one (cf. Section 7). Although this precludes redundancy elimination across residues, each dispatch function's evaluation strategy may still be freed from redundancies.

3. PROPOSITIONAL FORMULAS

As pointcuts are typically specified using a language akin to propositional logic, propositional formulas are a representation of considerable import; at the very least they will serve as the dispatch functions' intermediate representation. But formulas do not, by themselves, give rise to an evaluation strategy. In most programming languages their evaluation is therefore governed by a set of rules. The Java Language Specification [15], e.g., mandates left-to-right short-circuit evaluation. In contrast, AspectJ, despite its evocative use of Java's short-circuiting && and || operators, does not prescribe any particular order of evaluation. But since all atomic pointcuts are required to be side-effect free (cf. Section 2), this is not a problem but an asset; it allows for evaluation strategies optimized not only by reordering the atomic pointcuts' evaluations but also by removing redundant atomic pointcuts outright.

Still, the evaluation strategies chosen by both *ajc 1.5.4* and *abc 1.2.1* are ultimately based on the left-to-right short-circuit evaluation of a formula. Figure 2 exemplifies this. The resulting strategy is hereby depicted as an if-then-else straight-line program. By convention, the then- or 1-edges are drawn solid, whereas the else- or 0-edges are drawn dashed. If, e.g., the three atomic pointcuts are in state $x = (1,1,1)^\intercal \in \mathbb{B}^3$, then computation proceeds along the path $\langle x_1 \rightarrow x_2 \rightarrow \overline{x}_3 \dashrightarrow \overline{x}_1 \dashrightarrow 0\rangle$; hereby, the second evaluation of the atomic pointcut x_1, in the literal[3] $\overline{x}_1$, is redundant when it comes to computing the value $f(x) = 0$.

[3]A literal is either an atomic pointcut or its negation.

3.1 Redundancy Elimination

Both compilers do not directly derive evaluation strategies from a residue; instead, the strategies chosen revolve around a refined formula representation thereof: the original formula ϕ is brought into disjunctive normal form (DNF). Given this representation, either compiler generates code that performs short-circuit evaluation of the normalized formula. In addition, *ajc* makes use of the DNF representation to eliminate some redundancies by applying two laws of Boolean algebra: idempotence and boundedness. The compiler furthermore performs a minor optimization by reordering the literals in each conjunct in order of increasing runtime cost. Similarly, the conjuncts themselves are ordered. But, as Figure 2 illustrates, being limited to a two-level formula representation like DNF makes it frequently impossible to eliminate redundant evaluations of atomic pointcuts.

Some of these can, however, be eliminated after the dispatching logic has been generated by the weaver for the chosen evaluation strategy. Compilers which perform data-flow analyses like common subexpression elimination [1] can, e.g., eliminate the second evaluation of x_1 in Figure 2, which is redundant, as on all paths leading to the corresponding vertex the value of this common subexpression has already been computed. In contrast, the second evaluation of x_2 cannot avoided, as there is a path $\langle x_1 \dashrightarrow \overline{x}_1 \rightarrow x_2\rangle$ on which the value of x_2 has not previously been computed.

3.2 Extended Propositional Operations

As propositional formulas have traditionally been used to represent functions $f_{\phi_i} : \mathbb{B}^n \rightarrow \mathbb{B}$ only, AspectJ compilers generate dispatching logic that evaluates, for $i = 1, \ldots, m$, one residue ϕ_i after the other. But Boolean logic allows for a straight-forward extension to formulas which can cover the class of functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ for arbitrary m. Using this extension the m formulas $\phi_1, \ldots, \phi_m$ can be encoded into one. Hereby, the truth value of variables $x_1, \ldots, x_n$ is the same across all component formulas, i.e., each variable $x_1, \ldots, x_n$ still evaluates to either $\bot = (0, \ldots, 0)^\intercal \in \mathbb{B}^m$ or $\top = (1, \ldots, 1)^\intercal \in \mathbb{B}^m$. The propositional operators, however, are extended to operate component-wise on $\mathbb{B}^m$. To facilitate projections onto a single component the extension is also enriched by m constants $e_1, \ldots, e_m$ evaluating to truth values other than $\bot$ and $\top$, namely to the Boolean atoms $(1, 0, \ldots, 0)^\intercal, \ldots, (0, \ldots, 0, 1)^\intercal \in \mathbb{B}^m$. Disjunctions thereof thus cover the entire range of $\mathbb{B}^m$.

Considering the above extension, let there be m residues, represented by formulas $\phi_1, \ldots, \phi_m$ whose signatures jointly encompass the variables $x_1, \ldots, x_n$ satisfiable at a single join point shadow. Then the joint dispatch function $f_\Phi : \mathbb{B}^n \rightarrow \mathbb{B}^m$, or rather a formula representation thereof, is given by $\bigvee_{i=1}^m e_i \wedge \phi_i$. Conceptually, each residue ϕ_i is first evaluated separately. The result is then projected onto the i-th component, before all intermediate results are combined by means of disjunction. This is exemplified by Figure 3, which depicts such an extended formula with its constants distributed downwards to the level of literals. For a state $x = (0,0,0,0)^\intercal \in \mathbb{B}^4$, e.g., the subformula on the left evaluates to $(0,0)^\intercal$ whereas the subformula on the right evaluates to $(0,1)^\intercal$, which consequently is the value of $f_\Phi(x)$.

Unfortunately, this extended representation does not allow for short-circuit evaluation; while for a range of $\mathbb{B}$ the use of if-then-else instructions was sufficient for implementing evaluation strategies, code generated for the extended range

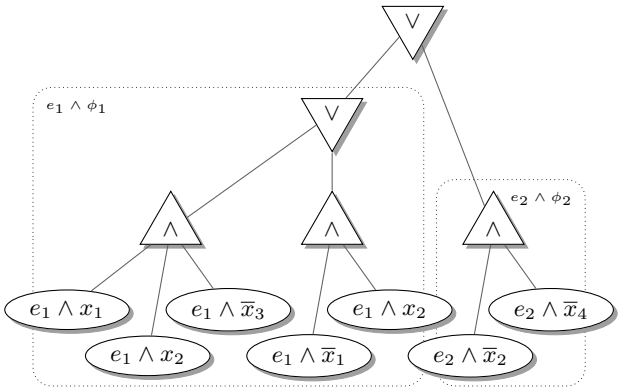

Figure 3: An extended formula, together with its two component formulas $\phi_1 = (x_1 \wedge x_2 \wedge \overline{x}_3) \vee (\overline{x}_1 \wedge x_2)$ and $\phi_2 = \overline{x}_2 \wedge \overline{x}_4$.

of $\mathbb{B}^m$ needs to evaluate the actual conjunctions and disjunctions. Nevertheless, propositional formulas are a useful representation. Not only are they the representation pointcuts are typically specified in, but they also allow for an elegant description of a pointcut's addition or removal: given an extended formula representation, the addition of another advice and its associated pointcut ϕ_{m+1} can be described simply in terms of disjunction. Similarly, their removal can be described in terms of conjunction and negation.

4. BINARY DECISION DIAGRAMS

Like formulas, binary decision diagrams (BDDs) are a natural representation of Boolean functions with a long history. But unlike the former, the latter make evaluation strategies explicit. They are thus of particular interest when such strategies are sought, as is the case when optimizing residual dispatch. Informally, a BDD is a rooted, directed, acyclic graph (DAG) built up from two kinds of vertices: *splits* and *sinks*. Hereby, all splits are labeled with variables $x_1, \ldots, x_n$ and have two outbound edges, labeled 0 and 1, respectively. Similarly, the sinks are labeled with values from a set $\mathbb{B}^m$ but do not have outbound edges.

Of the several equivalent definitions of a BDD's semantics [24], one closely reflects the intuition of a control-flow from source to sink: given a state $x \in \mathbb{B}^n$, computation begins at the root or *source* of the BDD G. At a vertex labeled x_i it then proceeds along the *low* or 0-edge if $x_i = 0$ and along the *high* or 1-edge otherwise. It finally stops, when a sink is reached; $f_G(x) \in \mathbb{B}^m$ is the value this sink is labeled with. This top-down approach to evaluation immediately gives rise to an evaluation strategy. Code generation is also straight-forward. Figure 4 exemplifies this by depicting such a strategy and thus the BDD itself as an if-then-else straight-line program. Note in particular that this BDD is *decision equivalent* to the extended formula of Figure 3; it represents the same function $f : \mathbb{B}^4 \to \mathbb{B}^2$, i.e., for every state $x \in \mathbb{B}^4$ evaluation of either representation results in the same value $f(x)$. But, as exposed by the BDD representation, ϕ_1 and ϕ_2 cannot be satisfied simultaneously, i.e., $f(x) \neq (1,1)^\intercal$; if code is generated for each combination of advice (cf. Section 1.2), then this fact may be exploited to avoid code generation for all combinations of advice.

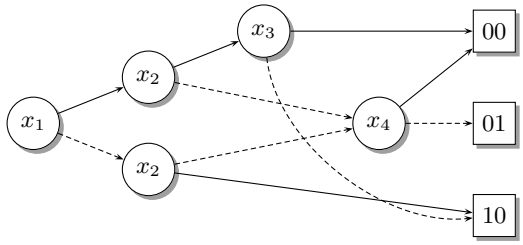

Figure 4: A BDD representation decision equivalent to the extended formula of Figure 3.

4.1 Redundancy Elimination

Since a BDD representation corresponds to an evaluation strategy, redundant evaluations of atomic pointcuts can be characterized by a simple syntactic property: the existence of paths from source to sink on which more than one split is labeled with the same variable or, equivalently, the existence of paths which are not taken for any state x [24]; BDDs without such inconsistent paths are called *free* or *read-once*.

While the read-once property is generally desirable, the conversion from unconstrained BDDs to free ones can cause an exponential blow-up in terms of size [24]. This blow-up is, however, in general no worse than that which a formula's conversion to either CNF or DNF can cause. Still, there are functions which allow for a polynomial-size normal form but which require a free BDD (FBDD) whose size is exponential in the number of variables [6]. But this is also true vice versa. Thus, the representational power of FBDDs, DNFs, and CNFs is only comparable on a case by case basis; neither representation is smaller for all functions.

Yet, there is one assumption one can reasonably make about the functions encountered during residual dispatch: they stem from a small and structured formula representation. This is because these formulas correspond to the pointcuts as written. It is hence reasonable to restrict the discussion to those formulas which are likely to form a pointcut in real-world programs. But this set of small, structured formulas is difficult to characterize. Nevertheless, such a characterization will be attempted in Section 5.2.

4.2 Propositional Operations

Given two (free) BDDs G and H, the problem of computing their conjunction or disjunction, a task known as *synthesis*, is NP-hard. (Negation, however, can trivially be applied by negating the value of all sinks.) Thus, a variety of syntactic constraints has been imposed on BDDs in order to facilitate efficient synthesis [22, 14]. The most prominent constraint [8] gives rise to a subclass of BDDs known as ordered binary decision diagrams (OBDDs): on each path from source to sink the variables are required to occur in the same order π. The OBDD shown in Figure 4, e.g., is a π-ordered representation of the extended formula of Figure 3, where $\pi = \langle x_1, x_2, x_3, x_4 \rangle$.

Obviously, every OBDD is free and hence gives rise to a redundancy-free evaluation strategy. The converse, however, is false; thus, the representational power of OBDDs is strictly smaller than that of FBDDs—although not smaller than that of either CNF or DNF (cf. Section 4.1). Nevertheless, OBDDs are of particular interest when dispatch functions are to be represented. In this setting, being con-

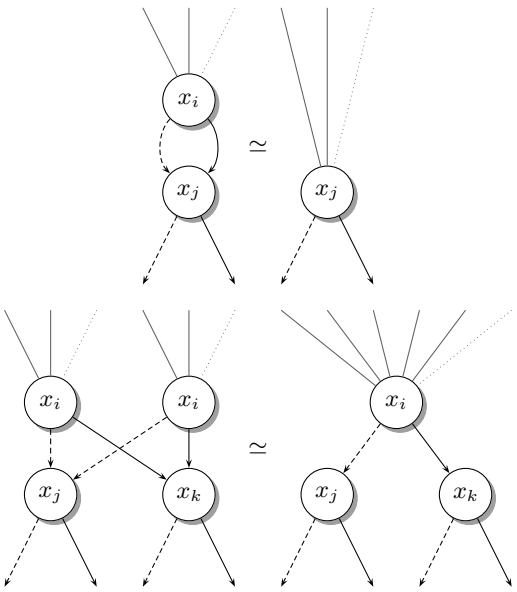

Figure 5: The *deletion rule* and the *merging rule*.

strained to a fixed variable ordering π is only a moderate impediment. In fact, ordering the atomic pointcuts simply in the order of increasing run-time cost is a heuristic which performs well in experiments (cf. Section 5.2).

Provided that both operands are π-ordered, there exists an efficient algorithm for computing the disjunction or conjunction of two OBDDs G and H [8]; employing memoization, synthesis is performed in $O(|G||H|)$. While originally devised for OBDD representations of functions $f : \mathbb{B}^n \to \mathbb{B}$, the algorithm can easily be adapted to sinks labeled with values from the larger set $\mathbb{B}^m$. It can furthermore be modified such that the result of each step is reduced, i.e., the number of vertices is minimized [7]. This is done by repeatedly applying the two *reduction rules* shown in Figure 5. Applying these rules during each synthesis step is desirable not only because it keeps intermediate results small, but also because it minimizes the size of the final OBDD.

5. ASSESSMENT

When assessing the utility of OBDD-based dispatch functions, two questions have to be answered: whether dispatch functions can be easily integrated with a compiler or an aspect-aware execution environment and whether an OBDD representation thereof can improve upon the average-case run-time cost of residual dispatch.

5.1 Implementation Experience

OBDD-based dispatch functions were implemented and integrated with an experimental execution environment for PA flavor languages developed as part of the Aspect Language Implementation Architecture project [5];[4] they supplanted the previous, DNF-based residual dispatching logic. Furthermore, dispatch functions were incorporated into the framework the environment builds on, with the abstraction completely hiding the chosen representation; whether the functions are represented by means of formulas, OBDDs, or even a combination thereof is immaterial to the framework itself. Only when the residual dispatching logic needs to be woven by an instantiation of the framework, e.g., a compiler or an execution environment, the functions' concrete representations have to be considered. The weaving approach which the default instantiation hereby follows avoids generating code for each combination of advice. Instead, it computes the joint dispatch function's value and, based on this, dispatches the applicable advice one after the other.

Overall, the changes required by the integration to both the framework and its instantiation were minimal. This fact can serve as indication that the notion of dispatch functions is a natural one. Deployment and undeployment of aspects in particular were found to be easily implementable in terms of the extended propositional operations (cf. Section 3.2).

5.2 Experimental Results

Traditionally, the complexity theory of Boolean functions has considered the class of functions $f : \mathbb{B}^n \to \mathbb{B}$ only. Consequently, any representation's expressiveness has been assessed primarily with respect to this class. When a representation suitable for residual dispatch has to be chosen, however, this assessment is of limited usefulness as all dispatch functions ultimately stem from pointcuts and their respective residues. Thus, an attempt was made to characterize the formulas likely to form residues in real-world programs: these *non-trivial* but *simple* formulas are those which are not decision equivalent to either $\bot$ or $\top$ and cannot be simplified by applying the laws of idempotence and boundedness.

Although there are, e.g., $2^{2^5} = 4,294,967,296$ Boolean functions in five arguments, there are only about 118 million non-trivial, simple formulas of signature $\langle x_1, \ldots, x_5 \rangle$ with up to six propositional operators; this set, which contains, e.g., the formula of Figure 2, is at the same time large enough to encompass most residues encountered in practice but also small enough to experiment with. It should be noted, however, that it contains various decision equivalent formulas. During the course of the experiments these were treated as distinct, for they may give rise to distinct evaluation strategies. The cost incurred by evaluating each of the five atomic pointcuts was assumed to be 1.0, 1.5, 2.0, 2.5, and 3.0, respectively, which reflects the range of relative costs observed for common atomic pointcuts like **this** and **cflow**.

Figure 6 charts the average-case costs of evaluation strategies derived from the original formulas and their DNF counterparts, respectively. Each data point hereby corresponds to numerous pairs of formula and DNF; its shade indicates how many representations give rise to a particular combination of average-case costs. While there are formulas whose DNF representation is considerably larger and thus incurs higher cost of evaluation, it is noteworthy that there are numerous formulas which benefit from conversion to this representation. The figure depicts these cases as data points above and below the bisector, respectively.

The aforementioned result is due to two causes: first, conversion allows for simplification to be applied twice; the laws of idempotence and boundedness were employed once before and once after conversion to DNF. Second, literals and conjuncts were reordered according to the run-time costs incurred by their evaluation. These two optimizations are

[4]The implementation is available to the public: `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ALIA/alia.html`.

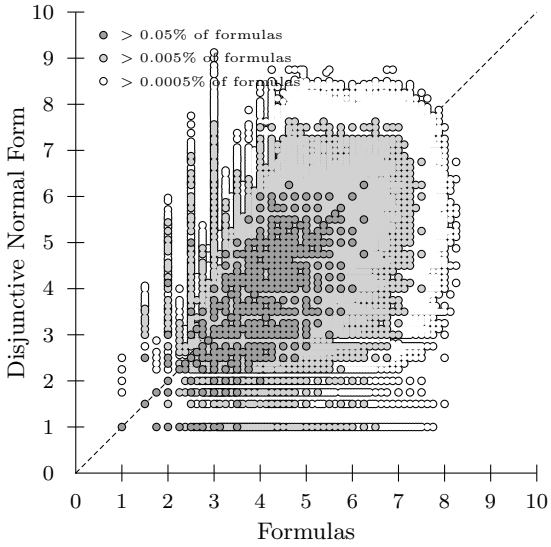

Figure 6: The average-case costs of evaluation strategies derived from formulas and their DNF.

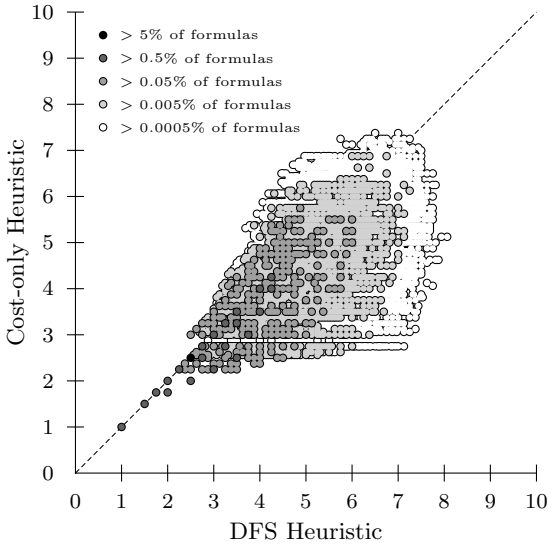

Figure 7: The average-case costs of evaluation strategies derived from OBDDs ordered with respect to two straight-forward heuristics.

precisely those used by *ajc* (cf. Section 3.1) and here have been proven effective; the evaluation cost averaged over all simple formulas considered is 4.398 for DNF representations, whereas their left-to-right short-circuit evaluation results in an average cost of 4.603.

In contrast to this marginal difference OBDD representations offer significant improvements over either formula-based representation. These improvements are only partly owed to the redundancy-free nature of OBDDs; the variable ordering chosen is also important, as illustrated by Figure 7. One of the variable ordering heuristics applied here, the so-called depth-first search (DFS) heuristic [13], derives an ordering from a depth-first traversal of the original formula. The second heuristic applied, the cost-only heuristic, derives an ordering from the costs incurred by the variables' evaluation. While the former heuristic ignores the atomic pointcuts' cost, the latter ignores the residue's structure. Both are straight-forward, but when the average-case cost of evaluation strategies is the primary concern, the cost-only heuristic was found to be superior to the DFS heuristic.

For 47.6% of the formulas considered, the cost-only heuristic gives rise to a strategy with average-case cost superior to that derived using the DFS heuristic. The latter heuristic is superior in only 18.2% of the cases. But with 3.438 and 3.721, respectively, the evaluation costs averaged over all formulas in either case are lower than that of the two formula-based representations.

6. RELATED WORK

Dispatch functions of the form $h : \mathbb{B}^n \to \{1, \ldots, m\}$ play an important role in predicate dispatch [12] and bear a close resemblance to the functions employed during advice dispatch [20]; in this setting, the n atomic predicates determine which of the m methods is ultimately executed. When the dispatch function is viewed as a composition $h = g \circ f$, with $f : \mathbb{B}^n \to \mathbb{B}^m$ and $g : \mathbb{B}^m \to \{1, \ldots, m\}$, this resemblance is most prominent. Hereby f characterizes applicability, whereas g determines the overriding relationship. This decomposition is particularly advantageous if, as suggested by the present paper, an OBDD representation is used for the former function. First, a representation of f is synthesized by applying the extended propositional operations (cf. Sections 3.2, 4.2). Then the sinks are relabeled according to g. Finally, subsequent applications of the reduction rules (cf. Figure 5) yield the reduced OBDD representing h.

Decision diagrams have been employed, under the name of *lookup DAGs*, for the efficient implementation of both multiple and predicate dispatch [9]. But these decision diagrams are not necessarily binary; this complicates synthesis. For residual dispatch, however, this complication is unnecessary as atomic pointcuts are propositional in nature. Construction of lookup DAGs is further complicated by the fact that the synthesis algorithm requires a canonical formula representation, namely DNF, to be used. In some cases, this can cause an exponential blow-up of intermediate results even though the size of the final result is moderate. Also, as the range of f is specified with respect to the DNFs' conjuncts instead of the m predicates themselves, this view prevents the straight-forward but elegant description of the addition or removal of predicates in terms of propositional operations (cf. Section 3.2). This paper therefore links the work on lookup DAGs with the theory of Boolean functions in general and that of BDDs in particular [24].

7. CONCLUSIONS AND FUTURE WORK

This paper has shown that, under three assumptions which typically hold for PA flavor languages, the average-case runtime cost incurred by residual dispatch can be improved upon by applying two optimizations. The first and foremost of these, namely the complete redundancy elimination across multiple residues, is facilitated by an alternative representation of the dispatch function in question: OBDDs. This representation does also allow for a straight-forward implementation of the second optimization, namely the reordering of atomic pointcuts in order of increasing cost. Fur-

thermore, as OBDDs are, like formulas, a representation of dispatch functions, they, too, allow for an elegant description of aspect deployment and undeployment.

It should be noted, however, that, strictly speaking, only point-in-time semantics [11] allow for the one-to-one correspondence between dispatch functions and join point shadows this paper so far has alluded to. Yet, dispatch functions can be used to good effect with AspectJ's region-in-time semantics [17] as well. They merely necessitate a separate treatment of **after returning** and **after throwing** advice. This is necessary, since in either case a parameter may be bound which is unavailable at the beginning of the join point's region-in-time. A straight-forward solution would require two dispatch functions, which handle the beginning of the join point's region-in-time and the end of the join point's region-in-time, respectively. Using several dispatch functions may also be advantageous in the presence of **around** advice which do not **proceed**. While such an advice may preclude the execution of other advice and is therefore narrowing, it leaves their pointcuts' satisfiability unaffected and hence does not violate the assumptions of Section 2. The precise workings of this scheme are, however, an area for future work.

Other areas for future work include methods which exploit static information, e.g., the fact that **args**(Integer) implies **args**(Number) or which perform profile-guided optimizations whose notion of optimality is based not on the average- but on the expected-case cost. In these areas, however, experiments require a detailed model of both the interdependencies of atomic pointcuts and the probability distribution underlying the set of states; such a model does not yet exist.

8. ACKNOWLEDGEMENTS

This work was supported by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349.

9. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, MA, USA, 2nd edition, 2006.

[2] The AspectJ Project. *The AspectJ Programming Guide*. `http://www.eclipse.org/aspectj/doc/released/progguide/`.

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. *ACM SIGPLAN Notices*, 40(6), 2005.

[4] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*. Springer, Berlin, Germany, 2006.

[5] C. Bockisch, M. Mezini, W. Havinga, L. Bergmans, and K. Gybels. Reference model implementation. Technical Report AOSD-Europe Deliverable D96, Technische Universität Darmstadt, 2007.

[6] B. Bollig and I. Wegener. A very simple function that requires exponential size read-once branching programs. *Information Processing Letters*, 66(2), 1998.

[7] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, 1990.

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

[9] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. *ACM SIGPLAN Notices*, 34(10), 1999.

[10] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Proceedings of the 22nd OOPSLA Conference*, 2007.

[11] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In *Proceedings of the 5th FOAL Workshop*, 2006.

[12] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th ECOOP Conference*, 1998.

[13] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, 12(1), 1993.

[14] J. Gergov and C. Meinel. Efficient Boolean manipulation with OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43(10), 1994.

[15] J. Gosling, W. N. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 3rd edition, 2005.

[16] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd AOSD Conference*, 2004.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th ECOOP Conference*, 2001.

[18] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th ECOOP Conference*, 2003.

[19] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th Conference on Compiler Construction*, 2003.

[20] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st AOSD Conference*, 2002.

[21] M. C. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT*, 2004.

[22] D. Sieling and I. Wegener. Graph driven BDDs: A new data structure for Boolean functions. *Theoretical Computer Science*, 141(1 & 2), 1995.

[23] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5), 2004.

[24] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

Certificate translation for specification-preserving advices

Gilles Barthe
INRIA Sophia-Antipolis
Gilles.Barthe@inria.fr

César Kunz
INRIA Sophia-Antipolis
Cesar.Kunz@inria.fr

ABSTRACT

Aspect Oriented Programming (AOP) has significant potential to separate functionality and cross-cutting concerns. In particular, AOP supports an incremental development process, in which the expected functionality is provided by a baseline program, that is successively refined, possibly by third parties, with aspects that improve non-functional concerns, such as efficiency and security. Therefore, AOP is a natural enabler for Proof Carrying Code (PCC) scenarios.

The purpose of this article is to explore a PCC architecture that accommodates an incremental development process. We extend our earlier work on certificate translation, and show in the context of a very simple AOP language that it is possible to generate certificates of executable code from proofs of aspect-oriented programs. To achieve this goal, we introduce a notion of specification-preserving advice, and provide a verification method for programs with specification-preserving advices.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Logics of programs

General Terms

Languages, Verification, Security

Keywords

AOP, Proof-carrying Code, Program Verification

1. INTRODUCTION

While reliability and security of executable code is an important concern, many program verification tools target high-level languages, and thus do not address the concerns of the code consumers, who require verification procedures that can be run on executable code and that dispense them from trusting code producers (that are potentially malicious), networks (that may be controlled by an attacker), and compilers (that may be buggy).

In a Proof Carrying Code (PCC) [20, 19] architecture, a certifying compiler returns, in addition to executable code, program annotations, which specify program invariants tailored to the desired policy, and a checkable proof, a.k.a. certificate, that the code is compliant to the policy. Through its associated verification mechanisms for executable code, PCC addresses the security concerns for mobile code. Nevertheless, current instances of certifying compilers mostly focus on basic safety policies and do not take advantage of the existing methods for verifying source code.

In order to overcome the limitations of certifying compilers, earlier work [6, 5, 7, 18] has considered expressive verification methods for executable code and established their adequacy with respect to verification methods for source programs. In particular, Burdy and Pavlova [7] have developed a proof compiler for Java, that enables certificates of Java bytecode programs to be constructed from source code verification with JML-based tools such as ESC/Java and Jack.

Proof compilation is an important step towards supporting expressive policies since proof compilers allow certificate generation to rely on widely used verification environments, and thus enables to address expressive policies (at the cost of interactive verification). Nevertheless, proof compilation currently targets Java programs and does not provide support for advanced programming idioms such as aspects.

Contributions. The main contribution of this work is to study proof compilation for a very simple AOP language.

In order to realize proof compilation, we introduce the notion of specification-preserving advice. Informally, an advice a is specification-preserving for an annotated piece of code $\{\Phi\}c\{\Psi\}$, where Φ and Ψ respectively denote the pre and postcondition for c, if the advised code $a \triangleright c$ satisfies the same specification, i.e. $\{\Phi\}a \triangleright c\{\Psi\}$. Specification-preserving advices are natural in the context of PCC with intermediaries, since many aspects related to security (resource management, logging, *etc.*) and efficiency (e.g. cached functions, optimized code,*etc.*) fall in this category. Moreover, specification-preserving advices support "separate verification" (as coined by [16]) and allow intermediaries to treat correctness proofs of the baseline code as black-boxes.

In summary, the contributions of this article are:

- the definition of the class of specification-preserving advices that support modular reasoning, and a mild

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

generalization of the classification of specification-preserving advices to sequences of advices;

- the relationship between specification-preserving advices and harmless advices [11], which are required to verify the stronger property of preserving the semantics of advised code, except for the possibility of modifying the termination behavior. Inspired by this relationship, we provide a simple static analysis that ensures that advices are specification-preserving;
- an algorithm that takes as input an AOP program p and a certificate c of its correctness, and returns a certificate for the compiled program $[\![p]\!]$.
- a mild generalization of the classification of specification-preserving advices to sequences of advices.

2. A BASIC MOTIVATING EXAMPLE

Consider the program p with a procedure main and another procedure twice advised unconditionally by a:

$$\begin{array}{rcl} \mathsf{main}(x) & = & y := \mathsf{twice}(x); z := y + x;\ \mathsf{return}\ z \\ \mathsf{twice}(x) & = & \mathsf{return}\ (x + x) \\ a(x) & = & x := 0; z := \mathsf{proceed}(x);\ \mathsf{return}\ z \end{array}$$

The correctness of the program is established w.r.t. a specification table Γ that associates to each procedure a triple consisting of a precondition, a postcondition, and a modifies clause that states which variables are modified. We choose the obvious specifications for main and twice, i.e.

$$\begin{array}{rcl} \Gamma(\mathsf{main}) & = & (\mathsf{true}, \mathsf{res} = x^\star + x^\star + x^\star, \emptyset) \\ \Gamma(\mathsf{twice}) & = & (\mathsf{true}, \mathsf{res} = x^\star + x^\star, \emptyset) \end{array}$$

(We consider that the variables y and z are local variables, and thus are not declared in the modified clauses).

One can generate for each procedure a verification condition that guarantees, in a traditional setting, that the procedure meets its specification. Both verification conditions hold obviously. Nevertheless all terminating executions of the program will simply return the value given as input, and thus the postcondition will not be satisfied if main is called with an input distinct from 0. In this case, the problem is caused by the fact that a forces twice to be executed with input 0. In other words, a is not parameter-preserving, i.e. causes f to be called with an input different from the one that is declared in the program.

A similar problem shall occur if an advice modifies a global variable that is otherwise unmodified by the procedures it advises. More generally, advices should, in addition to be parameter-preserving, preserve specifications. Consider the modified advice $a(x)$:

$$(\texttt{if}\ x = 0\ \texttt{then}\ z := \mathsf{proceed}(x)\ \texttt{else}\ z := 0);\ \mathsf{return}\ z$$

As in the previous case, the postcondition will not be satisfied if main is called with an input distinct from 0. The problem is caused by the fact that a is not specification-preserving. Indeed, consider the function $\hat{a}$ derived from a by replacing the proceed statement by a call to f:

$$\begin{array}{rcl} \hat{a}(x) & = & (\texttt{if}\ x = 0\ \texttt{then}\ z := \mathsf{twice}(x)\ \texttt{else}\ z := 0); \\ & & \mathsf{return}\ z \end{array}$$

One cannot prove that the procedure $\hat{a}$ satisfies the specification of twice, since the proof obligation for $\hat{a}$ with the same pre and postcondition as twice is logically equivalent to $x = 0 \Rightarrow x + x = x + x \wedge x \neq 0 \Rightarrow 0 = x + x$ which does not hold.

Now consider instead the correct advice $a(x)$:

$$(\texttt{if}\ x \neq 0\ \texttt{then}\ z := \mathsf{proceed}(x)\ \texttt{else}\ z := 0);\ \mathsf{return}\ z$$

The function $\hat{a}(x)$ derived from $a(x)$ by replacing the proceed statement by a call to f:

$$(\texttt{if}\ x \neq 0\ \texttt{then}\ z := \mathsf{twice}(x)\ \texttt{else}\ z := 0);\ \mathsf{return}\ z$$

is specification-preserving, since the proof obligation for $\hat{a}$ with with the same pre and postcondition as twice is logically equivalent to

$$x \neq 0 \Rightarrow x + x = x + x \wedge x = 0 \Rightarrow 0 = x + x$$

and it is thus valid. Note that the proof obligations for $\hat{a}$ relies on the specification of twice, but not on its code.

Commands	c	::=	$v{:=}e \mid c; c \mid v{:=}f(e)$
		$\mid$	$v{:=}\ \mathsf{proceed}(e)$
		$\mid$	$\texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c$
		$\mid$	$\texttt{while}\ b\ \texttt{do}\ c$
		$\mid$	$\texttt{skip} \mid \texttt{return}\ e$
Procedures	$proc$	::=	$f\ arg^*\ c_b$
Point-cut descriptors	ptd	::=	$\mathsf{if}\ b\ \mathsf{around}\ f$
Advices	$advice$	::=	$ptd^+\ a\ arg^*\ c_a$
Programs	$Prog$	::=	$proc^*\ advice^*$

Figure 1: SYNTAX OF SAL PROGRAMS

3. A SIMPLE AOP LANGUAGE

This section introduces SAL, a simple procedural language with aspects. For simplicity, SAL is restricted to around advices, to point-cuts at procedure calls, and to point-cut descriptors that do not refer to the control-flow graph.

3.1 Syntax

The syntax of commands can be found in Figure 1, where v ranges over the sets $\mathcal{V}$ of local variables and $\mathcal{X}$ of global variables, arg ranges over local variables, f ranges over the set $\mathcal{F}$ of procedure names, and a ranges over the set $\mathcal{A}$ of advice names. A baseline command is a command that does not contain any proceed command. We let c_b and c_a range respectively over baseline and advice commands.

Point-cut descriptors are of the form $\mathsf{if}\ b\ \mathsf{around}\ f$, where b is a boolean condition and f is a procedure name. Then, each procedure is composed of an identifier, its formal parameters and a command that represents its body. Each advice is composed of an identifier from a set $\mathcal{A}$ of advice names, a non-empty set of point-cut descriptors, its formal parameters, and an extended command that represents its body. A program is a given by a set of procedures with a distinguished main procedure and a set of advices.

3.2 Semantics

Advice weaving, which enables aspects to influence the execution of programs at designated program points and under certain conditions, is the fundamental mechanism that determines the semantics of AOP programs. Thus, the essence of SAL programs is captured by the transition rules for the commands call and proceed, which are described informally below. For simplicity, we restrict our attention to procedures

Logical expressions	$\bar{e}$	::=	$\mathsf{res} \mid x^\star \mid x \mid c \mid \bar{e}\ op\ \bar{e}$
Propositions	ϕ	::=	$\bar{e}\ cmp\ \bar{e} \mid \neg\phi \mid \phi \wedge \phi$
		$\mid$	$\phi \vee \phi \mid \phi \Rightarrow \phi \mid \ldots$

Figure 2: Specification language

and advices with a single formal parameter. The semantics of all remaining constructs is defined in the usual way.

Upon reaching a call statement of the form $v{:=}f(e)$, one checks in the order prescribed by the declaration of advices whether the guard of a point-cut descriptor for f is satisfied. If there is no point-cut descriptor for f such that the guard is satisfied, then one starts a new execution frame, initializes the local variable *par* with the value of e, and executes the body of f; otherwise, if a is the first advice for f whose guard is satisfied, then one starts a new execution frame, initializes the local variable *par* with the value of e, and executes the body of a.

Upon reaching a statement of the form $v{:=}\,\texttt{proceed}(e)$, one must examine the call stack to determine the current procedure, say f, and the current advice, say a. Then one checks for all advices that occur after a in the declaration of advices whether the guard of a point-cut descriptor for f is satisfied. If there is no point-cut descriptor for f such that the guard is satisfied, then one starts a new execution frame, initializes the local variable *par* with the value of e, and executes the body of f; otherwise, if a' is the first advice for f whose guard is satisfied, then one starts a new execution frame, initializes the local variable *par* with the value of e, and executes the body of a'.

Under such a semantics, the body of f will not be executed whenever a procedure call to f, say $v{:=}f(e)$, triggers an advice that does not contain any proceed statement, or contains a proceed statement that is not reached during execution. Furthermore, if an advice contains two or more proceed statements, then execution will stop upon reaching the second proceed statement.

Formally, the semantics of advice weaving is defined by compilation to an intermediate language SBL, defined in Section 6. For the purpose of the next sections, it is sufficient to know that the semantics of SAL programs can be modeled by judgments of the form $p, \mu \Downarrow v, \nu$ which read: the execution of program p with initial memory μ terminates with final memory ν and returns value v.

4. VERIFICATION OF BASELINE CODE

In this section, we focus on baseline programs, i.e. programs without advices, and introduce for such programs a verification method based on the idea of contract. Therefore, each procedure is specified in terms of a precondition, which captures the situations under which the procedure can be called, and a postcondition, which establishes a relationship between the inputs and outputs of the procedure, and a frame condition that specifies which variables are modified during the execution of f, and that is used by the verification condition generator to improve its context-sensitivity.

The set of propositions is defined in Figure 2, where $x^\star$ is a special, so-called starred, variable representing the initial value of the variable x, and res is a special value representing the final value of the evaluation of the program. Program specifications rely on particular classes of propositions:

- preconditions, which refer to the formal parameters of the function and global variables but do not refer to starred variables (since redundant at an initial state), nor the result (special variable res);
- postconditions, which refers to the formal parameters, and the initial and current state of global variables (respectively with starred and standard variables);
- loop invariants, which do not refer to the return value (i.e. the special variable res).

Each precondition Φ yields a predicate over states, denoted $\mu \models \Phi$ for a state μ, whereas a postcondition Ψ yields a ternary relation over an initial state, a final state, and a result, denoted $\mu, \nu, v \models \Psi$ for the states μ and ν and the value v. Likewise, invariants yield binary relations over an initial and a current state.

In order to reason effectively about programs, we assume that each procedure is annotated, i.e. that all while loops in its body carries an invariant (we use $\mathsf{while}_I(b)\{s\}$ to denote the loop $\mathsf{while}_I(b)\{s\}$ annotated with invariant I), and that we dispose of a specification table Γ that associates to each procedure f a triple $(\Phi, \Psi, \mathcal{W})$ where Φ is a precondition, Ψ is a postcondition, and $\mathcal{W}$ is a *modifies* clause that declares all variables that are modified during the execution of f. Furthermore, we let $\mathcal{V}_\Gamma$ be the set of variables that appear in the specification of baseline procedures.

It may be argued that the specification overhead can make the approach impractical. However, that depends strictly on the complexity of the properties we intend to specify. In a practical implementation, we can consider as specification the result of a static analysis represented in terms of logical formulae. In that case the specification overhead is reduced while the results presented in this paper are still applicable.

Given a specification table Γ, one can compute for each annotated procedure f a set $\mathsf{PO}_\Gamma(f)$ of verification conditions. The verification conditions are defined using an extended predicate transformer vcg, which takes as input a baseline command c and a postcondition Ψ, and returns a precondition Φ and a set of proof obligations Δ_f. Formally, the set $\mathsf{PO}_\Gamma(f)$ is defined as $\Delta_f \cup \{\Phi \Rightarrow \Phi'[{}^{y}/_{y^\star}]\}$, where $\varphi[{}^{e}/_{x}]$ stands for the substitution of the expression e for the free occurrences of variable x in the logic formula φ, $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, y stands for every variable in $\mathcal{V}_\Gamma$ and $\mathsf{vcg}(c, \Psi) = (\Phi', \Delta_f)$, where c is the body of f. We say that a procedure is valid if all its proof obligations are valid formulae, and that a program is valid if all its procedures are. The formal definition of vcg is given in Figure 3.

For the verification method to be sound, we must also check the correctness of the *modifies* clause. Even though we can propose a logic to verify this frame condition, we assume a sound but incomplete automatic analysis that checks its correctness.

The weakest precondition calculus is sound in the sense that if a program p is valid w.r.t. a specification table Γ with a main procedure specified by (Φ, Ψ), then all executions of p initiated with a memory μ satisfying Φ will terminate with a final memory ν and value v such that (μ, ν, v) satisfy Ψ.

Lemma 1 (Soundness). *Let p be a baseline program over a set $\mathcal{F}$ of procedures. Let Γ be a specification table for p and let $\Gamma(\mathsf{main}) = (\Phi, \Psi, \mathcal{W})$. Assume that p is valid w.r.t. Γ. Then, if $p, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$.*

$$
\begin{array}{rcl}
\text{let } \Gamma(f)=(\Phi,\Psi,\mathcal{W}) \text{ in} & & \\
\mathtt{vcg}(\mathtt{skip},\varphi) & = & (\varphi,\emptyset) \\
\mathtt{vcg}(x{:=}e,\varphi) & = & (\varphi[{}^{e}\!/_{x}],\emptyset) \\
\mathtt{vcg}(c_1;c_2,\varphi) & = & \text{let } (\varphi_2,S_2){=}\mathtt{vcg}(c_2,\varphi) \text{ in let } (\varphi_1,S_1){=}\mathtt{vcg}(c_1,\varphi_2) \text{ in}(\varphi_1,S_1\cup S_2) \\
\mathtt{vcg}(\mathtt{return}\ e,\varphi) & = & (\varphi[{}^{e}\!/_{\mathsf{res}}],\emptyset) \\
\mathtt{vcg}(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2,\varphi) & = & \text{let } (\varphi_1,S_1){=}\mathtt{vcg}(c_1,\varphi) \text{ in let } (\varphi_2,S_2){=}\mathtt{vcg}(c_2,\varphi) \text{ in}(b{\Rightarrow}\varphi_1\wedge\neg b{\Rightarrow}\varphi_2,S_1\cup S_2) \\
\mathtt{vcg}(\mathtt{while}\ b\ \{Inv\}\ \mathtt{do}\ c,\varphi) & = & \text{let } (\varphi',S){=}\mathtt{vcg}(c,Inv) \text{ in}(Inv,\{Inv{\Rightarrow}(b{\Rightarrow}\varphi'\wedge\neg b{\Rightarrow}\varphi)\}\cup S) \\
\mathtt{vcg}(x{:=}f(e),\varphi) & = & \Phi[{}^{e}\!/_{\mathtt{in}_f}]\wedge(\forall_{\mathcal{W}',\mathsf{res}}.\Psi[{}^{e}\!/_{\mathtt{in}_f}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}][{}^{\mathcal{W}}\!/_{\mathcal{W}^\star}]{\Rightarrow}\varphi[{}^{\mathsf{res}}\!/_{x}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}],\emptyset) \\
\mathtt{vcg}_f(x{:=}\ \mathtt{proceed}(e),\varphi) & = & \Phi[{}^{e}\!/_{\mathtt{in}_f}]\wedge(\forall_{\mathcal{W}',\mathsf{res}}.\Psi[{}^{e}\!/_{\mathtt{in}_f}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}][{}^{\mathcal{W}}\!/_{\mathcal{W}^\star}]{\Rightarrow}\varphi[{}^{\mathsf{res}}\!/_{x}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}],\emptyset)
\end{array}
$$

Figure 3: WEAKEST PRECONDITION FUNCTION

In the setting of PCC, we require that proof obligations are certified, i.e. that programs come equipped with independently checkable proofs of their validity. For the purpose of our work, we do not need to commit to any particular format for certificate, nor do we need to specify an algorithm to check certificates. Instead, we rely on an abstract notion of certificate. Finally, we define a certified program as one whose functions are certified, i.e. carry valid certificates for the proof obligations attached to them. Formally, let p be an annotated baseline program and Γ be a specification table. Then, a certificate for the program p w.r.t. Γ is an indexed set of certificates $(c_\delta)_{\delta\in\mathsf{PO}_\Gamma(f),f\in\mathcal{F}}$ such that $c_\delta :\vdash \delta$ for all δ belonging to $\mathsf{PO}_\Gamma(f)$ and for all procedures f. If such a certificate exists, we say that p is certified w.r.t. Γ.

If a program p is certified w.r.t. a specification table Γ, then it is obviously valid w.r.t. Γ.

5. VERIFYING AOP PROGRAMS

As illustrated by the examples of Section 2, soundness fails for programs with advice, as expected since verification condition generation is oblivious to aspects. The purpose of this section is to define a method to verify SAL programs; the verification method is based on the notion of specification-preserving advice, which is introduced formally below.

Throughout this section, we consider a program p in which all procedures are annotated, i.e. have loop invariants, and specified in a table Γ.

5.1 Specification-preserving advices

In order to reason about advices, we extend the verification condition generator to proceed statements. The extension is parametrized by the name of the advised function, and the proceed statement is interpreted as a call to this function; see Figure 3. Note that when reasoning about an advice a, in order for the verification condition generator to be effective we need one set of loop invariants for each procedure f that a is advising.

DEFINITION 1. *An advice a with guard b preserves the specification of method f w.r.t. Γ if it satisfies the specification $(b\wedge\Phi,\Psi,\mathcal{W}')$ where $\Gamma(f)=(\Phi,\Psi,\mathcal{W})$, and $\mathcal{W}'\cap\mathcal{V}_\Gamma\subseteq\mathcal{W}$.*

The condition $\mathcal{W}'\cap\mathcal{V}_\Gamma\subseteq\mathcal{W}$ states that the advice a only modifies in $\mathcal{W}$, unless they do not appear originally on the specification of the baseline program. We let $\mathsf{PO}_{\Gamma,f}(a)$ stand for the set of proof obligations required to prove that the advice a is specification-preserving w.r.t. f and Γ. Formally, if $\Gamma(f)=(\Phi,\Psi,\mathcal{W})$ and c is the body of a, the set $\mathsf{PO}_{\Gamma,f}$ is defined as $\Delta_{a,f}\cup\{\Phi\Rightarrow\phi[{}^{y}\!/_{y^\star}]\}$ where $(\phi,\delta_{a,f})=\mathsf{vcg}(c,\Psi)$ and $y^\star$ stands for every starred variable in ϕ.

If all advices are specification-preserving, then baseline program verification is sound. To state this result, one first extends the notion of valid advice, and valid program. Let (p,Γ) be an annotated program. We say that an advice a is valid if for all procedures f that it advises, the set of proof obligations $\mathsf{PO}_{\Gamma,f}(a)$ is valid. Then, we say that the program p is valid if all its procedures and all its advices are valid.

We can now state soundness of the verification method in the presence of advice weaving.

LEMMA 2 (SOUNDNESS). *Let (p,Γ) be a valid annotated program. Then, if $p,\mu\Downarrow v,\nu$ and $\mu\models\Phi$, then $\mu,\nu,v\models\Psi$.*

One can extend the notion of certified baseline program to programs with specification-preserving advices, by requiring that programs come equipped with a certificate that advices are specification-preserving.

Remark. We can extend the scope of this paper to a language with a richer set of point-cut descriptors, for instance to point-cut descriptors that refer to the control-flow graph. To this end, as an alternative to reasoning about the control-flow graph or the call-stack in our logic, we propose a stronger definition of specification preserving advices. An advice a is specification-preserving w.r.t. f and Γ if it satisfies the specification $(\Phi,\Psi,\mathcal{W}')$ where $\Gamma(f)=(\Phi,\Psi,\mathcal{W})$, and $\mathcal{W}'\cap\mathcal{V}_\Gamma\subseteq\mathcal{W}$. Notice that, in contrast to previous definition, the guard b does not appear in the precondition of a.

5.2 Example

To illustrate the approach with a running example we assume an extended program syntax. Consider a procedure $g\doteq\mathtt{slowRetrieve}$ of a SAL program p, that returns the value stored in a slow access memory. That is, given as parameter the integer *Address* `i`, the procedure g returns the value `mem`[`i`], where `mem` is a global array variable, if `i` is within the accessible range.

Since we plan to improve the efficiency of the procedure g, we consider two auxiliary global array variables `available` and `cache` and the SAL procedures $f_1\doteq\mathtt{updateCache}$ and $f_2\doteq\mathtt{isAvailable}$. Let ϕ stand for the consistency of the `cache` variable with respect to the array `availability`, i.e. $\phi\doteq\forall i.(\mathtt{available}[i]\Rightarrow\mathtt{cache}[i]=\mathtt{mem}[i])$. For simplicity, we assume that global variables `available` and `cache` are only accessible by these procedures.

Consider a specification table Γ such that $\Gamma(g)=(\Phi,\Psi,\mathcal{W})$ where $\Phi\doteq 0\le\mathtt{i}<N\wedge\phi$, $\Psi\doteq\mathsf{res}=\mathtt{mem}[\mathtt{i}]\wedge\phi$ and $\mathcal{W}=\emptyset$.

Similarly, we specify procedures f_1 and f_2 with their respective pre and postconditions:

$$
\begin{array}{l}
\Phi_1\doteq\Phi \\
\Psi_1\doteq\mathtt{cache}=\mathtt{cache}^\star[\mathtt{i}\mapsto\mathtt{v}]\wedge\phi \\
\Phi_2\doteq 0\le\mathtt{i}<N \\
\Psi_2\doteq\mathsf{res}=\mathtt{available}[i]
\end{array}
$$

Consider the introduction of an advice $a \doteq$ `fastRetrieve` that improves the store access time by taking advantage of the array variables `available` and `cache` and the procedures f_1 and f_2. This advice replaces the functionality of method g by receiving as parameter the store address `i` and returning the *cached* value if available or, otherwise, by permitting the original function g to continue:

```
around slowRetrieve(Address i) fastRetrieve {
    b:= isAvailable(i);
    if b
        return cache[i]
    else
        v:=proceed(i);
        updateCache(i, v);
        return v
}
```

Then, we can prove that a is specification preserving by showing that the proposition

$$\begin{array}{l}\Phi_2 \wedge \forall_b.(\Psi_2[{}^{b}/_{\mathsf{res}}] \Rightarrow \\ \quad b \Rightarrow \Psi[{}^{\mathtt{cache[i]}}/_{\mathsf{res}}] \wedge \phi \\ \quad \wedge \\ \quad \neg b \Rightarrow \Phi \wedge \forall_{\mathsf{res}}.(\Psi \Rightarrow \Phi_1 \wedge \\ \qquad \forall_{\mathtt{cache}'}.(\Psi_1[{}^{\mathtt{cache}'}/_{\mathtt{cache}}][{}^{\mathtt{cache}}/_{\mathtt{cache}^\star}] \Rightarrow \\ \qquad\quad (\Psi \wedge \phi)[{}^{\mathtt{cache}'}/_{\mathtt{cache}}])))\end{array}$$

is implied by Φ.

5.3 Harmless advices

In general, it is not decidable whether an advice a preserves the specification of a procedure f w.r.t. a specification table Γ. Therefore, it is of interest to develop automated approximate methods to detect specification-preserving advices. A natural condition is to require that the advice does not modify the variables in $\mathcal{V}_\Gamma$ and always executes a proceed statement. Since such requirements are closely related to the notion of harmless advice, we call such advices specification-harmless.

The set of SAL commands is extended with assertions $\mathsf{assert}(\phi)$ and ghost assignments $\mathsf{set}\ z' := z$, where ϕ is a proposition and z' is a ghost variable not appearing in the original program. The definition of vcg is extended accordingly:

$$\begin{array}{lcl}\mathsf{vcg}(\mathsf{assert}(\phi), \varphi) & = & (\phi, \{\phi \Rightarrow \varphi\}) \\ \mathsf{vcg}(\mathsf{set}\ z' := e, \varphi) & = & (\phi[{}^{e}/_{z'}], \emptyset)\end{array}$$

Formally, an advice a with parameters $\vec{y}$ and guard b is specification-harmless w.r.t. f and Γ if the procedure $\hat{a}$ whose body is obtained from the body of a by substituting $x{:=}\ \mathtt{proceed}(\vec{e})$ by

$$\mathsf{assert}(z^\star = \vec{z}); x{:=}f(\vec{y}); \mathsf{set}\ x', \vec{z'} := x, \vec{z}$$

satisfies the specification

$$(b \wedge \Phi, x' = \mathsf{res} \wedge \vec{z'} = \vec{z}, \mathcal{W}')$$

where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma = \emptyset$, and where $x', \vec{z'}$ are fresh ghost variables, and where $\vec{z}$ is an enumeration of $\mathcal{V}_\Gamma$. We classify an advice as *control flow preserving* if every path in its control flow contains exactly one `proceed` statement. We assume the existence of an automated approximate static analysis to check this condition.

LEMMA 3. *Let a be a control-flow preserving advice. Then, if a is specification-harmless with respect to f and Γ, then it is specification-preserving.*

$$\begin{array}{lcl}instr & ::= & \mathsf{nop} \mid \mathsf{push}\ v \mid \mathsf{load}\ x \mid \mathsf{store}\ x \\ & & \mid \mathsf{jmp}\ l \mid \mathsf{jmpif}\ cmp\ l \\ & & \mid \mathsf{invoke} \mid \mathsf{return}\end{array}$$

Figure 4: INSTRUCTION SET FOR SBL

Dantas and Walker [11] propose a mechanism to check that the execution of an advice does not interfere with the final value produced by the computation of the baseline procedure. It consists on a type-effect system inspired on information flow type systems that does not consider timing nor termination behavior. One can use this type system as a static analysis to detect whether an advice is specification-harmless.

5.4 Beyond harmless advices

There are many natural examples of advices that do not necessarily trigger a proceed statement. For example, advices that seek to improve efficiency by replacing a procedure call by a semantically equivalent but more efficient computation will not call a proceed statement. For such examples of advices, it is still possible to use the property of specification-harmless to ensure that the advice is specification-preserving for those paths in which a proceed statement is effectively called, and generate a proof obligation for all paths that do not call to proceed.

Recall the advice of the basic example shown in Section 2:

$$\begin{array}{lcl}a(x) & = & (\mathtt{if}\ x \neq 0\ \mathtt{then}\ z := \mathsf{proceed}(x)\ \mathtt{else}\ z := 0); \\ & & \mathsf{return}\ z\end{array}$$

Clearly, we have two possible execution paths depending on whether the input value is equal to 0. To verify that a preserves the specification of f, i.e. $(\mathsf{true}, \mathsf{res} = x^\star + x^\star)$, we consider each possible path separately. In case that the parameter x is not equal to 0 we know that exactly one `proceed` statement will be executed, that no variable is modified and that the expression returned by the proceed statement is passed unchanged by the advice. Thus, we can use a simple static analysis to detect whether this path is specification-harmless. However, the path corresponding to an input equal to 0 does not execute a `proceed` statement, so we need to generate proof obligations that ensures that the specification is still preserved. In this case, it corresponds to the valid proposition $x = 0 \Rightarrow 0 = x + x$.

6. COMPILING ADVICES

From an applicative perspective, AOP is transparent and compilers target typical back-ends: indeed, it is the role of the compiler to integrate these concerns into a single executable object, through a weaving mechanism that modifies the code of each procedure depending on the advices that operate over it. In this section, we define the compilation of SAL programs to a stack-based language.

6.1 Target language

The target language is a simple stack-based language (SBL) that can be used to compile the imperative core of SAL. The syntax of SBL instructions is given in Figure 4, where v and l ranges over integers, x ranges over program variables, *cmp* over relations between integer values, and g ranges over function names. A SBL program consists of a set of func-

$$\frac{p_f[i] = \mathsf{invoke}\ f}{\langle \mu, \langle f', pc, lm, v : os\rangle :: lf\rangle \rightsquigarrow \langle \mu, \langle f, 1, [par \mapsto v], \epsilon\rangle :: \langle f', pc+1, lm, os\rangle :: lf\rangle}$$

$$\frac{p_f[i] = \mathsf{return}}{\langle \mu, \langle f, pc, lm, v : os\rangle :: \langle f', pc', lm', os'\rangle :: lf\rangle \rightsquigarrow \langle \mu, \langle f', pc', lm', v :: os'\rangle :: lf\rangle}$$

Figure 5: Operational semantics of SBL

tion names, and for each function g a declaration of the form $g\ args^* = instr^*$. The operational semantics of SBL programs is standard, and defined by a small-step relation $\rightsquigarrow$ between states. A state is either final, in which case it consists of a global memory μ and a result value v, or intermediary, in which case it consists of a global memory μ and a list of frames lf, each frame consisting of the name of the function being called, of a program counter, of a local memory with a distinguished variable *par* that stores the parameter of the function being called, and of an operand stack. Figure 5 gives the rules for invoke and return instructions, where $[par \mapsto v]$ denotes the local memory that only assigns v to *par*.

6.2 Compiler

The compiler for SAL programs is defined in Figure 6 as a function $[\![]\!]$ that takes a command and returns a list of labeled instructions. It relies on a compiler for integer expressions and a compiler for boolean conditions, namely $[\![]\!]_{\mathsf{e}}$ and $[\![]\!]_{\mathsf{b}}$. The compiler $[\![]\!]_{\mathsf{e}}$ takes an integer expression e and returns a sequence of instructions whose effect is to push on top of the stack the evaluation of the expression e. The compiler $[\![]\!]_{\mathsf{b}}$ takes, in addition to a boolean expression b, a label l and outputs a sequence a instructions that forces the program execution to jump to the program point labeled l if the condition b evaluates to true. The compiler for commands is standard, to the exception of the function call statement, whose compilation involves advice weaving, and the `proceed` statement. Since SBL does not feature a dedicated mechanism for advice weaving, each advice is compiled multiple times, exactly once per procedure it advises, and the procedure call $x{:=}f(e)$ is compiled into

$$[\![e]\!]_{\mathsf{e}} :: \mathsf{invoke}\ \hat{a}_f :: \mathsf{store}\ x$$

where a is the first advice for f, and $\hat{a}_f$ is its specific compilation for f. The code of $\hat{a}_f$ is of the form

$$[\![b, l]\!]_{\mathsf{b}} :: \mathsf{load}\ par :: \mathsf{invoke}\ \hat{a}'_f :: \mathsf{return} :: [l : a_f]$$

where a_f is obtained by compilation from a by translating any proceed statement of the form $x{:=}$ `proceed`(e) by

$$[\![e]\!] :: \mathsf{invoke}\ a'_f :: \mathsf{store}\ x$$

where a' is the next advice for f. In other words, the code of $\hat{a}_f$ tests if the guard for a holds, and if so proceeds to execute the body of the advice, or lets $\hat{a}'_f$ proceed otherwise.

In order to achieve the desired effect, the compiler is thus parametrized by a procedure (used in the clause for procedure calls to trigger the appropriate advice), or by a procedure and an advice (used in the clause for proceed to trigger the appropriate advice). For readability, we use superscripts to indicate the parameter and omit the superscript in all cases where it is not used.

$$\begin{array}{ll}
[\![\texttt{skip}]\!] = & [l : \mathsf{nop}] \\
[\![x{:=}e]\!] = & \text{let } \mathsf{ins}_e = [\![e]\!]_{\mathsf{e}} \text{ in} \\
& \mathsf{ins}_e :: \mathsf{store}\ x \\
[\![c_1;c_2]\!] = & \text{let } \mathsf{ins}_1 = [\![c_1]\!] \text{ in} \\
& \text{let } \mathsf{ins}_2 = [\![c_2]\!] \text{ in} \\
& \mathsf{ins}_1 :: \mathsf{ins}_2 \\
[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!] = & \\
\quad \text{let } \mathsf{ins}_1 = [\![c_1]\!] \text{ in} & \\
\quad \text{let } \mathsf{ins}_2 = [\![c_2]\!] \text{ in} & \\
\quad \text{let } \mathsf{ins}_b = [\![b, l_1]\!]_{\mathsf{b}} \text{ in} & \\
\quad \mathsf{ins}_b :: \mathsf{ins}_2 :: \mathsf{jmp}\ l :: [l_1 : \mathsf{ins}_1] :: [l : \mathsf{nop}] & \\
[\![\texttt{while } b \texttt{ do } c]\!] = & \\
\quad \text{let } \mathsf{ins}_c = [\![c]\!] \text{ in} & \\
\quad \text{let } \mathsf{ins}_b = [\![b, l_c]\!]_{\mathsf{b}} \text{ in} & \\
\quad \mathsf{jmp}\ l :: [l_c : \mathsf{ins}_c] :: [l : \mathsf{ins}_b] & \\
[\![x{:=}h(e)]\!]^f = & \text{let } \mathsf{ins}_e = [\![e]\!]_{\mathsf{e}} \text{ in} \\
& \mathsf{ins}_e :: \mathsf{invoke}\ a_f :: \mathsf{store}\ x \\
[\![\texttt{return } e]\!] = & \text{let } \mathsf{ins} = [\![e]\!]_{\mathsf{e}} \text{ in} \\
& \mathsf{ins} :: \mathsf{return} \\
[\![x{:=}\ \texttt{proceed}(e)]\!]^a_f = & \text{let } \mathsf{ins}_e = [\![e]\!]_{\mathsf{e}} \text{ in} \\
& \mathsf{ins}_e :: \mathsf{invoke}\ a'_f :: \mathsf{store}\ x
\end{array}$$

Figure 6: Compiler for SAL programs

$$\begin{array}{lllll}
\textbf{stack expressions} & \bar{os} & ::= & \mathsf{os} \mid \bar{e} :: \bar{os} \mid \uparrow^k \bar{os} \\
\textbf{logical expressions} & \bar{e} & ::= & \mathsf{res} \mid x^\star \mid x \mid c \mid \bar{e}\ op\ \bar{e} \mid \bar{os}[k]
\end{array}$$

Figure 7: Logical SBL expressions

7. CERTIFICATE TRANSLATION

In this section, we show that a valid SAL program is compiled into a valid SBL program. To this end, we first define a verification method for SBL programs. The method is strongly inspired from earlier work, and in particular [6].

7.1 Verification of SBL programs

While program annotations are similar to those of SAL programs, the weakest precondition computation will produce propositions that refer to the operand stack, and thus the language of SBL annotations is extended to such propositions.

- The extended set of logical expressions is defined in Figure 7; the logical propositions are built as before. In the definition, os is a special variable representing the current operand stack and $\uparrow^k \bar{os}$ denotes the stack $\bar{os}$ minus its k-first elements. An annotation is a proposition that does not contain stack sub-expressions.
- An annotated bytecode instruction is either a bytecode instruction or a proposition and a bytecode instruction: $\bar{i} ::= i \mid (\phi, i)$
- An annotated program is a pair (p, Γ), where p is a bytecode program in which some instructions are annotated and Γ is a specification table that associates to

each procedure f a triple $(\Phi, \Psi, \mathcal{W})$ where Φ is a precondition, Ψ is a postcondition, and $\mathcal{W}$ is a *modifies* clause that declares all variables that may be modified during the execution of f.

Verification of SBL programs is defined in terms of a weakest precondition function wp that operates on annotated programs. In order for the wp function to be well-defined, we must restrict our attention to well-annotated programs [4, 6, 21], i.e. programs in which all cycles in the control-flow graph must pass through an annotated instruction. We characterize such programs by an inductive definition.

An annotated program p is well-annotated if every procedure is well annotated. A procedure g is well-annotated if every program point satisfies the predicate $\mathsf{reachAnnot}_g$ inductively defined by the clauses:

$$\frac{g[k] = (\phi, i)}{k \in \mathsf{reachAnnot}_g} \qquad \frac{g[k] = \mathsf{return}}{k \in \mathsf{reachAnnot}_g}$$

$$\frac{\forall k'.\ k \mapsto k' \Rightarrow k' \in \mathsf{reachAnnot}_g}{k \in \mathsf{reachAnnot}_g}$$

Given a well-annotated procedure, one generates an assertion for each label, using the assertions that were given or previously computed for its successors. This assertion represents the precondition that an initial state should satisfy for the procedure to terminate only in a state satisfying its postcondition.

Let (p, Γ) be a well-annotated program.

- The weakest precondition calculus over (p, Γ) is defined in Figure 8. Formally, the result of the weakest precondition calculus is a program in which all instructions are annotated.
- The set $\mathsf{PO}(f)$ of verification conditions of the procedure f is defined by the clauses:

$$\frac{}{\Phi \Rightarrow \mathsf{wp}_{\mathcal{L}}(0)[^{\vec{x^\star}}/_{\vec{x}}] \in \mathsf{PO}_\Gamma(f)} \qquad \frac{f[k] = (\phi, i)}{\phi \Rightarrow \mathsf{wp}_i(k) \in \mathsf{PO}_\Gamma(f)}$$

As before, an annotated SBL program is valid w.r.t. Γ if all its sets proof obligations $\mathsf{PO}_\Gamma(f)$ are valid.

7.2 Preservation of validity

The purpose of this section is to prove that valid SAL programs are compiled into valid SBL programs. To this end, we first extend the compiler of Section 6 so that compiled programs are well-annotated. This is achieved by modifying the compiler clause for loops:

$$[\![\mathsf{while}_I(b)\{c\}]\!] = \begin{array}{l}\text{let } \mathsf{ins}_c = [\![c]\!] \text{ and } \mathsf{ins}_b = [\![b, l_c]\!] \text{ in} \\ \mathsf{jmp}\ l :: [l_c : \mathsf{ins}_c] :: [l : (I, \mathsf{ins}_b)]\end{array}$$

where we denote (I, ins_b) the sequence of instructions obtained by annotating the first instruction of ins_b with I. In the rest of this section, for any SBL function g, we denote $g[l, l']$ the sequence of instructions $g[l] :: g[l{+}1] :: \ldots :: g[l'{-}1]$.

LEMMA 4. *Assuming the axioms* $(v :: \mathsf{os})[0] = v$ *and* $\uparrow (v :: \mathsf{os}) = \mathsf{os}$ *for stacks, the auxiliary compilers* $[\![]\!]_\mathsf{e}$ *and* $[\![.]\!]_\mathsf{b}$ *satisfy the following properties:*

i) for every integer expression e *and function* g *such that* $g[l, l'] = [\![e]\!]_\mathsf{e}$, $\mathsf{wp}_{\mathcal{L}}(l)$ *is equivalent to* $\mathsf{wp}_{\mathcal{L}}(l')[^{e::\mathsf{os}}/_{\mathsf{os}}]$;

ii) for every boolean expression b *and function* f *such that* $g[l, l''] = [\![b, l']\!]_\mathsf{b}$, $\mathsf{wp}_{\mathcal{L}}(l)$ *is equivalent to*

$$b \Rightarrow \mathsf{wp}_{\mathcal{L}}(l') \wedge \neg b \Rightarrow \mathsf{wp}_{\mathcal{L}}(l'')$$

Given a specification table Γ for SAL programs, Γ' is a specification table for SBL programs extending Γ if for every advice a and procedure f advised by a, $\Gamma'(\hat{a}_f) = (\Phi_f, \Psi_f, \mathcal{W}_f)$ and $\Gamma'(a_f) = (\Phi_f \wedge b, \Psi_f, \mathcal{W}_f)$, where $\Gamma(f) = (\Phi_f, \Psi_f, \mathcal{W}_f)$. In the following paragraphs, we implicitly consider the specification tables Γ and Γ' respectively for the verification of SAL and SBL programs.

LEMMA 5. *Let* g *be a* SBL *function such that* $g[l, l'] = [\![c]\!]$, *and let* $(\phi, S) = \mathsf{vcg}(c, \mathsf{wp}_{\mathcal{L}}(l'))$. *Then,* $\phi' \equiv \mathsf{wp}_{\mathcal{L}}(l)$ *and the proof obligations in* S *are equivalent to the proof obligations corresponding to the annotated instructions in* $g[l, l']$.

Consider a SBL program p' compiled from an annotated SAL program p. The following result states that if p is a valid SAL program w.r.t. Γ, then p' is a valid SBL program w.r.t. Γ'.

THEOREM 1. *Suppose that* (p, Γ) *is a valid annotated program. That is, for every procedure* f *and for every advice* a, *the sets of proof obligations* Δ_f *and* $\mathsf{PO}_{\Gamma,f}(a)$ *are valid. Then, for every function* f, a_f *and* $\hat{a}_f$, *the sets* $\mathsf{PO}_{\Gamma'}(f)$, $\mathsf{PO}_{\Gamma'}(a_f)$ *and* $\mathsf{PO}_{\Gamma'}(\hat{a}_f)$ *contain valid proof obligations.*

Furthermore, we can prove that a SAL programs certified with respect to Γ is compiled into a SBL program certified with respect to Γ'. More precisely, using the rules of the proof algebra extended with the axioms $(v :: \mathsf{os})[0] = v$ and $\uparrow(v :: \mathsf{os}) = \mathsf{os}$, for every equivalent proof obligations δ and δ', we can transform a certificate c_δ for δ to a certificate $c_{\delta'}$ for δ'. Therefore, if for every procedure $f \in \mathcal{F}$, $(c_\delta)_{\delta \in \mathsf{PO}_\Gamma(f)}$ and $(c_\delta)_{\delta \in \mathsf{PO}_{\Gamma,f}(a)}$ are indexed sets of certificates for a SAL program p, then for every function g of p' we can generate a certificate for the proof obligation $\delta \in \mathsf{PO}_{\Gamma'}(g)$.

8. INCREASING THE POWER OF VERIFICATION

Consider the following trivial example:

$$\begin{array}{rcl} a_1(x) & = & z := \mathsf{proceed}(x+1);\ \mathsf{return}\ z \\ a_2(x) & = & z := \mathsf{proceed}(x-1);\ \mathsf{return}\ z \end{array}$$

When executed in isolation around a function f, it is clear that neither a_1 nor a_2 preserves the behavior of f. However, when both are executed around f they collaborate, and the effect of a_1 is neutralized by the effect of a_2.

Then, since it may seem a bit restrictive to require that every advice in its own is specification-preserving, we propose a more general proof system to study instead whether a sequence of advices is specification preserving.

When verifying the behavior of a sequence of advices $\vec{a}$ executing around a function f, we are interested in verifying a specification for the sequence $\vec{a}$ around f (denoted $\vec{a} \triangleright f$), in addition to verifying each advice in isolation. As with functions and advices, the specification for sequences of advices executing around a function f consist on a precondition, a postcondition and a set of modifiable variables. This specification is inferred and proved from the specification of its components. For notational convenience, $\vec{a}$ may also stand for an empty sequence of advices.

let $\Gamma(f)=(\Phi,\Psi,\mathcal{W})$ and y represent every variable in $\mathcal{W}$:

$$
\begin{array}{rcll}
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{c::\mathsf{os}}/_{\mathsf{os}}] & \text{if } g[k]=\mathsf{push}\ c\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{(\mathsf{os}[0]\ op\ \mathsf{os}[1])::\uparrow^2\mathsf{os}}/_{\mathsf{os}}] & \text{if } g[k]=\mathsf{binop}\ op\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{x::\mathsf{os}}/_{\mathsf{os}}] & \text{if } g[k]=\mathsf{load}\ x\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{\uparrow\mathsf{os},\mathsf{os}[0]}/_{\mathsf{os},x}] & \text{if } g[k]=\mathsf{store}\ x\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(l) & \text{if } g[k]=\mathsf{jmp}\ l\\
\mathsf{wp}_i(k) & = & \quad(\mathsf{os}[0]\ \neq 0 \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[^{\uparrow^1\mathsf{os}}/_{\mathsf{os}}]) & \text{if } g[k]=\mathsf{jmpif}\ l\\
 & & \wedge\quad \mathsf{os}[0]=0 \Rightarrow \mathsf{wp}_{\mathcal{L}}(l)[^{\uparrow^1\mathsf{os}}/_{\mathsf{os}}]) & \\
\mathsf{wp}_i(k) & = & \Psi[^{\mathsf{os}[0]}/_{\mathsf{res}}] & \text{if } g[k]=\mathsf{return}\\
\mathsf{wp}_i(k) & = & \Phi[^{\mathsf{os}[0]}/_{\mathtt{in}}]\wedge & \text{if } g[k]=\mathsf{invoke}\ f\\
 & & (\forall \mathsf{res}, y'.\Psi[^{\mathsf{os}[0]}/_{\mathtt{in}}][^{y}/_{y^\star}][^{y'}/_{y}] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[^{\mathsf{res}::\mathsf{os}}/_{\mathsf{os}}][^{y'}/_{y}]) & \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } g[k]=\phi:i\\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Figure 8: Weakest precondition for SBL programs

For each nonempty sequence of advices $\vec{a_1}a\vec{a_2}$ executing around a function f, we call the sequence $\vec{a_2}\triangleright f$, i.e. the advices remaining to be executed around f when a executes a `proceed` statement, an *execution context* of a.

Verification proceeds in two steps. First, each advice a is verified in isolation, i.e. without considering the set of contexts in which the advice a may be executed. To this end, we must rely on a single specification for the expected behavior of the execution invoked by a `proceed` statement. In a second phase, for each context in which the advice may be executed, we check the consistency of the specification for the proceed statement w.r.t. the specification derived for the remaining context.

Verification of advices in isolation. We extend the specification of advices such that for every advice a we have, in addition to the tuple $(\Phi,\Psi,\mathcal{W})$, a specification for the code that may be invoked by a `proceed` statement. That enables to reason about the correctness of an advice abstracting from the possible contexts in which this advice may be invoked. The specification extension for an advice a consists on an extra and distinct tuple $(\Phi',\Psi',\mathcal{W}')$, in addition to the tuple $(\Phi,\Psi,\mathcal{W})$. The tuple $(\Phi',\Psi',\mathcal{W}')$ is such that $\mathcal{W}'$ specifies the set of variables that the code invoked by a proceed statement is allowed to modify, and Φ' and Ψ' are respectively the pre and postconditions of such invocation. The propositions Φ' and Ψ' may refer, in addition to the input and output arguments of a (`in` and res), to the input and output arguments of the invoked code, respectively represented with the new variables `in`$'$ and res'. It is the goal of the second phase to check, for every context in which the advice a may be executed, that the code allowed to proceed satisfies the specification $(\Phi',\Psi',\mathcal{W}')$.

The predicate transformer `wp` is extended for `proceed` statements, s.t. $\mathsf{wp}_a(x{:=}\ \texttt{proceed}(e),\phi)$ is defined as

$$(\Phi'_a[^{e}/_{\mathtt{in}'_a}] \wedge \forall_{y',\mathsf{res}'}.\Psi'_a[^{e}/_{\mathtt{in}'_a}][^{y'}/_{y}][^{y}/_{y^{\star\prime}}] \Rightarrow \phi[^{res'}/_{x}][^{y'}/_{y}][^{e}/_{\mathtt{in}'_a}], S)$$

where $(\Phi',\Psi',\mathcal{W}')$ correspond to the specification extension for the `proceed` statement and $y\in\mathcal{W}'$.

By using this modified `wp` function we can prove that the body of an advice satisfies its specification as long as the code invoked by a `proceed` statement satisfies the specification $(\Phi',\Psi',\mathcal{W}')$.

Verifying weaved code. After statically determining the sequence of advices $\vec{a}_f$ executing around f, we are interested in identifying a set of sufficient proof obligations that ensures that the sequence $\vec{a}_f$ is specification-preserving.

The collection of proof obligations is defined by induction on the length of the sequence of advices $\vec{a}_f$ executing around the procedure f. Since we do not require that every subsequence $\vec{a_f}'$ of advices preserves the specification, we generalize and accept the inference of pre and postconditions Φ and Ψ for $\vec{a_f}'\triangleright f$ without requiring Φ and Ψ to be compatible with the pre and postcondition of f. The goal of the verification for each subsequence $\vec{a}$ of $\vec{a}_f$ is a judgment of the form $\Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi\}\vec{a}\triangleright f\{\Psi\}$. For such a judgment, we do not require Φ and Ψ to be compatible with the pre and postcondition of f, i.e. the subsequence $\vec{a}$ is not necessarily specification-preserving.

To verify a judgment $\Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi\}\vec{a}\triangleright f\{\Psi\}$, we proceed by induction on the length of the sequence $\vec{a}$ to identify the set of proof obligations $\Delta_{\vec{a}}(\Phi,\Psi)$.

In the base case, i.e. when no advice is executed around the function f, we have the judgment $\Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi\}f\{\Psi\}$ without premises, where Φ and Ψ are the pre and postconditions of f.

Given a non-trivial sequence $\vec{a}=a\vec{a}'$, we consider two alternative sets of verification conditions, depending on whether we can statically ensure that the code of the advice a is *control flow preserving*. We assume an automated static mechanism to check this condition.

In case that it cannot be checked whether a is control-flow preserving we apply the following rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}}(a)=\langle(\Phi_a,\Psi_a,\mathcal{W}_a),(\Phi'_a,\Psi'_a,\mathcal{W}'_a)\rangle\\ \Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi'\}\vec{a}'\triangleright f\{\Psi'\}\\ \Phi'_a\Rightarrow\Phi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}] \qquad \Psi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][^{\mathsf{res}'}/_{\mathsf{res}}]\Rightarrow\Psi'_a \qquad \mathcal{W}_f\cup\mathcal{W}_{\vec{a}'}\subseteq\mathcal{W}'_a\end{array}}{\Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi_a\}a\vec{a}'\triangleright f\{\Psi_a\}}$$

For simplicity, we are not considering the boolean condition specified in the point-cut descriptor.

Unfortunately, the rule above makes hard to propagate the information carried by the specification (Φ',Ψ'), unless it is explicitly stated in the specification (Φ_a,Ψ_a) of a. However, under the hypothesis that a is a *control flow preserving* advice we can apply the following alternative rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}}(a)=\langle(\Phi_a,\Psi_a,\mathcal{W}_a),(\Phi'_a,\Psi'_a,\mathcal{W}'_a)\rangle\\ \Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi'\}\vec{a}'\triangleright f\{\Psi'\}\\ \Phi\Rightarrow\Phi_a\wedge\forall x'.(\Phi'_a[^{x'}/_{x}]\Rightarrow\Phi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][^{x'}/_{x}]) \qquad \mathcal{W}_f\cup\mathcal{W}_{\vec{a}'}\subseteq\mathcal{W}'_a\\ \Psi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][^{\mathsf{res}'}/_{\mathsf{res}}][^{y^\star}/_{y^\star}]\Rightarrow\Psi'_a\wedge\forall x'.(\Psi_a[^{\mathtt{in}}/_{\mathtt{in}_a}][^{x'}/_{x}]\Rightarrow\Psi[^{x'}/_{x}])\end{array}}{\Gamma,\Gamma_{\mathsf{a}}\vdash\{\Phi\}a\vec{a}'\triangleright f\{\Psi\}}$$

where x' represents the global variables potentially modified

by a, and W'_a specifies the variables that may be modified by the execution triggered by the `proceed` statement.

For every procedure f advised by $\vec{a}_f$, we define $\Delta_{\vec{a}_f}(\Phi, \Psi)$ as the set of proof obligations required to derive the judgment $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a}_f \triangleright f\{\Psi\}$. Assume the specification table Γ is such that $\Gamma(f) = (\Phi_f, \Psi_f, \mathcal{W})$. Then, we say that the sequence $\vec{a}_f$ is specification preserving with respect to f, Γ and Γ_{a}, if $\Phi_f \Rightarrow \Phi$, $\Psi \Rightarrow \Psi_f$ and the proof obligations in $\Delta_{\vec{a}_f}(\Phi, \Psi)$ are valid.

LEMMA 6. *Let p be a* SAL *program over a set $\mathcal{F}$ of procedures and a set $\mathcal{A}$ of advices. Let Γ be a specification table for $\mathcal{F}$ and Γ_a be a specification table for $\mathcal{A}$. Assume that for every procedure f that is advised by $\vec{a}_f$, the sequence $\vec{a}_f$ is specification preserving with respect to f, Γ and Γ_{a}. Then, if $f, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$, where Φ and Ψ are the pre and postconditions of f.*

The dynamic nature of some point-cut descriptors can make static verification a difficult task. Consider for example a cflow point-cut descriptor, for which program semantics must refer to a collecting call stack to decide whether a cflow condition is valid.

Although possible, it is cumbersome to reason explicitly about the call stack in the program logic. We propose, thus, the following simple derivation rule to reason in the presence of cflow point-cut descriptors:

$$\frac{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\} a\vec{a}' \triangleright f\{\Psi\} \qquad \Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a}' \triangleright f\{\Psi\}}{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\} a \overset{\mathsf{cflow}}{\triangleright} (\vec{a}' \triangleright f)\{\Psi\}}$$

where $a \overset{\mathsf{cflow}}{\triangleright} (\vec{a}' \triangleright f)$ denotes that the execution of the advice a is conditional on a cflow statement. The rule can be interpreted as the fact that the specification (Φ, Ψ) is still verifiable with respect to the sequence $a \overset{\mathsf{cflow}}{\triangleright} (\vec{a}' \triangleright f)$, regardless of whether the cflow condition is valid. Although incomplete, this rule may prove to be useful as long as the advice a is specification preserving with respect to (Φ, Ψ).

We have formally proved the soundness of the proof system proposed in this section. In addition, we have shown how to extended the compiler with a mechanism to translate a certificate of correctness of a SAL program to a certificate for the compiled code.

9. RELATED WORK

Reasoning about advices. As the invasive nature of aspects cause them to break modularity, the design of verification methods for AOP programs is challenging. Many works have explored the design space for such verification methods, and proposed different trade-offs between the modularity of verification and the generality of the method. In addition, there are been many works that isolate particular classes of aspects that are well-suited for modular reasoning and provide automatic analysis methods to detect when an advice fits in one of these classes.

Clifton and Leavens [9] define a notion of modular reasoning and show why modularity is not a general property in AspectJ. They define a classification for aspects as *spectators* or *assistants*: the former include aspects that only modify the state space they own and do not alter the control flow, whereas *assistants* can interfere with the original behavior of the program but only if explicitly accepted by the original program. Based on this classification, Clifton and Leavens suggest a verification method, detailed in [8]. More recently, Clifton, Leavens and Noble [10] have developed an effect system to verify the control and heap effect of aspects in the MAO language. The system verifies whether an advice is a spectator, and provides information exploitable by subsequent verification. To our best knowledge, there is however no sound program verification method based on these ideas. In a similar vein, Rinard *et al* [22] provide a a static analysis that automatically classifies aspects. They illustrate the usefulness of their analysis, but do not develop any verification mechanism based on it.

There have been several efforts to develop modular model-checking techniques for AOP. The prevailing trend to achieve modularity is to isolate specific classes of aspects that exhibit an appropriate behavior. Early work by Katz et al. [15] proposes a classification of aspects as *spectative*, *regulative* or *invasive*, and analyze the class of temporal properties that are preserved by aspects falling in these categories. In a subsequent work, Goldman and Katz [14] have formalized the idea that *weakly invasive* aspects preserve temporal properties. More recently, Djoko Djoko *et al* [12] have given a formal treatment of similar ideas based on a slightly different classification. These works resembles our own in the sense that they favor modularity of the verification process and makes emphasis on the preservation of original properties. Krishnamurthi *et al* [16] propose an alternative method where modularity is achieved by requiring that the set of point-cut designators is known statically.

Dantas and Walker [11] define the notion of *harmless advice*, which may preventing termination and may also perform I/O, but it does not interfere with the result of the baseline code. This weak interference property is an instance of specification-preserving advice, and thus permits to reason about the original program independently. They propose an information-flow type system over a core AOP language [23] to check harmlessness with respect to the main program. As discussed in Section 5.3, their type system can be combined to form part of our hybrid logic to certify and check that an advice does not interfere with the original global state.

Aldrich [1] has proposed a module system called "Open Modules" that enables class interfaces to explicitly control the visibility of internal control-flow points. Thus, it provides a mechanism to restrict the interference of external advice, by forbidding the attachment of advices to hidden internal join-points.

Proof compilation. There have been several efforts to study proof compilation for non-optimizing and optimizing compilers. Our work is most closely based on the work of [6], who show that a sufficiently simple compiler generates, from an imperative source program, a stack based low-level code, whose proof obligations are syntactically equal to that of the source program. Similar results are detailed by Pavlova [21], for a significant subset of Java Bytecode.

There has been a closely related effort by Zhao and Rinard [24] to provide state-of-the-art specification and verification tools for AOP, and to relate them to standard verification. They have defined Pipa [24], an extension to JML [17] for AspectJ [2], to support specification for aspects invariants, pre and postconditions for advices and variable introductions, and provided a compiler that transforms a Pipa-annotated AspectJ program into a JML-annotated Java pro-

gram. However, they do not provide any formal treatment to support their approach.

10. CONCLUSION

We have introduced the notion of specification-preserving advice, that mildly generalizes the notion of harmless advice of Dantas and Walker, and that is expressive enough to capture many advices related to security and efficiency. In addition, we have developed a modular verification method for programs with specification-preserving advices, and shown how proof compilation extends naturally to this setting. Our results, while preliminary, establish the feasibility of a Proof Carrying Code scenario with untrusted intermediaries modifying the code by aspects. In future work, we intend to build on proof compilation for Java and extend our results towards an expressive fragment of AspectJ, taking into account recent developments in optimizing compilation for aspects [3]. In addition, it would be interesting to target our compiler to low level languages with support for aspects [13], and investigate certificate translation in that setting.

11. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[2] AspectJ Team. The AspectJ programming guide. Version 1.5.3. Available from `http://eclipse.org/aspectj`, 2006.

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc* : An extensible aspectj compiler. 3880:293–334, 2006.

[4] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.

[5] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations for java. Draft paper, 2008.

[6] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, and S. A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.

[7] L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Symposium on Applied Computing*, pages 1835–1839. ACM Press, 2006.

[8] C. Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs.* Ph.d. thesis, Iowa State University, 2005.

[9] C. Clifton and G. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical report, Iowa State University, 2002.

[10] C. Clifton, G. T. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 451–475. Springer, 2007.

[11] D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM Press.

[12] S. Djoko Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 135–145, New York, NY, USA, 2008. ACM.

[13] R. M. Golbeck and G. Kiczales. A machine code model for efficient advice dispatch. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 2, New York, NY, USA, 2007. ACM.

[14] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect Languages Workshop (FOAL06)*, 2006.

[15] S. Katz. Aspect categories and classes of temporal properties. In A. Rashid and M. Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2006.

[16] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM Press.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, February 2007.

[18] P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, New York, NY, USA, 2007. ACM.

[19] G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[20] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.

[21] M. Pavlova. *Java bytecode verification and its applications.* Thése de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France, January 2007.

[22] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[23] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In C. Runciman and O. Shivers, editors, *ICFP*, pages 127–139. ACM, 2003.

[24] J. Zhao and M. C. Rinard. Pipa: A behavioral interface specification language for aspectj. In M. Pezzè, editor, *FASE*, volume 2621 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2003.

Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

Raffi Khatchadourian*
Computer Sc. and Eng.
Ohio State University
Columbus, OH, USA
khatchad@cse.ohio-state.edu

Johan Dovland
Department of Informatics
University of Oslo
Oslo, Norway
johand@ifi.uio.no

Neelam Soundarajan
Computer Sc. and Eng.
Ohio State University
Columbus, OH, USA
neelam@cse.ohio-state.edu

ABSTRACT

Reasoning, specification, and verification of Aspect-Oriented (AO) programs presents unique challenges especially as such programs evolve over time. Components, base-code and aspects alike, may be easily added, removed, interchanged, or presently unavailable at unpredictable frequencies. Consequently, modular reasoning of such programs is highly attractive as it enables tractable evolution, otherwise necessitating that the entire program be reexamined each time a component is changed. It is well known, however, that modular reasoning about AO programs is difficult. In this paper, we present our ongoing work in constructing a rely-guarantee style reasoning system for the Aspect-Oriented Programming (AOP) paradigm, adopting a trace-based approach to deal with the *plug-n-play* nature inherent to these programs, thus easing AOP evolution.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*; D.2.4 [**Software Engineering**]: Software Verification—*Formal methods*

General Terms

Languages, theory

Keywords

Aspect-oriented programming, modular reasoning, rely-guarantee

*A portion of this work was administered during this author's visit to the Computing Department, Lancaster University, United Kingdom.

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

1. INTRODUCTION

Aspect-oriented programming (AOP) [21] allows for modular implementations of crosscutting concerns. Since its inception, many authors [8, 24, 26, 33] have shown how aspects may be used to write localized implementations of important crosscutting concerns such as process synchronization, event logging, data persistence, exceptional situation handling, etc. The *separation of concerns* that AOP enables helps produce programs whose components are increasingly decoupled from one another as a direct consequence of the reduction of *scattered* and *tangled* code. As such, AO programs tend to enjoy *plug-n-play*-type capabilities where base and/or aspect components may be introduced, removed, and interchanged easily [24]. This inherent nature of AOP is beneficial in the sense that AO programs may evolve in a non-invasive fashion simply by "switching" features on and off. However, program evolution is bound to occur at unpredictable frequencies; therefore, programmers are often required to make key decisions and come to conclusions about a software components, base-code and aspect alike, utilizing either incomplete or highly volatile information at hand. Ergo, the ability to reason about individual AO program components modularly and to then compose these reasoning efforts, just as we would compose the components themselves, to obtain the actual behavior of the overall program becomes extremely desirable. This ability would permit AO programmers to avoid the unfortunate situation where the entire program must be reexamined upon each component change, thereby facilitating tractable evolution.

Despite its benefits, modular reasoning about AO programs indeed presents significant challenges [1, 3, 13, 5, 22, 23, 37, 29, 9, 38, 31, 12, 7, 34, 36, 30]. The problem is that, by circumscribing core concerns into classes and crosscutting concerns into aspects we are essentially creating *two* different systems, a *baseline* system (base-code) and an *augmented* system which is the result of applying aspects that alter the behavior of the baseline. Indeed, the ability of an aspect to change the behavior of the base-code that it advises, which is the very reason for much of the power of AOP, is also what causes difficulties for reasoning about the behavior of such software. In fact, as aspects "weave" in and out of (or "plugged" then "played") a software system, we may be forced to reason about the entire system, accounting for the interleaved execution of various pieces of advice with the base-code.

What we would aspire instead is to draw meaningful and useful conclusions about component code, e.g., base-code

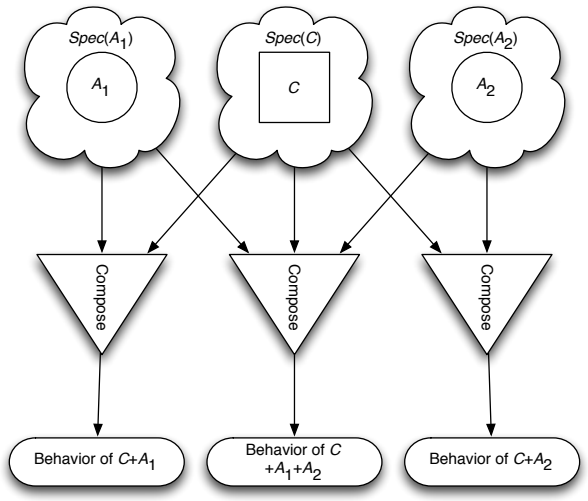

Figure 1: Schematic of behavior derivation using parameterized specifications of a component C under the influence of advice of A_1 and/or A_2.

which may reside in a method or an advice body that is itself subject to advice, *without* considering the actual advice code. Ideally, we would like to specify the behavior of AO components without any particular advice in mind such that in order to arrive at the behavior of the augmented system, just as aspects are *plugged-in* to enhance or *enrich* the behavior of the advised components, the specifications of applicable advice would be "plugged" into the matching behavioral specifications of the base-code. Furthermore, in order to arrive at useful conclusions that remain valid despite the addition of advice, it may be necessary to constrain possible advice behavior in order to preserve the intended semantics of the advised component. In other words, for a component to function correctly, assumptions may need to be made about potentially applicable advice such that these assumptions hold during evolution, with aspects entering, leaving, and *re*-entering the software.

Adopting such an idealized approach would allow developers as AO programs evolve to deduce the behavior of the augmented software without reexamining the *internals* of each component. In essence, the specified behavior of a component would be *parameterized* over the behaviors of all possibly applicable advice. Figure 1 helps to illustrate this notion, portraying schematically at a bird's-eye-view how these parameterized specifications can be hypothetically combined with the specifications of advice in order to obtain the overall system behavior, where $Spec(X)$ refers to the behavioral specifications of component X.

Although the above outlined approach may seem desirable, there are several key obstacles that must be overcome in its achievement:

Usefulness. As previously mentioned, we would like to draw useful conclusions about component code that is subject to the application of advice without considering the actual advice code. As we are focused on *evolving* AO software, the advice code may not yet exist; it may be added a later time. It is not clear, however, exactly how useful these conclusions can be considering that they should hold upon the application of any advice. What sorts of conclusions could we draw in this case?

Complexity. Since component specifications would be written in terms of any applicable advice, there is a strong possibility of these specifications becoming unwieldy. As such, any changes made to the *internals* (i.e., the implementation) of the advised component code would require a rather involved effort to rebuild the component's parameterized specifications. Also, situations may arise where a component `C` may not be under the influence of advice, yet *Spec*(`C`) would still be specified over all possibly applicable advice. Therefore, the complexity of the specification may be unnecessarily complicated. This situation is pictorially represented in 2 and further discussed later in this section. Making suitable restrictions on the behavior of potential advice via the use of language constructs, minimizing the join point model, providing behavioral constraining assertions by adapting a rely/guarantee [39, 18] methodology, which is the focus of our previous work [19, 35], and using predicates and/or functions on specifications themselves as in [16], which is a focus of our future work, may help alleviate several of these obstacles.

Obliviousness. Annotating the component code with parameterized specifications by very nature compromises the traditional *oblivious* [10] property intrinsic to AOP, in particular languages such as AspectJ [20]. Thus, by allowing AOP authors to construct their specifications in a parameterized fashion, and to further constrain the behavior of intangible advice (which would constitute the *actual* parameters), we are indeed forcing them to be at least cognitive of crosscutting concerns (CCCs). Nevertheless, it has been shown in [22] that even in (non-AOP) ordinary software, one must still be aware of CCCs, and [37] suggests that designing components subject to advice also requires the cognition of CCCs.

Modelling. How do we model specifications that abstract enough information from the internal details of components while simultaneously constraining the effects of potentially applicable advice that manipulates the internal implementation of these components? We will see later in this paper how our proposed approach, with the help of *traces*, may allow us to write such specifications for AOP.

Composition. Given the specifications of a component and its applicable aspects, how do we decide if the constituent advice is applicable especially considering that advice may be bound to lexical and dynamic pointcuts? A reasoning formalism should account for such situations if it intends to deal with lexical pointcut designators (LPCDs), e.g., `within()`, and/or dynamic pointcut designators (DPCDs), e.g., `cflow()`, `if()`. Then, given that advice is applicable to component code, how do we utilize the specifications of the advice and that of the parameterized component specifications to arrive at the overall behavior of the system, thus verifying that the software behaves as intended? Conversely, how do we derive the behavior of a component in which no advice is applicable given the nature of the components specifications? In fact, the schematic in Figure 1 is somewhat misleading as it fails to mention the situation where no advice is applicable. That is, in order to derive the behavior of a component that is not under the influence of advice we must either (i) obtain the behavior of a component `C` taking its parameterized specification and then composing it with a "empty" aspect specification $\mathtt{A}_{null}$ (portrayed in

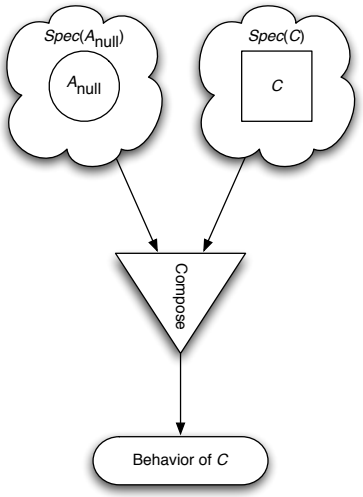

Figure 2: Schematic of behavior derivation using parameterized specifications of a component `C` not under the influence of advice.

Figure 2), or (ii) supply a second specification for each component which would correspond to the situation where no advice is applicable.

The focus of this paper is to (i) present new ideas in our ongoing work in this area, (ii) discuss our proposals to combat several of the above mentioned obstacles, and (iii) facilitate interesting discussion in respects to some of the open issues that faces our approach, in which we highlight throughout the paper.

2. CONSTRAINING AND ENRICHING THE BEHAVIOR OF PROGRAMS

Several approaches [1, 12, 13, 7] have been developed to restrict the behavior of AO components in order to reduce the efforts required in reasoning –both formally and informally– about such systems. In this paper, however, as mentioned in section 1, we are interested in specifying the intended runtime behavior of components under the possible influence of advice that accounts for the unique evolutionary nature of AOP; hence, assertions of these components should reflect this notion. Such a solution seems to call for a technique that would to need to be more flexible than existing approaches in the way that constraints on acting advice are expressed. In this section, we will briefly discuss how behavior of processes within concurrent programs are suitably constrained to achieve *interference freedom* using the rely/guarantee approach. This section will draw a parallel with concurrent programs and that of AOP, outlining our previous work in adapting the rely/guarantee approach for AOP. In addition, we will overview how assertions for evolving AO programs can be written, and how specifications can be composed to arrive at the *effective* behavior of the overall system, that is, the behavior of the components augmented with the behavior of applicable aspects.

Constraining behavior of concurrent programs. Consider a concurrent program with two processes P_1 and P_2 that share some variables that either of them may read and write. Standard modular reasoning would require us to reason about each process independently of the other and then combine the results of the two reasoning tasks in an appropriate manner to arrive at the behavior of the whole program, $[P_1//P_2]$. But since the two processes will be interleaved during execution, whatever conclusions we may have drawn about each of them when reasoning about them independently may not, in fact, be valid. In effect, the actions of each process may *interfere* with the other process thereby invalidating whatever results we may have established by reasoning about that other process.

The *rely-guarantee* approach [18, 39] addresses the problem of interference in concurrent programs as follows. Let σ be the state, i.e., the set of all program variables of the program consisting of two processes P_1 and P_2 running in parallel. When reasoning about P_1, we recognize that the actions of P_2 may modify the state. Hence, we write our assertions in the proof outline of P_1 in such a manner that they continue to be satisfied even in the presence of such actions. To enable this, we identify a relation $rely_1()$ that is a predicate over two states, σ_a and σ_b. This relation means the following: suppose at some point in the execution of P_1 the current state is σ_a and that some part of P_2 is now interleaved in the execution; suppose that the state when P_1 gets control back is σ_b; then $rely_1(\sigma_a, \sigma_b)$ must be satisfied. In other words, when reasoning about the behavior of P_1, we assume that any interleaved action that P_2 (or any other process in the case of programs with more than two processes) may change the state but *only within the constraints* specified by $rely_1()$. If this is satisfied, the correctness of the proof outline of P_1 will not be affected by the actions of P_2. Conversely, when reasoning about P_2, we introduce a relation $rely_2()$ that imposes constraints on the changes in the state that may be caused by P_1's actions.

Next, we must verify that P_2 and P_1 meet the requirements contained respectively in $rely_1()$ and $rely_2()$. To make this possible, when reasoning about each process, we establish a *guarantee* clause. This clause, denoted $guar_1()$ in the case of P_1, is again a relation over two states; it is a guarantee provided by P_1 that any change it makes in the state when executing any instruction in it will obey the constraints specified in $guar_1()$. The specification of P_1 is of the form $(pre_1, rely_1, guar_1, post_1)$ which denotes: if P_1 starts in a state that satisfies pre_1 and if all transitions, i.e., state changes, made by P_2 satisfy the constraints specified in $rely_1()$; then each transition made by P_1 will satisfy the constraints specified in $guar_1()$, and the state, when P_1 finishes execution, will satisfy $post_1()$. The *parallel composition* rule requires us to check, using $guar_1()$ and $guar_2()$, that the *rely* clauses of both processes are satisfied.

Constraining behavior of AO programs. As we noted earlier, reasoning in concurrent programs seems to have some resemblance to reasoning about AOP. Suppose, for example, that the code of a component, say a class `C`, subject to advice contains an assignment statement assigning a value v returned from a method call `m()` on an object `obj` to a particular instance variable `x` of `C`. When reasoning about the code of `C`, we might have established an assertion following the assignment that states that the value of `C.x` would be equal to v. Suppose now that an aspect is added that encompasses a piece of *after*-advice that applies at the *call*-join point associated with the invocation of `obj.m()`. Immediately following the execution of the assignment of the returned value v to

C.x, the *after*-advice would execute and, possibly, invoke a mutator method of C that assigns a new value to C.x. When control returns to C, at this point, the assertion we previously established may no longer satisfied. In other words, the aspect has *interfered* with the component code.

While this seems highly analogous to the case of the concurrent program, there are notable differences between the situation in AOP and that of concurrent programs. Firstly, AO programs are intrinsically sequential, making the interleaving of their constituent statements more predictable. Nevertheless, when reasoning about components independently we must recognize that advice may be weaved in, out, or around each join point, thereby detracting this otherwise innate predictability. Secondly, the severity of possible interference is governed by the join point model of the underlying AO language. Possible interference is thus dictated by the types of join points, the control structures, and the mutable contexts that are exposed and available to advice to manipulate. Thirdly, there is an asymmetry between components that does not possess the ability to advise other components, e.g., a method within a class in AspectJ (sans annotation-based mechanisms such as @AspectJ), and components, e.g., aspects, that do possess this ability. In particular, while advice can *intercept* the execution of a class, a class does not intercept the advice. As such, control is solely at the mercy of aspects as opposed to other paradigms like concurrent programming, and coroutines [6] where control is explicitly released and suppressed at various points. Lastly, in the case of parallel programs, concurrent processes are typically designed hand-in-hand, while in AO programs aspects may be added, removed, and/or changed to a system at unpredictable intervals as the software evolves.

A key observation underlying our approach is that assertions contained within component subject to advice should be in the form of a relation over two states σ_a and σ_b. Here, σ_a refers to the state of the advised component prior to control transferring to advice, and σ_b refers to the state of the advised component immediately following the point where it reacquires control. Therefore, components subject to advice can effectively detail the sorts of constraints on advice behavior required for it to behave properly regardless if advice is applicable at the moment. Principally, when reasoning about a component C we recognize that its behavior may be modified as a result of aspect(s) being applied to it. As C executes, if control were to reach a *join point* that matches a *pointcut* at which a particular advice is applicable, control will transfer to the advice before, after, or around (potentially bypassing) the statement at that point. The advice would then execute, possibly changing the values of some of the instance variables of C and/or other accessible parameters. Finally, control would then return to C which would continue execution.

The approach discussed in this paper is based on augmenting leverages a existing technique made for improving modularity in AO programs. We extend the notion of *pointcut interfaces* [13] by annotating pointcuts with associated specifications that must be met during the execution of the matching join points by both the component (through a *guar* clause) and applicable advice (through a *rely* clause). We will discuss related work in more detail in section 4, even so, it is worth mentioning here that the contractual obligations between advice code and *advised* code is similar in spirit to Crosscutting Interfaces (XPIs) [12], however, our interest lies in establishing run time *behavioral* properties exhibited at compile time, i.e., through use of an axiomatic proof method.

Deriving effective behavior. Unlike concurrent programs where a prime concern is preserving process interference freedom [32], the of addition of aspects typically corresponds to *enriching* existing program behavior. Indeed, it is the possibility of such enrichment that is the source of much of AOP's power. For this purpose, we introduce the concept of *join point traces* (*JPT*s). A JPT is used when reasoning about a component C under the potential influence of advice, to record the flow-of-control through various join points contained within C. These join points are the ones "exposed" by the pointcut interface of C, where items of advice may be applied to enrich or otherwise affect its behavior. We will delve into the details of the structure of JPTs in section 3, but the central idea is to specify the behavior of C in terms of assertions involving not just the variables of C but also *abstractly* in terms of the state changes caused by various items of advice that could possibly be applied at the various join points recorded in C's JPT, *without* referencing the actual advice. When reasoning about C, we will not, of course, know what these state changes will be since the aspect(s) in question may not yet have been constructed (or even if they have been, we have not yet reasoned about them). Hence, in our reasoning, we have to allow for a range of possible state changes –subject to the constraints of the appropriate *rely*() clauses– that these items of advice may carry out; essentially the assertions characterizing the behavior of C will allow for various such changes and, corresponding to each, specify how C will behave. In effect, the behavior of C will be *parameterized* with respect to the possible behaviors that each item of advice code may engage in at the various join points, with the JPT being used to record the "parameter values" representing these behaviors. The next step, given a particular set of advice specifications, is to *compose* our JPT-based specification of C with the specified behaviors of the aspects to arrive at the resulting enriched behavior of the composed system as illustrated earlier in Figures 1 and 2. Formally, this will be carried out by appealing to our rule of composition of aspect and the components they advise.

3. SPECIFICATION AND VERIFICATION

In this section, we will explore possible ways to specify and curtail the behavior of AO programs in order to improve reasoning in these systems that is natural to the way they evolve, intuitively similar to what is portrayed in Figure 1. We will then examine several inference rules using a highly distilled version of an AspectJ-like AO language that will allow us to show that the *composition* of AO components meets a certain specification. Our goal in this paper is not to provide a complete formal set of rules but rather to indicate the types of considerations involved in them. In future work we intend to define the syntax for a complete but simplified version of AspectJ, present its operational semantics, extend our set of proof rules to apply to this language, define a formal operational model based on the notion of JPTs, and address questions about soundness and completeness of the rules with respect to the model.

Specifications and pointcuts. To demonstrate the crux of our proposal, we will only consider *call*-join points and *after* advice. *Before* advice could be theoretically handled in a symmetric manor; *around* advice, however, poses some in-

teresting complications as advice could alter both the calling and callee objects and avoid the execution at the join point entirely by opting not to call `proceed()`. We leave *around* advice as a problem open to discussion with the possibility of leveraging existing work from [4]. For further simplicity of the presentation, we will also not consider such constructs as lexical pointcut designators (LPCDs) but will consider them in future work.

The flexibility and expressiveness that we desire with our specifications may lead to undesired complexity since many join points may be traversed as a result of a given method invocation. This complexity, however, depends heavily on the strength of the associated *rely* clauses as any applicable advice must respect it. Thus, although our specifications will be over the behavior any applicable advice, we do not need to consider potential behavior that does not abide the *rely* assertion. Another possibility would be to follow the conventions in [1], allowing only external calls to methods within a component `C` listed on the interface of `C` to be subject to advice, that is, `C` is "sealed." Pointcuts appear on `C`'s interface in order to export important internal events within `C` that the author of the component feels aspects may be interested in advising. Note that importantly the author does not examine existing aspects to come to this decision, instead, she solely examines `C` to determine which internal events should be exported on its interface. In much the same way, in the context of our proposal, the author of the component determines the necessary constraints to place on possible advice that would apply to the it based on the internals of that component alone, deciding what essential constraints are necessary to place on the behavior of advice either currently in existence or to be developed in the future. Moreover, as another possible extension to our approach, it may be worthwhile to break the sealing of a component for *observer* aspects [3]. That way, less intrusive aspects, e.g., logging aspects, would be allowed to advise the execution of the entire program. We leave both of these issues open for discussion.

A common challenge with providing a reasoning scheme for software that contains objects (and aspects) is aliasing, which tends to cloud the vision as to what an object's state precisely consists of. A component `C`, say a class for instance, will in general define a number of (instance) variables. Some of these will be of primitive types (`int`, `boolean`, etc.), others will be of reference types. Consider an instance `obj` of `C`. The state of `obj` at any time will consist of the values of the variables of primitive types plus the values of references to objects. Generally we will not consider the states of objects that `obj` contains references to as part of the state of `obj`. Only changes to the values of its primitive variables resulting from execution of methods invoked on `obj` will be reflected as changes in the state of `obj`. As these methods execute, they will in general invoke methods on objects that `obj` has references to, resulting in changes in the states of those objects; these latter changes are not part of the changes in the state of `obj`. In effect, we are assuming that there is a *heap* in the background that holds all the objects, retains their current states, and makes them available to us as needed. These considerations are, of course, common to reasoning frameworks for all object-oriented languages. Hence we will not consider them in any detail when presenting our formalism.

Join point traces. From this point forward we will typically consider the situation where we wish to specify and verify a method in a class that may potentially be under the influence of advice. It is not inconceivable to conversely apply our proposal to reasoning about a piece of advice in an aspect which itself may be open to advice (possibly even itself), however, as noted earlier, *around* advice does pose several interesting problems (e.g., constraining the behavior of calls to `proceed()`). For now, however, consider, in general, a method `m()` of a class `C`. The JPT for this method will record the flow-of-control through the various join points in the body of `m()` where items of advice defined in various (perhaps yet-to-be-developed) aspects may apply. Each join point is a call to a method either of the same class `C` or of a different class. Note that we do not have to worry about *every* call that appears in the body of `m()`. Suppose there is a call to a method `n()` of a class `D`. If `n()` appears in an exported pointcut on the pointcut interface of `D`, then this call, in our current approach, will be recorded on the JPT of `m()`. If not, however, it will not be record since this call in this case would have no advice applicable to it. The designers of `D` are responsible for deciding whether or not calls to `D.n()` should be included in one of the pointcuts of `D`. Our specifications, using JPTs, will reflect these decisions.

Traces of various kinds have been widely used for specifying the behaviors of different types of systems ranging from ADTs [16, 17] to processes in a distributed system [15, 28]. In each case, elements in the traces are used to record information about important events in the system; the ordering of the elements in the trace represents the order in which the corresponding events took place. Specifications of the systems are written in terms of conditions that must be satisfied by the structure of the trace and by the information recorded in the individual elements. In the case of JPTs, the events of interest are the arrival of control at various join points. Since the only kind of join point we are considering is the call-join point and since the only type of advice we are considering is *after* advice, each element of the JPT will correspond to the completion of a call. In order to deal with DPCDs such as `cflow` ad `cflowbelow`, it is also convenient, from the point of view of specifying our methods, to record on the JPT the events corresponding to the *start* of method execution as well as its end.

Let us now consider the structure of the elements that appear in a JPT. Consider the completion of a call of a method `m()` of a class `C` invoked on object `obj`. Suppose we have a pointcut `pc` defined in the pointcut interface of `C` that includes calls to `m()`. Suppose `A` is an aspect that includes an (*after*) advice that applies to this pointcut. We will assume that any variables defined in `A` will be of primitive types. The code of the advice may update the values of these variables and also update the state of `obj` by updating values of the primitive variables in that state. For simplicity in the presentation, in this paper, we will not consider the possibility of `A` invoking additional methods. Trace models are, in general, powerful enough to handle such complications but the resulting specifications tend to be rather complex. The exclusion of such calls means we do not have to worry about additional items of advice associated with calls to such methods being triggered. Nevertheless, we encourage open discussion on how some of these restrictions may be relaxed.

During actual execution of the system, if an aspect such as `A` considered above had been defined, control will transfer to the corresponding advice code. That code will execute,

possibly resulting in changes in the state of the aspect as well as in the state of `obj` (e.g., through exposing context at the join point using the `target()` PCD), or even the caller of `obj.m()` (e.g., through exposing context at the join point using the `this()` PCD). Hence, in this element of the JPT, we will record the state of `obj` at the time that control reached the completion of the call to `obj.m()` and its state when control returns from the advice, and likewise for the calling object of `obj.m()`. If there is no applicable advice, either because calls to `obj.m()` are not included in any pointcut, no aspect such as `A` had been defined to apply at the pointcut, or because the conditions for the advice to be applied are not satisfied, these two states will be identical, since the state will not, in this case, change between the time that the call completes and the code of the calling method continues execution.

However, the completion of the call to `obj.m()` is not the only point at which advice may apply. That is, calls made to additional methods within the body of `C.m()` may themselves be subject to advice. As such, the specifications of these method calls will also be parameterized over applicable aspects. Thus, although until now we have been considering an individual method `m()` of a class `C` and described the JPT as if it corresponded to just (a single call to) this method, there is, in fact, a single JPT for the *entire system*. This JPT is initialized, at the start of the system's execution, to the empty sequence. We will use the symbol $\gamma\tau$ to denote this (global) trace. Each time a method is called, an element is added to the JPT to record the start of this invocation. And when the method call completes, an element corresponding to the completion is added to the JPT, in fact, this element corresponds to the join point at that location. The effect of any (*after*) advice that applies at this call-join point is recorded in this latter element. The completion elements of the JPT will have the structure $(oid, mid, aid, args, res, \sigma, \sigma')$ where *mid* is the identity of the method whose call just completed; *oid* is the identity of the object on which the method *mid* was invoked; *aid* is the identity of an applicable aspect instance, *args* and *res* are the arguments and result (if any) of this method. σ and σ' are state vectors as follows. $\sigma[oid]$ is the state of the callee object at the time the method call completed (immediately prior to transferring control to the advice if any) and $\sigma'[oid]$ the state at the time immediately following the completion of the advice code. Likewise, $\sigma[\texttt{this}]$ is the state of the calling object at the time the method call completed and $\sigma'[\texttt{this}]$ the state at the time immediately following the execution of advice. If there is no applicable advice at this join point, then $\sigma[\texttt{this}] = \sigma'[\texttt{this}]$. These state vectors also contain the state of aspects in a similar fashion. Specifically, $\sigma[aid]$ refers to the the *aspect state* immediately following the completion of the execution of the method, whereas $\sigma'[aid]$ is the aspect state immediately following the completion of the advice code. If there is no applicable advice at this join point, these elements will not be included. We should note again that the above description is an *operational* picture of the JPT.

Let us now consider the structure of the elements of the JPT that record method call invocations. Consider again the call of a method `m()` invoked on object `obj`. If we wanted to account for *before* advice that might apply at this point, the element of the JPT recording this invocation would have to include information very similar to the above. In this paper though, we are only considering `after` advice, hence we can omit much of this information. The only information that does need to be included are the identities of `m()` and `obj` since these may be used to control the applicability of some advice, especially those point to pointcut expressions containing DPCDs.

Inference rules. We will consider three rules corresponding respectively to accounting for the advice that may apply when a method call completes; for the call a method body makes to another method; and for combining the specification of a method `C.m()` with that of the aspects that apply, including those that apply to the various methods that `m()` calls during its executions to arrive at the resulting "enriched" behavior of `m()`. Let us first consider the rule depicted in Figure 3 corresponding to completion of a method call to `C.m()`. The specification of such a method will be a 4-tuple, $(pre, post, guar, rely)$. Here, *guar* is obtained from the conjunction of each *guar* clause of each exported pointcut which corresponds to calls to `C.m()`. *rely* is obtained in a similar manor except that it is derived from the disjunction of each *rely* clause. This specification annotation means that if *pre* is satisfied when `C.m()` is called; and if the methods called in the body of `C.m()` satisfy their respective specifications –these calls may be to methods of `C` or other classes or both–; and if any advice applicable to calls to `C.m()` satisfy the *rely* clause; then when body of `C.m()` finishes execution, it will satisfy *post* as well as the requirements specified in the *guar* clause. Note that the post-condition will, in general, involve the JPT since that is what will allow us to later enrich this specification, accounting for the action of advice defined in any aspect that may be developed to apply to calls to `C.m()`. But this JPT is not the global JPT, $\gamma\tau$; rather, it corresponds to a single execution of this method and we will use the symbol $\lambda\tau$ to refer to it.

$$\frac{\begin{array}{c} pre \wedge [\lambda\tau = \langle (inv, \texttt{C.m}) \rangle] \Rightarrow p \\ \{p\} S \{q\} \\ q \Rightarrow guar(\sigma[\texttt{this}]) \\ \big[q \wedge rely(\sigma[\texttt{this}], \sigma'[\texttt{this}])\big] \Rightarrow \\ post[\lambda\tau \leftarrow \lambda\tau^\frown(\texttt{this}, \texttt{C.m}, ?, args, res, \sigma, \sigma'),\ \sigma \leftarrow \sigma'] \end{array}}{\texttt{C.m} :: \langle pre,\ post,\ guar,\ rely \rangle}$$

Figure 3: Rule for method specification.

S is the body of the method, with pre-condition p. This differs from *pre* since *pre* does not give us any information about $\lambda\tau$; thus the first line essentially tells us that when the method body starts execution, $\lambda\tau$ has been initialized to contain the element representing the start of the invocation (*inv*) of this method (`C.m()`. The post-condition of S is as denoted q. The second line of the rule requires us to show that when S completes, q will indeed hold. Moreover, `C.m()` will, in general, provide a guarantee to any advice that may apply when the call to `C.m()` completes. This guarantee is represented by *guar* and this has to be satisfied when S finishes; i.e., the post-condition q of S must imply this assertion.

The next requirement, split over the next two lines of the rule, essentially allow us to go from the post-condition q of the body of the method to *post*, the post-condition of the method. The difference between these two assertions arises

because the JPT has an extra element added to it to represent the completion of this method; and the state might be modified from σ to σ' as a result of an advice that has been defined to apply at the completion of a call to this method. Any such advice is required, as indicated in the third line, to satisfy the *rely* clause. The element being appended to $\lambda\tau$ at this point corresponds to the completion of this method call. The first element denotes the fact that the object in question is simply the *this* object. The *aid* element represents the identity of the aspect, if any, that may apply at this point. Since we are only considering the method at this point, we have no requirements with respect to an aspect state, hence the question mark. But the state of the current object may itself change; this change is represented by the elements denoted $\sigma[\texttt{this}]$ and $\sigma'[\texttt{this}]$; these elements may, as just noted, be assumed to satisfy the *rely* clause of this method (since if not, the aspect is considered unacceptable). Once we have added this method completion element to the JPT, we change the state of the object to whatever the aspect assigns to it. The rule requires us to show that, following this assignment, *post*, the specified post-condition of the method is satisfied.

The rule looks rather involved but much of it is *notational* complexity. Intuitively, the rule may be summarized by saying, it requires us to show that the body S of `C.m()` behaves according to its specification; and that when S finishes, the state satisfies the requirements specified by the *guar* clause so any *after* advice that is defined can legitimately assume this clause. The rule also requires us to take account of the fact that when the method is invoked, before the body starts execution, the trace must be appropriately initialized. And when the method finishes, the trace must be finalized by adding a completion element that also records the effect of any advice that may be applied corresponding to calls to this method.

Next let us consider the rule portrayed in Figure 4 for dealing with a method call. Suppose the method invocation is of `obj.m(args)` where `m()` is a method of the class `C`. Let us assume that the specification of `C.m()` is $\langle pre, post, guar, rely\rangle$. Let us further assume that the (local) JPTs for the calling and called methods are named $\lambda\tau_1$ and $\lambda\tau_2$, respectively. At the start of the method, we have to account for the fact that the object `obj` of the calling method will play the role of the `this` object of the called method and substitute the actual arguments for the formal parameters of `C.m()`. We let $p[\overline{x}/\overline{e}]$ denote p where all free occurrences of a variable in the list $\overline{x}$ are replaced by its respective expression in $\overline{e}$. When the method finishes, we have to substitute in the reverse direction *and* append $\lambda\tau_2$ of `C.m()` to $\lambda\tau_1$ of the caller.

$$\frac{p \Rightarrow \texttt{C.m}.pre[pars/args] \qquad \texttt{C.m}.post \Rightarrow q[\lambda\tau 1/\lambda\tau 1\hat{}\lambda\tau 2,\ args/pars]}{\{\ p\ \}\ \texttt{obj.m(args)}\ \{\ q\ \}}$$

Figure 4: Rule for method call

Essentially, the JPT records appropriate information about the various (potential) join points through which control flows. The value of $\lambda\tau_1$ at the time of the method call, i.e., just prior to control being transferred to `C.m()`, represents all join points we have encountered thus far in the calling method. Now control continues in the body of `C.m()`. As that body is executed, additional methods may be called and information about the corresponding call-join points should be accumulated in $\lambda\tau_2$, the local JPT of `C.m()`. But this is, in fact, simply a record of the additional join points that we are encountering as execution continues. Hence, as control returns from `C.m()` to its caller, we need to append this record to the JPT that we already had immediately prior to the call, i.e., to $\lambda\tau_1$. This will ensure that, in $\lambda\tau_1$, we will have a *complete* record of the control-flow along join points that occurs during the entire execution of the caller of `C.m()`, *including* the flow that occurs during the execution of the methods that are called.

The final rule (Figure 5) we consider is for *applying* an aspect to a class, in particular to a class method, to arrive at the resulting enriched behavior of the method. More precisely, this aspect has been defined to apply at a pointcut that includes the call join point to `C.m()` and we want to arrive at the enriched behavior of this method as a result of this aspect. Let us first consider a simpler form of the rule ignoring the possibility of DPCDs. `A` refers to the aspect being applied and $\texttt{A}_{adv}$ is the applied advice defined in the aspect.

$$\frac{\{\ guar(\sigma) \wedge ap\ \}\ \texttt{A}_{adv}\ \{rely[\sigma/\sigma@pre, \sigma'/\sigma] \wedge\ aq\ \} \qquad \texttt{C.m} :: \langle pre,\ post,\ guar,\ rely\ \rangle}{\{pre \wedge ap\ \}\ \texttt{C.m}() + \texttt{A}\ \{post \wedge aq\ \}}$$

Figure 5: Rule for aspect application (simple version)

In this rule, the *rely* and *guar* clauses are part of the specification of `C.m()`. The $\sigma@pre$ notation denotes the state at the start of the execution of this code. The first line thus requires us to check that this code meets the *rely* and *guar* clauses. In particular, the post-condition of $\texttt{A}_{adv}$ ensures that the *rely* clause with $\sigma@pre$ playing the role of the starting state of the clause and σ playing the role of ending state, is satisfied. The *ap* and *aq* are assertions over the state of the *aspect*. The second line simply states that we have already established the required result about `C.m()`. The post-condition in the conclusion of the rule shows us the effect of the enrichment resulting from the application of the aspect.

There are some problems with this rule. First, the pre-condition requires not only the expected pre-condition of the *method* is satisfied but also, *ap*, which is a condition over the aspect state. The value of this state would not be affected by the execution of the body of `C.m()`, so if it is satisfied at the start of the execution of this body, it will also be satisfied when the code $\texttt{A}_{adv}$ starts execution. But how will we ensure that this condition is, in fact, satisfied at the start of `C.m()`? This state is going to be modified only as a result of execution of various pieces of advice code of this aspect, not by the body of `C.m()`. Nevertheless, there is nothing in the pre-condition of `C.m()`, as we have it so far, that will ensure that the aspect state, as it existed at the start of `C.m()`, satisfies this assertion. Hence we need to add this as an additional part of the pre-condition of `C.m()`. Therefore, this information needs to be provided as part of the *invocation* element that is added to the JPT at the start

of C.m().

However, although the execution of the body of C.m() will not directly modify the state of the aspect, there may be calls in this body to other methods, and those methods might be subject to this same aspect as well. Thus the state of the aspect when the execution of the body of C.m() completes and control is transferred to the advice code may not be the same as it was when C.m() started execution. Instead, it will be whatever it was when the most recent such call finished execution. These calls will, of course, be recorded on the JPT as per our method call rule. Further, the effect of the advice acting on the called methods will result in the aspect state that exists at the end of each such call to be recorded on the JPT. We can address these considerations by making two changes to the above rule. First, we modify the pre-condition of A so that the assertion *ap* applies to the state of the aspect as recorded in the invocation element of the JPT. Second, we need to modify the post-condition so that the assertion *aq* applies to the (final) aspect state recorded in the JPT when this method returns to its caller.

The third problem with the rule has to do with bound pointcut expressions containing DPCDs. The rule above assumes that the advice code $\mathtt{A}_{adv}$ will apply to the execution join point of C.m(). But it may not. Or, rather, we may have a condition that depends on the call stack (as is the case with such DPCDs as `cflow`) that will determine whether or not it is applicable. And this, of course, cannot depend on anything that is contained in the body of C.m() nor can it be specified as part of the *pre-condition* of C.m(). Instead, it will depend on the state of the call stack for each call to C.m() that we have to deal with in reasoning about the behavior of the overall system. In order to handle this, when reasoning about C.m(), if the pointcut associated with this advice is dynamic, we will allow for both possibilities – when the associated condition *is* satisfied and when it is *not* satisfied. For this purpose, we will have *two* (possibly) distinct post-conditions with the method, corresponding respectively to the cases when the method is called with the state of the call stack satisfying the condition of the dynamic pointcut and when this condition is not satisfied. These are marked, in the rule depicted in Figure 6, with the labels d and $\neg d$ respectively, d being the condition specified in the dynamic pointcut for deciding whether or not the advice should apply when the execution of C.m() finishes.

$$\frac{\{\ guar(\sigma) \wedge ap\ \}\ \mathtt{A}_{adv}\ \{rely[\sigma/\sigma@pre, \sigma'/\sigma] \wedge\ aq\ \} \quad \mathtt{C.m} :: \langle pre,\ post,\ guar,\ rely\ \rangle}{\{pre \wedge ap\ \}\ \mathtt{C.m}() + \mathtt{A}\ \{\langle d : post \wedge aq\ ,\ \neg d : post \wedge ap \rangle\ \}}$$

Figure 6: Rule for aspect application (revised version)

There is one final complication – the combination of the two problems identified above. That is, the items of advice applicable to methods called within the body of C.m() may *also* have dynamic pointcuts associated with them! This means the effect of dynamic pointcuts is not going to result in just *two* possibilities in the post-condition of C.m() but rather all possible combinations of the conditions corresponding to these various dynamic pointcuts being or not being satisfied! In the worst case, this would give rise to 2^n possibilities, n being the number of calls in the body of C.m(). What this tells us is that while dynamic pointcuts are undoubtedly powerful, using them too liberally can lead to systems that are extremely difficult to specify or reason about. In fact, it is precisely this problem that forced Krishnamurthi *et al.* [23], in their model checking approach, to introduce a *depth* parameter that is used as threshold to combat this explosion. This problem is indeed more general as illustrated above as it would also be encountered when a join point resides either in a loop or is traversed multiple times as a result of recursion. We will not present a formal version of the rule that accounts for this problem as a solution is currently being investigated.

4. RELATED WORK

Several authors have proposed restrictions to AOP in order to address the complexity of the associated reasoning. Clifton and Leavens [5] present MAO, a language that extends AspectJ [20] with concern domains and control-limited advice. MAO, via static analysis, allows developers to restrict the behavior of advice, e.g., to allow accesses to only certain parts of the heap belonging to a particular *concern domain*. MAO also allows for restricting the manipulation of control-flow by advice thereby forbidding it to perturb the control-flow of the base-code in inappropriate ways. Such restrictions are expected to help simplify reasoning about AOP since developers can examine the *signatures* of each advice declaration to reason about its potential effects. Similar restrictions are also conceivable using our proposed *rely* clauses; however, our advice restrictions are more flexible and fine-grained in that *rely* clauses take an arbitrary assertion over two states σ and σ', the state at the point in which advice obtained control and the state corresponding to the point when it released it, respectively. Furthermore, MAO does not provide the proper facilities to *combine* the specifications of the base-code and the advice to arrive at the *overall* behavior exhibited by the augmented system. Nevertheless, it should be possible to borrow some of MAO's ideas to help simplify our formalism.

Dantas and Walker [7] propose "Harmless Advice," a restricted form of advice that has minimal effects on the base-code, and develop a type system that enforces such behavior statically. Harmless advice cannot alter state (with the exception of I/O) and control-flow that is visible to the base-code. What is interesting about such advice is that, although highly constrained, it is shown to be quite useful especially in the domain of security. With the use of *rely* clauses, our approach could conceivably be adopted to relax some of the constraints on harmless advice in order to make it more "helpful" while maintaining effective local reasoning. This would allow the base-code developer to explicitly state on a fine-grained level what kinds of advice behavior he or she considers "harmless" by means of a less restrictive *rely* assertion, and then use JPTs to reason about the overall effects of the advice applied to the base-code.

Krishnamurthi *et al.* [23] propose a verification technique which can, using model-checking [2], modularly verify advice independent of the base-code. The proposal, given the base-code represented as a finite-state model, a set of properties that the augmented system (i.e., the base-code combined with the aspects) must satisfy, and a set of pointcuts where potential advice may be applicable, automatically generates enhanced interfaces which can be used for verifying the ad-

vice when it becomes available. Essentially, the interface captures the state of the model checking process prior to advice being added to the system. Goldman and Katz [11] present a related technique using their MAVEN tool. While these approaches, as well as the approach presented in this paper, all employ techniques that do not require repeated analysis of the entire augmented system each time a developer adds, removes, or changes advice, there are several key differences. Firstly, our proposed proof technique relies on deductive logical reasoning while model-checking entails a fundamentally different approach in which an abstract model is exhaustively examined for violations of a certain property. Furthermore, our approach is centered on combining the specifications of the base-code and that of the aspects using JPTs in order to assist developers in obtaining the overall behavior of the system. As such, our proposal does not require a specific property that neither the base-code nor the augmented system must exhibit.

Devereux [9] also attempts to exploit the similarities between AOP and concurrent programs. The approach translates an aspect-oriented program into to an equivalent, low-level concurrent program in an alternating-time logic formalism. The reasoning then is performed on this concurrent program using an assume-guarantee paradigm [14]. The modus operandi is focused on preserving particular properties of the base-code despite the addition of advice. Our approach, through the use of JPTs, on the other hand, allows a developer to reason about the behavior of the base-code *parameterized* over any applicable aspect; therefore, reasoning about the base-code does not need to be reconstructed for *each* property being verified nor a specific property that the base-code must evince. We are interesting in obtaining the enriched behavior of the combined system as opposed to solely verifying the existence of interference freedom [32]. Moreover, transformation from an aspect-oriented program to a concurrent program may cause the task of reasoning about the original program to be more difficult. That is, a change to either the base-code or an aspect could possibly result in previous reasoning efforts being invalidated. Also, assume-guarantee reasoning in concurrent programs are normally leveraged with the acknowledgement that other processes may exist in the system. In AOP, nevertheless, aspects may not even have been developed yet or may be interchanged between different systems. As such, the proposal presented in this paper is designed more towards how AOP is used, especially to the *plug-n-play* capability inherent to aspects.

Several approaches [1, 12, 13, 25] attempt to augment traditional interfaces with various degrees of information regarding crosscutting concerns in order to improve reasoning. In particular, Kiczales and Mezini [22] argue that in the presence of crosscutting concerns we cannot expect to work with the standard interfaces provided by a class' methods and their behaviors. Instead, we must define a more detailed interface for the class that includes information pertaining to how the system is intended to be deployed. These *aspect-aware* interfaces, which include the various join points at which advices defined in the aspects are applicable, accompany traditional interfaces, thus adding to their usefulness.

5. FUTURE WORK AND CONCLUSION

Reasoning, specification, and verification of AO programs indeed presents unique challenges especially as such programs evolve over time. Constructing an approach general enough to reason about components subject to the unpredictable frequency of advice applicability poses many obstacles including but not limited to usefulness, complexity, obliviousness, abstraction, and composition. In this paper, we have presented our ongoing work in developing such a technique that attempts to overcome these obstacles in an effort to enable tractable evolution of AOP. We propose an approach that is aimed at tailoring specifications of these systems to their evolutionary *plug-n-play* nature and enhancing the expressiveness of constraints made on their constituent components.

In future work we intend to extend our set of proof rules to account for many additional AOP mechanisms. We also intend to define a formal operational model based on the notion of JPTs and address questions about soundness and completeness of the rules with respect to that model. One interesting direction for further work is to investigate multiple aspect instances as provided by the *association* facilities (e.g., `perInstance`) of AspectJ. The close relation between an aspect instance and an object should be reflected in reasoning mechanisms expressing the tight connection between the object state and the state of the aspect instance. We also plan to address mechanisms for member introduction and class hierarchy modifications, possibly utilizing techniques employed in [27]. Another possible avenue to explore is the notion of *specification weaving* as it may help in prevailing over some of the aforementioned hurdles. Additionally, we have listed several unresolved issues below in hopes of provoking interested and related discussion.

Execution-join points vs. call-join points: Suppose we have two classes `C` and `D` and there is a call in the body of `C.m`() to the method `D.m′`(). Further suppose there is an aspect that contains advice that applies (on calls to) `D.m′`(). When reasoning about what the advice code does, we are allowed to assume the *guar* clause given to us as described in section 3, but how it is exactly derived is not yet clear. It seems it should solely be from `D.m′`(). But we discuss how, at various points in the *body* of a method, the *rely*/*guar* clause can be assumed (for *rely*) or must be shown hold (for *guar*). That would mean we are referring to `C.m`(). But the advice that applies to the call-join point associated with the call to `D.m′`() is not concerned with `C.m`(); it is concerned with `D.m′`(). Furthermore, it is not concerned with what happens *inside* the body of `D.m′`() because it is associated with the call-join point, meaning that the question is only about the state at the *end* of `D.m′`(). Thus, is there a need to consider the *rely*/*guar* clauses in the middle of various methods?

Classes vs. objects. We are supposed to be specifying classes but we often treat it as if we are dealing with a specific object with a specific history (of method calls, etc.). Such an approach makes sense when dealing with processes in a concurrent language because each process is an actual run time entity and there is only one instance of a given process; but there can be any number of instances of a given class and the approach presented in this paper does not currently deal with this appropriately.

Heap access. We assume that we can access the states of all the relevant objects in the system. However, our formalism does not have any provision to ensure that the heap is properly updated, etc. That is, we are assuming that there is an operational system "running alongside" that keeps track of the heap and gives us the states of all the objects when-

ever we need them. This notion conflicts, however, with our goal of developing an axiomatic reasoning approach. Additionally, we assume access to the state of the aspects, e.g., *ap* and *aq* in the rules depicted in Figures 5 and 6. However, how is the state of the aspect maintained and how can we access it? This is especially problematic if there are multiple instances of the class that an aspect applies to, or if an aspect applies to multiple classes, because each method applied to each instance of a given class `C` will, potentially, trigger the aspects associated with `C` to execute and have that state modified. Somehow, we will have to keep track of this state; essentially, we are treating the aspect state as if it was part of the "static state" of the class `C` without having, in the formalism, any way to deal with such state. And the situation is, of course, worse for aspects that apply to multiple classes since then this state becomes part of the static state of each of these classes.

Acknowledgments

Many thanks to Gary Leavens of the University of Central Florida for valuable discussion and feedback. We would also like to thank the anonymous referees for their extremely useful comments and suggestions.

6. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *Eur. Conf. Object-Oriented Programming*, 2005.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[3] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect-Oriented Languages*, 2002.

[4] C. Clifton and G. Leavens. MiniMAO: Investigating the semantics of proceed. In *Foundations of Aspect-Oriented Languages*, 2005.

[5] C. Clifton, G. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *Eur. Conf. Object-Oriented Programming*, 2007.

[6] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 1963.

[7] D. Dantas and D. Walker. Harmless advice. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[8] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. SyncGen: An AOP framework for synchronization. In *Int. Conf. on Tools and Alg. for Construction and Analysis of Sys.*, 2004.

[9] B. Devereux. Compositional reasoning about aspects using alternating-time logic. In *Foundations of Aspect-Oriented Languages*, 2003.

[10] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Advanced Separation of Concerns*, 2000.

[11] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect-Oriented Languages*, 2006.

[12] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 2006.

[13] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[14] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *International Conference on Computer Aided Verification*, 1998.

[15] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[16] D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 1988.

[17] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Transactions on Software Engineering*, 2001.

[18] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 1983.

[19] R. Khatchadourian and N. Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *Software Engineering Properties of Languages and Aspect Technologies*, 2007.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Eur. Conf. Object-Oriented Programming*, 2001.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Eur. Conf. Object-Oriented Programming*, 1997.

[22] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering*, 2005.

[23] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2004.

[24] R. Laddad. *AspectJ in Action*. Manning, 2003.

[25] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 2003.

[26] M. Lippert and C. Lopes. A study on exception detection and handling using AOP. In *International Conference on Software Engineering*, 2002.

[27] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, 2006.

[28] J. Misra and K. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 1981.

[29] K. Ostermann. Aspects and modular reasoning in nonmonotonic logic. In *Foundations of Aspect-Oriented Languages*, 2007.

[30] K. Ostermann. Reasoning about aspects with common sense. In *Int. Conf. Aspect-Oriented Software Development*, 2008.

[31] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Eur. Conf. Object-Oriented Programming*, 2005.

[32] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 1976.

[33] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Int. Conf. Aspect-Oriented Software Development*, 2003.

[34] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 2003.

[35] N. Soundarajan, R. Khatchadourian, and J. Dovland. Reasoning about the behavior of aspect-oriented programs. In *IASTED Int. Conf. Softw. Eng. and Apps.*, 2007.

[36] F. Steimann. The paradoxical success of aspect-oriented programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[37] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[38] J. Xu, H. Rajan, and K. Sullivan. Aspect reasoning by reduction to implicit invocation. In *Foundations of Aspect-Oriented Languages*, 2004.

[39] Q. Xu, W. de Roever, and J. He. Rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 1997.

Incremental Analysis of Interference Among Aspects

Emilia Katz Shmuel Katz
Computer Science Department
Technion – Israel Institute of Technology
{emika, katz}@cs.technion.ac.il

ABSTRACT

Often, insertion of several aspects into one system is desired and in that case the problem of interference among the different aspects might arise, even if each aspect individually woven is correct relative to its specification. In this type of interference, one aspect can prevent another from having the required effect on a woven system. Such interference is defined and specifications of aspects are described. An incremental proof strategy based on model checking pairs of aspects for a generic model expressing the specifications is defined. When an aspect is added to a library of noninterfering aspects, only its interaction with each of the aspects from the library needs to be checked. Such checks for each pair of aspects are proven sufficient to detect interference or establish interference freedom for any order of application of any collection of aspects in a library. Implemented examples of interfering aspects are analyzed and the results are described, showing the advantage of the incremental strategy over a direct proof in space needed for the model check. Early analysis and detection of such interference in libraries of aspects will enable informed choice of the aspects to be applied, and of the weaving order.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Verification, languages

Keywords

Aspects, interference, model-checking, detection, specification

1. INTRODUCTION

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

Aspects have proven useful for a wide variety of tasks, and aspect languages, such as, e.g., AspectJ [12], have become increasingly popular. Often reuse of aspects is desired, as the same concern might arise in many different systems. In some cases there is more than one aspect we would like to reuse in multiple systems of similar purpose, and then a library of reusable aspects is created. This is the case, for example, in [13], where a library of aspects was created to implement the ACID properties for transactional objects.

However, use of aspects raises many questions of reliability and correctness. It is crucial to establish both that each aspect individually is correct when woven alone, and also to consider possible interference among multiple aspects woven to the same underlying system. This paper considers the second question, giving a precise definition of semantic interference among aspects, showing how to detect it, and how to use examples of interference to modify the aspects or their specifications. The technique is most appropriate for establishing usage guidelines for reusable aspects, especially as libraries of reusable aspects are developed.

We define an incremental proof strategy based on model-checking that establishes whether there exists a legal underlying system in which the aspects interfere. For this strategy, assume-guarantee specifications of aspects described in Section 2 are used to define interference freedom in a way analogous to interference freedom among processes in shared-memory systems [16]. In that classic work, interference freedom among processes is defined in terms of whether independent and local Hoare-logic proofs of correctness for each parallel process are invalidated by operations from other processes. The individual proofs that each aspect is correct when woven alone correspond to the n local proofs of [16], while the approach in this paper deals with the n^2 checks of interference-freedom. A key point, also adapted here, is that the other processes may change the values of shared variables, but there is no interference as long as the independent proofs are not invalidated. The level of interleaving in shared memory systems is much finer than for aspects: every local assertion about memory values can be invalidated by another assignment by a different processor. The fact that aspect advice is only activated at joinpoints means that less stringent conditions can be used, and that modular model checking can be used as a proof component.

The work presented here is the first definition of semantic interference for aspects that uses the specification of the as-

pects as the interference criterion, and applies model checking to detect interference or establish noninterference among collections of aspects. In our method the interference checks are performed on pairs of aspects, and the results of these pairwise checks are shown to be sufficient to determine interference freedom for all the aspects in the library. However, as shown in Section 4, to enable such incremental proofs we have to "pay" by additional incompleteness.

There has been previous work on detecting whether the pointcuts of aspects match common joinpoints or overlapping introductions [4, 7]. This is important because the semantics of weaving can be ambiguous at such points, and be the source of errors. However, as will be shown, aspects can interfere even if there are no common joinpoints.

Some work has also been done in identifying potential influence by using dataflow techniques showing that one aspect changes (or may change) the value of some field or variable that is used and potentially affects the computation done by the advice of another aspect [17]. Slicing techniques for aspects [20, 1, 18] can also be used for such detection. Since such potential influence is often harmless, many false positives can result.

Interferences between an aspect and the base program were discussed in [11]. The reasoning about the influence of the aspects on the base system is based on the assume-guarantee paradigm, which is also used in our specifications of aspects. However, in [11] interactions between aspects are not considered, and no automated verification procedure is presented.

Verification based on model checking has mainly concerned the verification of a single aspect relative to an underlying system. In [14] a modular approach is presented to show that assertions true of a given underlying system remain true when an aspect is added. In [5] a modular verification of an aspect relative to the specifications considered here is shown. Both of these techniques can be used as components in the checking of interference defined here, but the tool presented in [5] is used in this paper. Other work based on model checking determines whether the weaving of aspect scenarios is done correctly [8], or whether a full woven system is correct, using annotations that help construct the (non-modular) proof task for each weaving [10].

2. SPECIFICATIONS OF ASPECTS

A specification of an aspect consists of two parts: its *assumption* about any system into which it may reasonably be woven, and its *result assertion* (also called its *guarantee*) about properties of the result of weaving the aspect into any system satisfying the assumption.

The form of the specification is an instance of the *assume-guarantee* paradigm but generalized to relate to global properties of the system. The assumption of an aspect can include information on what is expected to be true at joinpoints, global invariants of the underlying system, or assumed properties of instances of classes or variables that may be bound to various parameters of the aspect when it is woven. (If the assumption states only properties required from the join-points of the aspect, then it can also be called the *precondition* of the aspect, otherwise the term *precondition* might be misleading.) The result assertion can include both new properties added by the aspect, and those properties of the basic system that are to be maintained in a system augmented with the aspect. Both parts of aspect specification are expressed in linear temporal logic.

Note that, as all the desired properties of the join-points are encapsulated by the assumption, the influence of the pointcuts on the behavior of the aspect is also hidden in it.

In general, there might be a large number of properties of the base system that should be maintained by the aspect. However, due to the analysis presented in [9], many of these properties do not need to be explicitly mentioned in the guarantee of the aspect. In that work ([9]) syntactically identifiable categories of aspects are presented. For each category of aspects broad classes of syntactically identifiable temporal properties preserved by all the aspects of this category are identified. Thus for each aspect and for each temporal property of the base system that should be maintained in the woven system as well, it is possible to efficiently decide whether or not this property should be explicitly included in the aspect's guarantee. Once we have stated, for example, that all safety properties of the base system are also true in the woven system, instances of such properties do not have to be listed in the guarantee. Treating such properties separately makes specification and verification of the rest of the properties much easier.

DEFINITION 1. *An aspect is* correct *with respect to its assume-guarantee specification if, whenever it is woven (by itself) into a system that satisfies the assumption, the result will satisfy the guarantee.*

The question of determining "interference" between an aspect and a base system, i.e., whether or not the aspect maintains all the desired properties of the base system, and whether or not the aspect succeeds to ensure its guarantee, is taken care of in [5] and is out of scope of the current paper. In this paper we assume that all aspects are correct with respect to their specifications, and consider only possible interference among multiple aspects.

3. FORMALIZING SPECIFICATIONS, NON-INTERFERENCE AND PROOFS

We will first define semantic interference between two aspects and present a proof strategy for interference detection. Then it will be shown that performing these pairwise checks is enough to determine non-interference between any collection of aspects.

Given two aspects and their specifications, we can establish whether they interfere semantically, independently of any specific underlying system, while we rely on the correctness of the weaving process of the language in which the aspects are written. There might be some ambiguity in the weaving process, for example regarding the order of weaving aspect advices at a common joinpoint. However, here we assume a standard weaving strategy consistent with the aspect language used for implementation. First let us define system-dependent interference, and then define interference independent of any underlying system. (Recall that

we assume that each aspect is correct with respect to its specification.)

DEFINITION 2. *Given two correct aspects A and B, and an underlying system S that satisfies the assumptions of both A and B, we say that* A does not interfere with B with respect to S *if the following property holds: Let S' be a system obtained from S by first weaving A into S, and then weaving B into the resulting system. Then in S' the guarantees of both A and B hold.*

DEFINITION 3. *Given two correct aspects A and B, we say that* A does not interfere with B *if for every system S satisfying the assumptions of both A and B, A does not interfere with B with respect to S.*

Notice the non-symmetry in the above definition of non-interference: if A does not interfere with B, it does not necessarily mean that B does not interfere with A, and vice versa.

Two aspects are semantically non-interfering if each does not interfere with the other in terms of Definition 3. Note that an aspect can change the values of variables used by the other even if they do not interfere, as long as the correctness of the specification is unchanged.

Let (P_A, R_A) be the specification of aspect A, where P_A is the *P*recondition of A (called the *assumption*), and R_A - its *R*esult assertion (called *the guarantee*). In the same way, let (P_B, R_B) be the specification of B. Note that P relates to a system before the aspect has been woven into it, and R to that system augmented by the aspect, and each aspect is by itself correct. The correctness of the aspects is in terms of Definition 1, and it means that we assume that for every base system that satisfies P_A, the result of weaving A into this system satisfies R_A, and for every base system that satisfies P_B, the result of weaving B into this system satisfies R_B. Then for an underlying system S satisfying both assumptions (P_A and P_B), we need to prove that both sequential weavings - that of A before B and that of B before A - result in a system satisfying both guarantees (R_A and R_B).

DEFINITION 4. *In general, a set $\{A_1, \ldots, A_n\}$ of aspects is* interference-free *if whenever the assumptions $P_1, \ldots, P_n$ hold in a system, the augmented system obtained after weaving the aspects in any order satisfies the guarantees $R_1, \ldots, R_n$.*

Now let us describe non-interference and its proof more formally: The specifications of aspects we consider here are written in Linear Temporal Logic [15]. They relate to computations, which are sequences of states. (In fact, the definitions also hold for branching temporal logic, but the proof method does not.) The notation $S \models \psi$ is used to say that a temporal logic property ψ holds for every computation of a system S. Similarly, to say that a state predicate p holds at a state s of S we write $s \models p$.

To prove that A does not interfere with B, we need to show that

$$OK_{AB} = \forall S((S \models P_A \wedge P_B) \rightarrow ((S + A) + B \models R_A \wedge R_B))$$

holds, where by (S+A) we denote the result of weaving A into S (and by ((S+A)+B) - the result of weaving B into (S+A)). In the same way, to prove that B does not interfere with A we need to show

$$OK_{BA} = \forall S((S \models P_A \wedge P_B) \rightarrow ((S + B) + A \models R_A \wedge R_B))$$

Notice that these are two distinct statements, as in many cases the result of weaving A before B, $(S + A) + B$, will differ from the result of weaving B before A, $(S + B) + A$, as the advice of the previously woven aspect may not apply to the last-woven one. For the same reason both orderings above might differ from the result of simultaneous, AspectJ-like, weaving - the relation between them will be discussed in Section 7. The order of weaving will matter, for example, in the Composition Filters model [2], and in languages with dynamic aspects introduction. Moreover, even in AspectJ, if we first weave A into S and compile the program, and then weave B into the obtained Java bytecode, we do not get the same result as if A and B were woven into S at the same time by the AspectJ weaver.

There exists a way to prove the two above statements - OK_{AB} and OK_{BA} - directly, that will be described later, but it has several disadvantages. The following theorem will enable us to use an *incremental* step-by-step method:

THEOREM 1. *Let A and B be two aspects with the specifications (P_A, R_A) and (P_B, R_B) respectively, and assume that both aspects satisfy their specifications. Then to prove that A does not interfere with B it is enough to show that the following statements hold:*

$$KP_{AB} = \forall S((S \models P_A \wedge P_B) \rightarrow (S + A \models P_B))$$

*("*K*eeping the* P*recondition of B when weaving* A *before* B*")*
and

$$KR_{AB} = \forall S((S \models R_A \wedge P_B) \rightarrow (S + B \models R_A))$$

*("*K*eeping the* R*esult of A when weaving* A *before* B*")*

Proof.
In other words, we need to prove that if A and B are correct aspects, and the KP_{AB} and KR_{AB} statements hold, then A does not interfere with B. The KP_{AB} statement means that the weaving of A into a system S satisfying the assumptions of both aspects does not invalidate the assumption of B. Such an S, in particular, satisfies the assumption of A. We know that after weaving A into a system S that satisfies P_A, R_A is true, for it is assumed as proven that A satisfies its specification. Thus together we have that (S+A) will satisfy not only the assumption of B, but also the guarantee of A, and the following statement will be true:

$$KP'_{AB} = \forall S((S \models P_A \wedge P_B) \rightarrow (S + A \models R_A \wedge P_B))$$

KR_{AB} means that weaving B into a system in which the guarantee of A holds does not invalidate this guarantee. B also satisfies its specification, so in the same way as for A, $S + B$ from KR_{AB} satisfies R_B, and we have

$$KR'_{AB} = \forall S((S \models R_A \wedge P_B) \rightarrow (S + B \models R_A \wedge R_B))$$

Now we can combine KP'_{AB} and KR'_{AB} by substituting S+A instead of S into KR'_{AB}. As a result we will obtain the desired property, OK_{AB}.
Q. E. D.

Symmetrically, to prove that the weaving of A can be performed after the weaving of B we need to show KP_{BA} and KR_{BA}, which, combined, will imply

$$OK_{BA} = \forall S((S \models P_A \wedge P_B) \rightarrow ((S + B) + A \models R_A \wedge R_B))$$

THEOREM 2. *Let $A_1, \ldots, A_n$ be aspects with the specifications $(P_1, R_1), \ldots, (P_n, R_n)$ respectively, and assume all these aspects satisfy their specifications. Assume also that for every pair of indices i, j $KP_{i,j}$ and $KR_{i,j}$ are true. Then the set $\{A_1, \ldots, A_n\}$ is interference-free.*

In order to prove the theorem, the following lemma will be useful:

LEMMA 1. *For every set of $n \geq 2$ aspects $\{A_1, \ldots, A_n\}$ satisfying their specifications $(P_1, R_1), \ldots, (P_n, R_n)$, if for every pair of indices i, j $KP_{i,j}$ is true, then for every base system S such that $S \models P_1 \wedge \ldots \wedge P_n$, the following holds: For every $0 \leq k < n$, $(\ldots(S + A_1) + \ldots + A_k) \models P_{k+1} \wedge \ldots \wedge P_n$ (where the case of $k = 0$ means that no aspects are woven into the system S).*

Proof (Lemma 1).
The proof is by induction on k.
Basis: $k = 0$. We need to show that $S \models P_1 \wedge \ldots \wedge P_n$, but this statement is one of the premises of the lemma.
Induction step: Assume that for every k such that $0 \leq k < m < n$ the statement holds, and let us prove it for $k = m$. Let S be a system such that $(S \models P_1 \wedge \ldots \wedge P_n)$. Let us denote the system $(\ldots(S + A_1) + \ldots + A_{m-1})$ by S'. We need to show that $(S' + A_m) \models P_{m+1} \wedge \ldots \wedge P_n$. From the premises of the lemma, for every $m + 1 \leq i \leq n$ the $KP_{m,i}$ property holds. Also, from the induction hypothesis, $S' \models P_m \wedge \ldots \wedge P_n$, and, in particular, $S' \models P_m \wedge P_i$. Together we have that indeed $(S' + A_m) \models P_i$ for every $m + 1 \leq i \leq n$.
Q. E. D.(Lemma 1)

Now let us prove Theorem 2. Let us be given a permutation $(i_1, \ldots, i_n)$ of indices. Without loss of generality, we can call them $(1, \ldots, n)$. (Clarification: We can always permute the aspects in the library so that for every j, aspect number i_j will stand on the j-th place. Then the order $1, \ldots, n$ on the permuted library will give the same sequence of aspects as the order $i_1, \ldots, i_n$ on the original one.) We need to prove that for every base system S, if $S \models P_1 \wedge \ldots \wedge P_n)$ then $(\ldots(S + A_1) + \ldots + A_n) \models R_1 \wedge \ldots \wedge R_n$. The proof is by induction on n.
Basis: If $n = 1$, there is only one aspect, A_1. Let S be a system satisfying P_1. The aspect A_1 satisfies its specification, thus the statement $(S \models P_1) \rightarrow (S + A_1 \models R_1)$ holds.
Induction step: We assume that the statement holds for any $1 \leq k < m$ aspects from the n given, and prove it for $k = m$. Let us be given a base system S satisfying $P_1 \wedge \ldots \wedge P_n$. We will denote by S' the system $(\ldots(S + A_1) + \ldots + A_{m-1})$. From the induction hypothesis we have that $S' \models R_1 \wedge \ldots \wedge R_{m-1}$. Lemma 1 is applicable here, so we also have that $S' \models P_m \wedge \ldots \wedge P_n$. In particular, $S' \models P_m$. Thus, as A_m is correct according to its specification, $S' + A_m \models R_m$. And for every $i \neq m$, $1 \leq i \leq n$, the $KR_{i,m}$ property holds, thus from the fact that $S' \models P_m \wedge R_i$ it follows that indeed $S' + A_m \models R_i$. Together we get that indeed $(\ldots(S + A_1) + \ldots + A_m) \models R_1 \wedge \ldots \wedge R_m$.
Q. E. D.

A proof that uses Theorem 1, that is, a proof that shows the assumption of the second woven aspect and the resulting assertion of the first one are preserved, is called an *incremental* proof. A *direct* proof merely shows that the weaving of the two aspects achieves both results.

Note that, as opposed to the incremental proofs assumed in Theorem 2, a direct proof of non-interference among pairs of aspects does not generalize to weaving of more than two aspects: even if aspects A, B, and C are pairwise interference-free, and are correct relative to their assumptions and guarantees, weaving of all three into a system with $P_A \wedge P_B \wedge P_C$ does not guarantee $R_A \wedge R_B \wedge R_C$ in the resulting system. For example, $S + A$ might not satisfy P_C, and thus, when B and C are woven, R_C might not result (even though just weaving C would give R_C). Thus the incremental method is essential for showing interference-freedom among groups of aspects of any size.

In some cases a conflict in the specifications of the aspects exists, which means that the specifications do not allow some composition of the aspects. Then for some order of weaving these aspects will always interfere, regardless of their advice implementation, as will be seen in Section 5.2. This composition of the aspects will be called *not feasible* according to the following definition:

DEFINITION 5. *Given two aspects A and B with specifications (P_A, R_A) and (P_B, R_B) respectively, the composition of A before B is* feasible *iff all the following formulas are satisfiable: $P_A \wedge P_B$, $R_A \wedge P_B$, $R_A \wedge R_B$*

If a composition of A before B is not feasible, it means that A has to interfere with B. Thus as a first step in detection of interference, a *feasibility check* can be performed - i.e., a satisfiability check on the appropriate formulas. It is recommended to perform a feasibility check before starting the full verification process described below, because this check is much easier and quicker, and then proceed to the verification only if the composition of the aspects is feasible. However, this is not an obligatory stage of the verification process, because if some contradiction exists, the verification method below will also detect interference and provide a counterexample.

4. PROOF IMPLEMENTATION USING MAVEN

In order to perform a verification without having to consider each possible underlying system, we use and adopt to our purposes the proof method suggested in [5] and the MAVEN tool presented there, while also improving MAVEN and making it more robust. The basic idea of that work, described for the verification of a single aspect relative to its

specification, is that a single model can be generated from the aspect assumption, the pointcut description, and the advice, and used to model check the result assertion. If that model check succeeds, the augmented program resulting from the weaving of the aspect to any underlying system satisfying the aspect assumption is guaranteed to satisfy the result assertion of the aspect.

The single model to be checked is built from the tableau (state machine) that corresponds to the linear temporal logic assertion of the aspect assumption. This tableau is a generic model for all the systems satisfying the assumption of the aspect, and the state machine fragments that correspond to the advice are woven according to the pointcut descriptions. The tableau contains all the possible behaviors of the base systems into which the aspect can be woven. In other words, for any given underlying system that satisfies the assumption of the aspect, for every computation of this system there is a corresponding computation of the tableau, satisfying the same LTL properties. In [5] it is shown that the same holds for the woven systems: if the system S_1 results from weaving the aspect into some appropriate base system, and the system S_2 is the result of weaving the aspect into the tableau, then for every computation of S_1 there exists a corresponding computation of S_2. The properties we check are LTL properties, and an LTL property holds in a system iff each computation of this system, taken alone, satisfies this property. So if there exists a "bad" base system S such that S satisfies the assumption of the aspect, but the resulting assertion of the aspect is violated when it is woven into S, it means that there exists a "bad" computation in the woven system, violating the guarantee of the aspect, and for this bad computation there exists a corresponding bad computation in the (tableau + aspect) state machine. It follows that indeed it is enough to model-check the resulting assertion of the aspect on the (tableau+aspect) system only.

The state-machine fragments corresponding to the advice can be obtained from high-level code, e.g. in AspectJ or Java, using existing tools, such as Bandera [6]. Alternatively, the state machines can be created at earlier stages of the programming cycle to serve as abstract models of aspects during their design, before code is generated. Note that the aspect can consist of multiple pointcuts and advices, but not only the correctness of theorems 1 and 2 from Section 3 is not invalidated in this case, but this case is also supported by the implementation of the proof method presented below. However, if such complicated aspects do interfere, the diagnosis of the interference cause would be more efficient if the aspects were split into simpler ones, each with one advice only - if such a splitting is possible.

In order to use the MAVEN tool, we have to pose one restriction on the aspects: they should both be of the *weakly invasive* category, as defined in [9]. An aspect is weakly invasive if whenever its advice completes its execution, the resulting state, when the local variables and private objects of the aspect are ignored, already existed in the original underlying system. Notice that the advice can change state variables of the underlying system during its execution, and that local aspect variables can be modified with no restriction. The restriction to weakly invasive aspects is not a strong one, as most aspects fall into this category, including all of our examples. When an aspect is not weakly invasive, it may return control to the basic system in a state that was not previously reachable, and thus the effect of the base system's code is not limited by the assumption. The restriction on the aspects is not necessary theoretically: our statements are sound for all types of aspects. As MAVEN improves, or other verification tools for aspects become available, this restriction may become unnecessary.

Note also that a verification using MAVEN is relative to the standard weaving strategy built into that tool. Again, this is not inherent to our approach.

In order to use MAVEN as a subsystem for our technique, it was extended in several ways. Most significantly, - now it is possible to automatically determine whether the aspect verified is weakly invasive. In the first version of MAVEN the given aspect could be woven only into a tableau built from its assumption, while now it is possible to weave an aspect into an arbitrary transition system (in the NuSMV format). It also now allows initializing aspect variables globally, and preserving aspect values between activations of advice.

In our context, to show that A does not interfere with B, we will use the *incremental* method. The verification can be preceded by a feasibility check of the composition of aspects. The verification process, based on Theorem 1 and using the improved MAVEN tool, will be as follows:

1. To prove KP_{AB}, build a tableau that corresponds to the conjunction of the assumptions of the aspects, $P_A \wedge P_B$, weave the advice of A and show that the assumption of B, P_B, is true of the result.

2. In order to prove KR_{AB}, a tableau that corresponds to the conjunction of the assumption of B and the guarantee of A, $R_A \wedge P_B$, is built. Then after the advice of B is woven in, show that the guarantee of A, R_A, still holds for the result.

The proof that B does not interfere with A is symmetric. When we prove non-interference in both directions, although there are four steps of verification, we need to build only three different tableaus: one for the proofs of KP_{AB} and KP_{BA}, and the other two - for the proofs of KR_{AB} and KR_{BA}, respectively.

The above method is sound, due to Theorems 1 and 2, but not complete. There are two cases when the OK_{AB} check fails though the aspects do not interfere (the case of OK_{BA} is, of course, symmetric): The first case is a failure due to the incompleteness of the model-checking itself. If the model we are model-checking is infinite, or finite but too large, the model-checking will collapse without providing any answer. So, as always when model-checking is used, the models and the verified properties should be described at a sufficient level of abstraction. The second case is when the specification of some aspect is not the most general possible. Then there are two possibilities for failure - one arises when the assumption of aspect B, P_B, is not the weakest possible, and the other - when the guarantee of A, R_A, is not the strongest possible. In the first case, as P_B is not

the weakest possible, it might happen that aspect A does not preserve the assumption of aspect B, but assures some other property, P'_B, that is enough for aspect B to operate correctly. Then the KP_{AB} check fails, but the OK_{AB} is true. In this case, if P_B was the weakest possible (i.e., such that $(S \models \neg P_B) \rightarrow (S + B \models \neg R_B)$), this possibility of interference would be eliminated. In the second case, symmetrically, it might happen that we can not prove that aspect B preserves the guarantee of A, because the assumption $R_A \wedge P_B$ is not strong enough to ensure R_A after B is woven, but the OK_{AB} property is true because A guarantees a stronger statement, R'_A, and with this assumption B is able to preserve R_A (for every system S, if $S \models R'_A \wedge P_B$, then $S+B \models R_A$). Notice some non-symmetry in the above statement - we have to assume R'_A, but can guarantee only R_A, because that is the property proven by the successful OK_{AB} check. In fact, by demanding that aspect B will preserve R_A when woven into *any* system that satisfies $R_A \wedge P_B$, we pose too strong a restriction, because we are interested in this statement only for base systems in which aspect A is present. This is an additional source of incompleteness in this case.

One could try to eliminate the second cause of incompleteness by verifying the OK_{AB} statement directly: Build a tableau that corresponds to the conjunction of the assumptions of the aspects, $P_A \wedge P_B$, weave the advice of A into the above tableau, and then weave the advice of B into the resulting state machine. Then verify both guarantees, $R_A \wedge R_B$, on the result. However, as we saw before, the direct method can not be generalized to more than two aspects. Moreover, even if we are interested only in detecting interference between two aspects, the analysis below shows that the space complexity of the direct method is higher than that of the incremental one, and thus the model-checker might fail to perform the direct verification, while succeeding to perform the incremental one.

For the complexity analysis, we assume, for convenience of presentation, that all the specifications are of (approximately) the same size, and all the advice machines are of (approximately) the same size as well. Given two aspects, A and B, with the specifications (P_A, R_A) and (P_B, R_B) respectively, we denote by r the maximal length of a formula from the aspect specifications $(max(|P_A|, |R_A|, |P_B|, |R_B|))$, and by M - the maximal size of the advice model of the aspects $(max(|M_A|, |M_B|))$.

LEMMA 2. *The space complexity of the incremental method is* $O(2^{3r} \cdot M)$.

Proof.
The size of the tableau built from an LTL formula of length k is $O(2^k)$ (as shown in [3]). In our case the tableau is always built from two properties, so the size of the tableau is $O(2^{2r})$, and the size of the woven system on which the resulting assertion is model-checked is $O(2^{2r} \cdot M)$. When a formula of size k is verified on a model of size m, the space complexity of the model checking is $O(m \cdot 2^k)$ ([3]). In our case, $m = O(2^{2r} \cdot M)$ and $k = r$, so the altogether space complexity is $O(2^{2r} \cdot M \cdot 2^r) = O(2^{3r} \cdot M)$.
Q. E. D.

LEMMA 3. *The space complexity of the direct method is* $O(2^{4r} \cdot M^2)$.

Proof.
Here first the assumptions tableau is built from the two assumptions of the aspects, and its size is $O(2^{2r})$. Then two advice models are woven into it, one after another, so the size of the woven system is $O(2^{2r} \cdot M^2)$ (M^2 appears here because there might be join-points of the second aspect inside the advice machine of the first). The property verified on this woven system is the conjunction of the two resulting assertions of the aspects, so the property size is $O(2^{2r})$. Together we have that the space complexity is $O(2^{2r} \cdot M^2 \cdot 2^{2r}) = O(2^{4r} \cdot M^2)$.
Q. E. D.

5. EXAMPLES

5.1 Encrypting Passwords

In this example we discuss two reusable aspects, E and F, that may appear in a security-aspects library and might be used in a password-protected system. An example of such a system can be the internet terminals of a bank, providing the possibility of viewing and/or updating the user's account via the internet.

Aspect E is responsible for encrypting the passwords before sending.The joinpoint E advises is the moment when the password-containing message is to be sent from the login screen. E's advice is a "before" advice, that encrypts the message. E should guarantee that each time a password is sent, it is encrypted, and the assumption of E might be that password-containing messages are sent only from the login screen in the base system. In fact, there is more to E: each time a password is received, it is be decrypted. But this part is irrelevant to our example, so we'll ignore it here. A possible specification for E can thus be that E assumes that password-containing messages are sent only from the login screen in the base system, and guarantees that each time a password is sent, it is encrypted. More formally it can be written as follows:

$$P_E = G(psw\_send \leftrightarrow login\_psw\_send)$$

and

$$R_E = G(psw\_send \rightarrow encrypted\_psw)$$

where the predicate *psw_send* means that a message containing a password is being sent, and *login_psw_send* means that the password is being sent from the login screen. The assumption of the aspect might seem arbitrary, but this choice was guided by a possible implementation of the aspect. It might be the case that the aspect is unable to identify password-containing messages from the message content only, and then the pointcut could be defined as the creation of a message containing information from some specific field of the graphic user interface. Note that the aspect is generic (to enable reuse), and thus the field from which the information is taken should be a parameter bound to the aspect when adding the aspect to a concrete system.

Aspect F is responsible for treating a situation when the user forgets the password. Usually in password-guarded systems there is a way of retrieving your password once you forget

it. F provides a list of security questions to the user, and if the questions are answered correctly, F guarantees that the user will get his password via an e-mail. In order to add this functionality to the system, F should add some introductory operation. For example, a new button - "Forgot my password" - can be added to the system so that we can define the pointcut of F as the moment when this button is pressed. F's advice then provides the dialog with questions, checks the answers and in case all the answers are correct - sends an e-mail to the user. The button is added by F itself, thus one of the possible ways to specify F is to say that F assumes nothing about the system (because F itself adds to the system the possibility to report forgetting the password), and guarantees that whenever the security check is passed, the forgotten password will be sent to the user (and if the check is never passed, the password remains forgotten forever as it was in the base system). More formally, F's assumption is

$$P_F = true$$

And F's guarantee is

$$R_F = [G((button\_pressed \wedge quest\_answered) \rightarrow F(psw\_send)))]$$

where *button_pressed* is the flag that means forgetting the password has been reported and not yet treated.

Let us check the possibility of sequential weavings. F's assumption is *true*, thus E can not violate it. Thus in order to check the possibility of weaving F after E, we need to prove only that the weaving of F maintains the guarantee of E (the KR_{EF} statement):
$\forall S((S \models G(psw\_send \rightarrow encrypted\_psw)) \rightarrow$
$(S + F \models G(psw\_send \rightarrow encrypted\_psw\ )))$
This statement seems to be reasonable, and the feasibility check succeeds, but the advice of aspect F is implemented in such a way that the password sent from it is not encrypted. Thus when trying to verify the KR_{EF} statement, a counterexample is obtained. It is a computation in which at some state s_1 the predicate *button_pressed* became true, and at the same time the predicate *encrypted_psw* was false. Two states after that, at a state s_2, due to the operation of the aspect F, *quest_answered* became true (while *button_pressed* was still true), and in the next state, s_3, *psw_send* became true. But F does not encrypt the passwords, thus *encrypted_psw* was still false at s_3, contradicting the implication in R_E, so the verification failed.

In order to check the possibility of weaving E after F, we need to prove that the weaving of F to a system satisfying both assumptions maintains the assumption of E (the KP_{FE} statement):
$\forall S((S \models G(psw\_send \leftrightarrow login\_psw\_send) \wedge (true)) \rightarrow$
$(S + F \models G(psw\_send \leftrightarrow login\_psw\_send)))$.
However, the implementation of the advice of F leads to violation of the assumption of E, because F does not send the password from the login screen. Note that in this case, again, there is no contradiction in the specifications of E and F, so the feasibility check succeeds, and the interference is detected during the verification only.

Two remarks about the example: (1) If E and F were the only two aspects existing in the library, it would be very easy to detect the interference just by looking at the library, but, as already noted, in real life many different aspects are added to systems and libraries by different groups of people, and an automated solution is needed. (2) In this example we see that the conflicting aspects do not share any joinpoints, and the interference doesn't emerge from updating common variables.

A variant of this example, both as Java code and abstract models, is presented at `http://www.cs.technion.ac.il/ssdl/pub/SemanticInterference/`
It includes the input for verification by the indirect method, checking whether E interferes with F, including aspect descriptions in the NuSMV format, and appropriate LTL assertions (the KP_{EF} stage is not interesting in our case, so only the KR_{EF} stage is presented), and the output of the verification - the counterexample provided by NuSMV. Some statistics for this variant, comparing the verification by indirect and by the direct method, appear in Figure 1. $|M.|$ means the model size and is measured by the number of BDD nodes in the model, and $|Ex.|$ means the number of states in the counterexample found (0 means the result of verification was *true*). These statistics show the additional disadvantage of the direct method: not only is this method applicable only for the case of two aspects, but also the models it creates for verification of two aspects interference are much bigger than those created by the incremental method. In average for this example the models created by the direct method are more than 4 times bigger than those created by the incremental, and the maximal ratio of model sizes for this example is almost 7.

Remark: as a result of verification of KP_{FE}, a counterexample was obtained. Thus it would be possible to stop the verification at this stage and try to amend the aspects and/or their specifications before continuing to verification of KR_{FE}.

Direct method			**Incremental method**		
Check	$\|M\|$	$\|Ex\|$	**Check**	$\|M\|$	$\|Ex\|$
OK_{EF}	7778	18	KP_{EF}	1127	0
			KR_{EF}	1283	12
OK_{FE}	8700	18	KP_{FE}	2375	12
			KR_{FE}	2450	0

Figure 1: Security example statistics

5.2 ATM Communication and Card Theft

In this example we consider two aspects that can be used in a system with remote authorized access. They are most useful for systems in which each user can have only one open session at a time. The first aspect (aspect C below) treats communication failures in the system, that occurred during authorization process or while some authorized user was logged in. Its goal is to assure that the user will be able to log in again after the communication is restored. The second aspect (aspect T) prevents identity-theft: for example, if a wrong password is provided in several consequent attempts of logging in, the aspect guarantees that the user is blocked. One possible system in which these aspects might be used is an ATM system of a bank, consisting of several ATM machines and a server. The user interacts with this system by first inserting a card and code, and then, if permission is

granted, entering a request for some bank operation (money withdrawal, or account balance check).The ATM machine communicates with the server to process user requests, and the server grants or denies permission to perform operations, and processes the operations permitted. From the point of view of the aspects, the card serves as a user-login, and code - as a password. To make the example more intuitive, all the descriptions below are written in terms of the ATM system (note that the aspects are still general and reusable, even after this concretization, because many different implementations of the above-described ATM system are possible). A more detailed description of the aspects is as follows:

Aspect C (for *C*ommunication) is responsible for treating communication failure between the server and an ATM machine. In case of communication failure, the aspect checks whether there is a card stuck in the ATM machine, and returns it to the user. One of the reasonable specifications of C is: C assumes that the only case when a card can get stuck in a machine is when a communication failure occurred while the card was in the machine. In such a case C guarantees that a card is never stuck in a machine forever. Formally,

$$P_C = G(card\_in \rightarrow F(\neg card\_in \vee comm\_fail))$$

(which means that if a card was inserted, either it will be eventually returned, or a communication failure - indicated by $comm\_fail$ predicate - will happen), and

$$R_C = G(card\_in \rightarrow F(\neg card\_in))$$

(which means that if a card was inserted, it will eventually be returned). Note that the $comm\_fail$ predicate does not necessarily represent a general communication failure in the whole system. Our abstraction here is that the flags $comm\_fail$ and $card\_in$ relate to a communication failure and card status at a particular ATM.

Another aspect, T (for *T*heft), comes to prevent card-theft. A possible specification is: T assumes that there exists a possibility to detect that the card is stolen, and if the card is stolen it will remain stolen forever. T ensures that such a card will never return to the user. Formally,

$$P_T = G(card\_stolen \rightarrow G(card\_stolen))$$

and

$$R_T = G((card\_in \wedge card\_stolen) \rightarrow G(card\_in))$$

Let us examine the possibility of sequential weaving of T after C. One of the statements we need to show is that T does not violate the guarantee of C,KR_{CT}:
$\forall S((S \models (G(card\_in \rightarrow F(\neg card\_in)) \wedge (card\_stolen \rightarrow G(card\_stolen)))) \rightarrow (S+T \models G(card\_in \rightarrow F(\neg card\_in))))$.
There is a contradiction in the requirement from $S+T$: On one hand, we require R_C, that says that an inserted card will eventually be returned in every case. On the other hand, T satisfies its specification, and P_T was true in S, thus R_T should be true in $S+T$. By that we require that in some special case (that of a stolen card) the card will never be returned, which contradicts our first requirement (which is the R_C assertion). This contradiction is found when the R_C assertion is model-checked on the Tab+T system, built by weaving T into the tableau Tab of $R_C \wedge P_T$. Thus it is impossible to weave T after C, at least not with such a specification. Note that in this case the feasibility check is also able to detect interference, because there is a contradiction between R_C and R_T. Similarly, we will find a contradiction and counterexample to the weaving of C after T.

Verification statistics for this variant, comparing the verification by indirect and by the direct method, appear in Figure 2. Note that here, as in the previous example, the model sizes of the incremental method are much smaller than those of the direct (about $\frac{1}{3}$ of their size).

Direct method			**Incremental method**		
Check	$\|M\|$	$\|Ex\|$	**Check**	$\|M\|$	$\|Ex\|$
OK_{CT}	3154	8	KP_{CT}	1038	0
			KR_{CT}	776	8
OK_{TC}	3045	5	KP_{TC}	1028	8
			KR_{TC}	1098	11

Figure 2: ATM example statistics

6. ERROR ANALYSIS

When interference has been detected between two aspects, the cause of the verification failure should be localized - which property was violated, and which advice is "guilty". The verification process is divided into stages, making the localization straightforward: if we fail to prove the OK_{AB} and there is a problem in violating the assumption of B, the proof of KP_{AB} will fail, and if the advice of B violates the guarantee of A, the failure will occur in the proof of KR_{AB}.

After the cause of the failure is localized, one needs to decide on what steps should be taken next. In many cases there is a need to add the functionality of both aspects to the base system, in spite of the interference detected between them. There are several possible ways to handle this problem, depending on the type of the interference detected, and the results of the feasibility check (thus it is recommended to perform the feasibility check of the specifications as a first step of error analysis in case an interference is detected). One should then decide whether to change the advice of one of the aspects (or both), and whether the specification of the aspects should be refined. For the examples from Section 5, the way to interference elimination might be as follows:

For the example in Section 5.2, the weaving of C before T appears to be non-feasible, thus the first step to elimination of the interference is to try and refine the specification of the aspects in such a way that the composition becomes feasible. And indeed, there is a possibility of such a refinement: if we knew of the possibility of stealing the card, or, more generally, of special events other than communication failure that can cause the card to be stuck, we could update the guarantee of C to treat these events: $R_C = G((card\_in) \rightarrow F(special\_event \vee \neg card\_in))$, and then add $(card\_stolen \rightarrow special\_event)$ to P_T. Then C would not interfere with T. Note that if such a refinement is possible, it means that the specification provided by the user for one of the aspects (or, maybe, for both) was too strong.

For the example in Section 5.1, on the other hand, aspect F interferes with E, though the composition of E after F is

feasible. In this example it is impossible to eliminate the interference by changing only the specification of the aspects, and a change in one advice, or in both, is necessary. For instance, we can change the advice of F to bring the user to a version of the login screen where the password can be changed, instead of sending the e-mail with the password. In this case, if E is woven after F, the password-sending operation of F is done by the user as another login-password send and thus will be a legal joinpoint of E. Therefore the advice of E will be performed and no password will be sent unencrypted. More formally: the specification of F can stay the same, but as a result of the change in the advice, whenever *psw_send* is true, so is *login_psw_send*. Aspect E and its specification will stay as before. Now the verification will be of F's new code relative to the specifications, so that KP_{FE} and KR_{FE} now will hold. This means that the sequential weaving of first F and then E is possible. Notice, however, that weaving first E and then F would still be problematic.

7. JOINT WEAVING

The above discussion treated only sequential weaving. Let us now consider the case of simultaneous weaving. Such a weaving at every point of the program decides whether to apply A, or B, or both, and in which order (as opposed to sequential weaving, where the possibility of inserting only one aspect at a time is checked). One approach is to reduce joint weaving to sequential weaving, whenever possible. Then given aspects A and B, we would like to check whether weaving both A and B together into some base system is equivalent to one of the sequential weavings (A after B or B after A) into the same base system. If A and B have a common join-point, then the ordering of application may not be well defined, and this is well-known to create possible ambiguity. The lemmas below assume no common join-points, because some of the alternative semantic meanings violate the lemmas.

The following definitions will be useful to us:

DEFINITION 6. *Let S be a system, and A and B - two aspects. Let us denote by J the set of all the joinpoints that are matched by B in S, and by J′ - the set of all the joinpoints that are matched by B in (S+A). We say that A* creates a joinpoint *matched by B if there exists a joinpoint* $j_1 \in J'$ *such that* j_1 *is not in J (that is, J′ is not included in J). We also say that A* removes a joinpoint *of B if there exists a joinpoint* $j_2 \in J$ *such that* j_2 *is not in J′ (that is, J is not included in J′)).*

Thus if A does not create or remove joinpoints matched by B, it means that the joinpoints matched by B in the original system S are exactly the same as in (S+A) - the system obtained by weaving A into S.

The following lemma shows that if weaving aspect B into a base system does not affect join-points of A (i.e, the joinpoints of A in the woven system are the same as in the base one), and the symmetric statement holds - weaving aspect A into a base system does not affect join-points of B - then the order of weaving of the aspects "does not matter" for the final result:

LEMMA 4. *Let S be a system such that there is no joinpoint in S matched by both A and B, and they do not create or remove joinpoints matched by each other. Then the simultaneous weaving of A and B into S (S+(A,B)) is equivalent to both sequential weavings: of A before B ((S+A)+B) and of B before A ((S+B)+A). That is, the weaving is both associative and commutative.*

It is also not difficult to treat the possibility of adding joinpoints of the second woven aspect in the advice code of the first, as seen in the following lemma.

LEMMA 5. *Let S be a system such that there is no joinpoint in S matched by both A and B, and B does not create or remove joinpoints matched by A. Let it be possible for A to create joinpoints matched by B, but only inside its (A's) own advice and without removing joinpoints matched by B. Then the simultaneous weaving of A and B into S (S+(A,B)) is equivalent to weaving A before B ((S+A)+B). That is, the weaving is associative, but not necessarily commutative.*

The proofs of the lemmas appear at the same site as the example. In order to check that the above lemmas can be applied, we need to establish that A and B do not match common joinpoints. For that purpose existing tools mentioned in Section 1 can be used ([4, 7]).

8. CONCLUSIONS

In this paper we have defined semantic interference among aspects relative to their specifications and shown an effective way to detect interference or prove interference-freedom of multiple aspects in a library.

The interference-detection method is modular: the library of aspects is checked independently of any base system. Thus when the user would like to weave multiple aspects from the library into some base system, the only check that should be performed is that the base system satisfies the assumptions of all the aspects that will be added to it.

The result of the verification process is not a "yes" or "no" answer, stating whether or not the current library is interference-free: the results of the verification are more informative. For each aspect we know with which aspects it does not interfere, and also for every aspect with which an interference exists, we know what is the cause of the interference, and in which order of weaving it occurs. All this information can serve as usage guidelines for the developers who would like to use aspects from the verified library. In case the library as a whole is not interference-free, a developer might chose some interference-free subset of the library (recall that pairwise interference-freedom of the aspects in any set is enough to guarantee interference-freedom of the set as a whole), or decide on an appropriate weaving order of the aspects to prevent interference.

There already exist libraries of reusable aspects. One of them - a library implementing ACID properties for transactional objects - is described in [13], and different kinds of interference among the aspects from this library are mentioned there. As future work we would like to apply our verification method to detect interference among aspects from

that library, and analyze the different types of interference described in that paper.

Currently we have started to work on an aspect case study demonstrator of the AOSD-EUROPE project, based on the Toll System [19] written by the Siemens company: a system designed for computing fees and charging the drivers for the use of toll roads. The goals of the case study are formalization and verification of aspects from the Toll System and include detection of possible interference among them. As one example, we have discovered and are analyzing interference between an aspect to impose fines and an aspect to give discounts on the use of the toll road.

This paper deals with interferences between the aspects, but the formalization and proof methods it provides can be easily extended to treat some other types of aspect interactions that can be formalized and checked by similar means. For example, aspects may work cooperatively when one aspect is dependent on another to establish its assumption and cannot be woven into a system unless that other aspect is also there.

9. REFERENCES

[1] D. Balzarotti, A. D'Ursi, L. Cavallaro, and M. Monga. Slicing AspectJ woven code. In *Proc. of Foundations of Aspect Languages Workshop (FOAL05)*, 2005.

[2] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *CACM*, 44:51–57, 2001.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[4] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse, and interaction analysis of stateful aspects. In *Proc. of 3th Intl. Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, 2004.

[5] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Proc. of TACAS 2007*, volume 4424 of *LNCS*, pages 308–322, 2007.

[6] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to model-check properties of concurrent Java software. In K. G. Larsen and M. Nielsen, editors, *Proc. 12th Int. Conf. on Concurrency Theory, CONCUR'01*, volume 2154 of *LNCS*, pages 39–58. Springer-Verlag, 2001.

[7] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07*, pages 85–95. ACM Press, 2007.

[8] E. Katz and S. Katz. Verifying scenario-based aspect specifications. In *Proc. Formal Methods: International Symposium of Formal Methods Europe (FM05)*, volume 3582 of *LNCS*, pages 432–447. Springer, 2005.

[9] S. Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect Oriented Software Development (TAOSD)*, 1:106–134, 2006. LNCS 3880.

[10] S. Katz and M. Sihman. Aspect validation using model checking. In *Proc. of International Symposium on Verification*, LNCS 2772, pages 389–411, 2003.

[11] R. Khatchadourian and N. Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *Proc. of Software Engineering Properties of Languages and Aspect Technologies SPLAT'07*, 2007.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. ECOOP 2001*, LNCS 2072, pages 327–353, Jun 2001. `http://aspectj.org`.

[13] J. Kienzle and S. Gélineau. AO challenge - implementing the ACID properties for transactional objects. In *Proc. 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 202–213. ACM, 2006.

[14] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Proc. SIGSOFT Conference on Foundations of Software Engineering, FSE'04*, pages 137–146. ACM, 2004.

[15] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[16] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[17] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. of International Conference on Foundations of Software Engineering (FSE04)*, 2004.

[18] M. Storzer and J. Krinke. Interference analysis for AspectJ. In *Proc. of Foundations of Aspect Languages Workshop (FOAL03)*, 2003.

[19] Toll system demonstrator. `http://www.aosd-europe.net` (under the "Industry" section).

[20] J. Zhao. Slicing aspect-oriented software. In *IEEE International Workshop on Programming Comprehension*, pages 251–260, 2002.

A synchronized block join point for AspectJ

Chenchen Xi Bruno Harbulot John R. Gurd

The University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom

xic AT cs.man.ac.uk bruno.harbulot AT manchester.ac.uk jgurd AT cs.man.ac.uk

Abstract

Designing and implement model for a synchronized block join point to encapsulate crosscutting synchronization concerns into single unit in AspectJ.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Language.

Keywords Synchronized Block Join Point, AspectJ.

1. Introduction

Synchronization is a concern that developers of distributed systems must deal with whenever guarded access to a shared resource is required. Access to devices, files and shared memory are all situations that typically require synchronization, and they also often require careful management of multiple threads and synchronization devices such as locks. Avoiding tangling of the code responsible for these various concerns is difficult, as is encapsulating them for reuse in diverse situations.

In the context of distributed Java programs, three distinct synchronization situations are encountered, namely, sharing data among multiple threads, sharing among clusters of JVMs and sharing among clusters of physical computers. The former is the logical concern of sharing data between concurrent threads; the latter two are practical concerns about the particular 'distributed environment' in which the code is executed. The problem is how to distribute the logically necessary Java threads transparently across the physical vagiaries of the distributed environment.

Moreover, parallel programming is often difficult simply due to the complexity of dealing with lock-based synchronization. As a result, there have been proposals to simplify parallel programming by using various forms of *transactional memory* to replace lock-based synchronization in existing parallel Java programs. Once *Java Memory Model* (*JMM*) issues have been addressed, conversion of lock-based synchronization into transactions is largely straightforward. However, it is still problematic to avoid code tangling while effecting this conversation.

Aspect-Oriented Programming (*AOP*) has the potential to modularise such synchronizations so that user code can become oblivious to the distributed environment. Indeed, a join point based on the `synchronized` method has been proposed. However, the `synchronized block` has not yet been treated in *AspectJ*.

This paper [1] shows how to separate the synchronization concern by designing and implementing models for a synchronized block join point and a synchronized block body join point to encapsulate crosscutting synchronization concerns into logical units using the concepts of *AOP*. It is also shown how the two new join points can help such modularization to plug into the *JMM* so as to maintain its semantics, along with the semantics defined in the *Java Language Specification*. The models achieve reusability of synchronized code and thread control management in Java to such an extent that concurrency can be fully handled by a single aspect. The models augment the capabilities of join points for synchronized methods in intercepting and modifying synchronization actions in distributed, Java-based, aspect oriented software. The work is applicable in any aspect oriented environment, but emphasis is placed on compatibility with the most commonly used language *AspectJ*.

The proposed join point model is enhanced with a mechanism for removal of unnecessary synchronization, which is vital for reducing overheads associated with the lock. There is also a facility for re-introducing necessary synchronization that has previously been removed.

```
void around(HashMap map): synchronize()
        withincode(method()) && args(map){
        atomic{ rm_proceed(map) ; }  }
```

Consider the above simple example, which is captured by the `synchronize()` pointcut. With transactional execution, there is no need to use anything other than a non-locking HashMap since the caller specifies its atomicity requirements. The `rm_proceed()` method provides the ability to remove the synchronization on HashMap `map` and `atomic` warp it as a transactional object. Extensions for the `abc` compiler which implement the two new join points are briefly presented and are shown to meet the requirements of various applications.

[1] Full text can be found at http://www.cs.man.ac.uk/~xic/SBJP_AspectJ.pdf

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

Onspect: Ontology Based Aspects

Parisa Rashidi
Washington State University
Pullman, WA
US, 99164
001-509-335-1786
prashidi@wsu.edu

Roger T. Alexander
Washington State University
Pullman, WA
US, 99164
001- 509-335-0922
rta@eecs.wsu.edu

ABSTRACT
In software engineering community, semantic interoperability usually has been ignored despite its significant importance. To achieve semantic level interoperability, ontology as a powerful means of expressing and sharing knowledge can be used to add meaningful standard semantics to syntactic annotations. In this paper we describe semantic pointcuts based on ontology modeling. Current AOP models, like many other programming models, primarily rely on a syntactic representation and mostly ignore pointcut expression at semantic level. We present a pointcut modeling approach based on semantics instead of underlying program's syntax, by using ontology modeling to conceptually modularize crosscutting concerns.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Classes and object;* F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *Denotational semantics.*

General Terms
Algorithms, Design, Languages.

Keywords: Semantic Pointcuts, Ontology, Onspect.

1. Onspect
In AOP community, like most other software engineering communities, less attention has been paid to semantic interoperability and semantic pointcuts, and most current mainstream AOP techniques separate crosscutting concerns based on mere syntax. Lack of semantics in AOP has led to problems such as fragile pointcuts, due to tight dependence of aspects on the syntax [1][4]. Various solutions have been proposed by different researchers to define semantic pointcuts, such as [5], [4][4], [2]. Though most of the above works address the fragile pointcut problem through use of semantic elements, neither focuses on a standard semantic modeling tool such as ontology to provide semantic interoperability and unambiguous sharing of pointcut semantics. In our work, we present a new approach toward semantic aspect modeling based on ontology, called Onspect. Ontology as an explicit specification of a conceptualization can be used as a powerful tool for adding semantics to syntactic forms, and for sharing knowledge unambiguously among different implementations and organizations [1]. Using ontology for modeling aspects allows for conceptual modularization of crosscutting concerns among heterogeneous nodes, thus reducing problem of fragile aspects.

To represent programming domain ontology, we introduce a formal model based on concepts, attributes, relationship and constraints to model basic ontology elements of a program. The program itself is modeled as an agent; its set of methods and functionalities are modeled as behaviors; and objects and their roles in the program are modeled as subjects and roles. We then map our formal template into a simple and easy-to-use set of Java annotations to annotate those semantic units. Java annotations are automatically converted into OWL constructs [6], which is a standard ontology modeling language providing a set of necessary reasoning and querying operators. The semantic pointcuts are then defined using a set of semantic quantifiers that refer to the Java annotation elements as semantic concerns. To achieve semantic interoperability between heterogeneous remote nodes, we use JAsCo's hook and connector style for declaring Onspects [3].

In summary, current approach provides a model for defining semantic pointcuts in a heterogeneous environment based on ontology to reduce dependability of crosscutting concerns on mere syntax. However as a first approach for modeling semantic pointcuts based on ontology, current model has its own limitation and shortcoming. In our future works, we plan to extend the ontology model to provide further semantic information and also to develop standard template contents for the ontology model.

FOAL 2008, April 1, 2008, Brussels, Belgium. Copyright 2008 ACM
ISBN 978-1-60558-110-1/08/0004 ... $5.00

2. REFERENCES
[1] T.R. Gruber. Towards Principles for the Design of Ontologies used for Knowledge Sharing. Journal of Human Computer Studies, 1993, pages 907--928.

[2] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In Proceedings of AOSD 2003, 2003.

[3] JAsCo: An aspect-oriented approach tailored for component based software development. In AOSD Proceedings, ACM Press, 2003, pages 21–-29.

[4] Andy Kellens, Kim Mens, Johan Brichau, Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. ECOOP 2006: 501-525.

[5] Klaus Ostermann, Mira Mezini, Christoph Bockisch. Expressive Pointcuts for Increased Modularity. ECOOP 2005:214-240.

[6] OWL Standard, http://www.w3.org/TR/owl-ref/.

De-constructing and Re-constructing Aspect-Orientation

William Harrison
Department of Computer Science
Trinity College
Dublin 2, Ireland*
(+353) 1-896 8556

Bill.Harrison@cs.tcd.ie

ABSTRACT

Through its decade-and-a-half long evolution, the aspect-oriented software community has occasionally struggled with its identity – revisiting the question "What kinds of technologies make up aspect-oriented software and who should be interested in it?" We attempt to de-construct "aspect-oriented" into several issues making up its foundation, believing that the community is inclusive and that work exploring or exploiting any of these concepts fits within the community. Although their historical setting contributes somewhat to the understanding of why different authors have emphasized one or more of these issues, we analyze them from an intrinsic point-of-view, to highlight broader or deeper issues that may lie behind the constructs currently made available, in the hope that "aspect-oriented" software technologies can be extended to provide an even stronger basis for software than they do today.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: *Data abstraction, Information hiding, Languages*

D.3.3 [**Language Constructs and Features**]: *Modules, Packages, Concurrent programming structures*

General Terms

Languages, Design

Keywords

Aspect-oriented, separation-of-concerns, encapsulation, software-composition, event-flow, broadcast, obliviousness, complex-event-processing, specification, modularity, malleability.

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

1. INTRODUCTION

Much discussion has taken place about Aspect-oriented Programming (AOP) and Aspect-oriented Software Development (AOSD), both in characterization of the field as a whole and in attempts to use that characterization to evaluate its value. The adjective "aspect-oriented" came into its current use[1] to qualify "programming". As such it characterized a research view in which different "aspects" of a program, like distribution or storage layout, were addressed by different languages and language processors, with results "woven" together by an aspect weaver [18]. When a single-language focus was adopted that resulted in AspectJ, the term "aspect" was also refocused, and defined, for patent purposes, as an "aspect ... comprising: a cross-cut ... and a cross-cut action" [17]. The emphasis here was on the fact that unlike earlier work, both crosscut specification and consequent action must occur together in the aspect. Several other groups of researchers and developers were pursuing related approaches to the expression of software in which different concerns are expressed in separate artefacts linked by exchange of messages [10] [1]. These groups of researchers coalesced to form the growing community working on AOSD. In that expanded context, the characterization of AOP as "quantification and obliviousness" [7] which was correct for AspectJ is too limited for AOSD. The subtlety of the distinction between AOP and AOSD has led to an inclination by the wider software community to find aspects inappropriate or too limiting, and even to find AOSD's success to be "paradoxical" [27], so it is important to emphasize that AOSD researcher has always addressed a wider set of issues.

In fact, the wider set of issues of interest to the AOSD community is becoming ever more relevant to problems emerging in future computing environments. The computing world has undergone considerable change since the advent of object-oriented technology. From its early development, Aspect-Oriented technologies [6] have generally avoided the narrow view of the common object model in which a

1 It had been previously used in [25] for a role-like concept.

* This work is supported by a grant from Science Foundation, Ireland

message is sent to and handled by an isolated "target" object. This evasion is prominent in many emerging technologies as well. Service-Oriented Computing [32], Grid Computing [31], Ubiquitous Computing [33] and Complex-Event Processing [21] all provide a view of software in which the infrastructure re-routes messages or method calls from the apparent target to one or more real target objects. Various proxies are employed to filter or redirect messages. In some cases even, the point-to-point communication model for sending messages to objects is being replaced by a broadcast model in which messages are sent for delivery to whichever objects hold their implementation.

2. UNDERLYING ISSUES

Much work on aspect-oriented software deals with at least one of four issues. In roughly the order in which they emerged, these issues are:

- [aspects] representation of concern-separated artefacts and control of their interconnection or composition at points where they must join,
- [pointcuts] identification and exposure of appropriate points in concerns at which their behaviour should be joined,
- [context] characterization of complex behaviour and exploitation of inferred program state, and
- [mining] identification and possible extraction of concerns from tangled software.

Let us look in some more detail at the characteristics of the solutions proposed.

2.1 Aspects and Composition

Many AOSD approaches allow the state and behaviour of objects to be separated into elements that can be re-grouped into "subjects" [10] or "aspects" [16] to address separate issues of concern to the architecture or implementation. The various elements of objects in the separated concerns are put back together by a process of composition or weaving.

Several reasons have been put forward to motivate the importance of aspects:

- attachment of systemic[1] behaviour to objects, to support transactions, persistence, etc. [1][24]
- enhanced development characteristics such as simple extension and concurrent development of functional concerns like editing, verification, or display in an IDE [10]
- support for late selection and combination of functional features of product-lines [15]
- support for large-scale middleware construction [5]

In its broadest form, software aspects are generally modelled as a body of material which, when executed, produces events or cooperative method calls. In object-oriented formulations, aspects may be classes or may associate fields and methods with one or more classes. Aspects need not form complete programs, although some aspect-oriented approaches require completeness of some aspects.

Composition, or weaving, can be thought of as a form of dispatch, but one which employs more complex delivery rules than does the common target-directed model for objects, and one in which the message may result in behaviour that was defined in several aspects. While Subject-oriented Programming and Aspect-oriented Programming treat events in terms of a broadcast and focus on the end-points of message delivery, other work attempts to deal with the handling of messages in a way that allows each object along the path to influence the delivery targets [20] [11].

2.2 Pointcuts and Join-Points

The second prominent issue among AOSD researchers is the specification of points in an aspect at which the events or cooperative method calls originate. The call's points of origin form a set of "join-points" – the points at which aspects join. They generally provide the means by which the various aspects of an object's behaviour are woven together. In Subject-oriented Programming, the join-points were specifically identified to be method calls because they represent the points at which a software developer using subject technology might "expect" unknown or additional behaviour to take place [23]. At method-call points, subclasses may provide overridden or extended behaviour, or target objects might have extended behaviour. This restricted view avoided the breaking of encapsulation, but made development of subjects require the same careful thought about extensibility as does the development of frameworks. AspectJ [16] introduced the concept of a pointcut – a query-specified cut through a program that identifies a set of join-points. The pointcut query is associated with an advice, identifying which of the join-points originate messages to be delivered to the advice body. In AspectJ, the advices and their pointcuts lie outside the base aspect to which the queries are applied, in-effect "injecting" additional behaviour into the base. Because these join-points are injected without participation of the original developer, they have been criticised as breaking the encapsulation. But they also greatly increase the degree of "obliviousness" [7] and provide for a much wider spectrum of points at which aspects may join.

The underlying concept behind "pointcut" as a set of join points has evolved and variations are in use by authors of a number of approaches for AOSD:

[1] Often called non-functional, although most customers would prefer to reject software that does not function or provides no function.

- **Cooperative operation** [10], behaviours in aspects are attached to the set of execution points that are method calls intended to allow for extended behaviour
- **Pointcut** [16], a query identifying specified a set of points in execution, used as a clause in an aspect's advice declaration. Viewed as a query, the pointcut's variables are bound when matched to an instance. These bindings can be passed as parameters to be advice body. A **user-defined pointcut** is given a name and signature. The signature selects from the variables bound by the query. An **abstract pointcut** is a user-defined pointcut that omits the query. It may be used as a clause in an aspect's advice but must eventually be made concrete by associating it with a particular query that provides bindings for its parameters
- **Exported pointcut** [9], is a user-defined pointcut that is specified in a module of the base rather than in an attached aspect. This effectively produces a cooperative operation, but expressing it as a query over the module's content makes it useful for after-the-fact use in the module.
- **Methoid** [12] explicitly treats an exported pointcut as a call to a method. To do so, the query specifies a set of regions in the execution of material to which it is applied. The content of the region is treated as a method that can be reduced to a single join-point - a cooperative operation call to that method. The method can be extracted or materialized as to perform the behaviour identified by the query if needed to form compositions.

2.3 Context: Gross Program State, CFlow, and Complex-Event Processing

AspectJ's pointcuts included the ability to filter the circumstances of a join, on the basis of dynamic information much like the filters of composition filters [1]. Among the filtering criteria is the "cflow" construct. An important use of cflow is in determining the gross program state of a base so that the aspect can respond in an appropriate manner. In the example in [2], the authors employ a series of correlated pointcuts using cflow to track processing within wizards.

But cflow is a weak mechanism for attacking an important problem. The gross program state of a system is often a more complex function of its flow history than examination of the current flow stack will reveal. So it is advantageous to use an aspect follow the series of occurrences in a system that indicate a change in gross state This is done in [28], where the authors observe that "interesting states of the system can be described in terms of previous events and the ordering of them." The aspect can summarize the state in a variable used in the pointcuts of other aspects. Such an aspect can be used to perform the kind of task usually associated with complex-event processing [21].

2.4 Aspect Mining: Analysis, Identification and Extraction of Concerns

As the value of concern-separated software became more evident and greater tool support for its use became available, the importance of being able to deal with the concern structure of legacy software grew. Program-slicing had been of interest for a long while, but was generally viewed as a compilation or debugging technique, perhaps for lack of ability to reflect the sliced program as a proper software artefact. A good review of work in this area can be had in [3].

3. OPPORTUNITIES

Much service-oriented software today treats services like objects, with composite services managed as intermediate objects that route calls to the objects they use internally. Exploiting aspect-oriented constructs allowing us to treat services as behaviour attached to cooperative operations offers the opportunity to introduce a much more flexible and malleable service structure.

Software builders conventionally work as if they have a complete view of the software they are building. This point-of-view is reinforced by the constructs we use in thinking about problem decomposition. Perhaps the deepest is the "call" construct. It is traditional for a developer to look at a specification for some service and develop an algorithm for satisfying it, examining and selecting from available software components to perform services that the algorithm itself needs in turn. Traditionally, this examination looks far deeper into the component than any formal specification. The developer may look at issues like:

- the performance characteristics, perhaps determined by examining the code if not the present in the documentation,
- the malleability or extendibility of the code, perhaps reflecting the need to create subclasses or attach aspects,
- the "burden" – additional options provided that contribute to overhead but are not needed,
- the stability of the code base, reliability of its developer, etc.

This analysis takes place even if the component is a built-in element of the programming language, like those that perform arithmetic on numbers, but the characteristics become ingrained to form part of a developers' natural repertoire. Having selected a component, the developer often adapts the algorithm under development to reflect additional choices and capabilities potential in its use by, for example exploiting public or private knowledge about the logic, state representation, or class structure of the component being used. The resulting dependencies are called EEK in [29].

This is, to some extent, changing. There is an impetus to treat components as "services" [22], contracted for when software is executed, rather than when it is written. While it may seem a small change, the impact is enormous because the trade-offs described above can no longer be made by the developer, but must be made later when the "service-finder" is operative. Fully exploiting these changes requires a change in the programming languages we use to make the artefacts more malleable and the information on which they depend more manifest to the service-finding mechanisms. The Continuum programming language [30] is being used as the basis for researching both the language and the implementation issues involved in this shift.

Most aspect-oriented approaches lie somewhere in-between these extremes of early- and late-bound selection. In asymmetric approaches, binding activities are performed by the developer of the attached aspect rather than the developer of the base aspect. This effectively reverses the usual situation by having the service (aspect) developer become familiar with the details of the client (base).

Symmetric treatments of aspects forgo assigning responsibility to either component developer, and require the developer of composition rules to be familiar with both (or all) participating services. As it is for a software "service-finder," the developer's task is made simpler, or indeed possible, if the needed information is made explicit in the software rather than having to be dug out from its latent places in the code.

The following parts of this section explore how the generalisation from aspects to services provides potential for change in the way we deal with some important issues.

3.1 Classic vs. Co-operative Method Call

Much discussion of aspect attachment could be clarified by explicit recognition that the named, parameterized pointcut effectively defines a cooperative method call and that the rest of the mechanisms for dealing with aspects can be applied in general. Except for the issue of where the pointcuts are specified and applied, there is often no natural difference between the structures of the concerns themselves. Implementations of functionality can as easily be placed in one concern as another, in a way that reflects requirements rather than a dominant decomposition like "class". Attaching it as an aspect can yield equivalent results as keeping it in the base. In fact, it has been observed [26] that even "class" is just another dimension for separating concerns. This effectively points out that whether the developer of a class decides to put it in the base or to put it in a separate concern makes little difference to the operation of the base. Of course, depending on the AOSD approach in use, it may affect the syntactic expression.

No matter where the pointcuts are specified that expose them, the join-points in a concern effectively become points of cooperative method call [10]. From a mechanistic point-of-view, a cooperative method call can be thought of as identical to a classic method call. Classically, we think of a method call as belonging to (defined by) a client or consumer, the one who makes the call. But the cooperative method-calls in a concern are the join-points it exposes. Ordinary method-call is concerned with what the call will do for the client. But cooperative method call is concerned with what the call can do for the community as well. Behaviour provided in the originating concern or in any other concern cooperates to provide the actual behaviour associated with the cooperative call. So, from the point-of-view of system-structure, dispatch resolution, intention, specification, etc, classic method call and cooperative method call have quite the opposite conceptions even though they are mechanically identical. Potential impact on several of these areas is addressed in the following sub-sections. A common way to look at their mechanical similarity is discussed in Section 3.2. The fact that this reversal can exploit more information about intent is discussed in Section 3.3. Section 3.4 explores potential for availing of greater concurrency in the software we write, and Sections 3.5 and 3.6 discuss language support that both increases dynamicity and malleability and enables the more concurrent style.

3.2 Event Flow and the Dual Role Of The Base

The role of the "base" is perhaps the most vexing issue in treating AOSD [19] [13]. Virtually all approaches connect the behaviour in the separated concerns with cooperative method calls in the base, specified implicitly or via pointcuts. The base acts both as a body of code making cooperative method calls to which aspect behaviour can be attached and as an aspect providing some of the behaviour for them. In order to accommodate the diverse collection of emerging technologies that can all benefit from aspect-oriented approaches, we should separate event behaviour attachment from the description of overall event flow. In this view, the base contains no code itself. It is a specification of a sea of events on which the aspects float, each associating its behaviour with some of the events. The differentiation of cooperative method calls from classic method calls becomes the role of the base. This view intends to accommodate either view of joinpoints – that they are intended or that they are injected, as explored further in Section 3.3

Separating the abstract model of the underlying flow of behaviour from the attachment of providers' behaviour can give some insight into the role of pointcuts and their relationship to the base. Specifically, we can construct the "base" from the collection of abstract pointcuts whose behaviour is provided by the aspects. In Figure 1, we show an aspect concern both exporting some pointcuts (cooperative method calls) and providing behaviour for others. The base could be specified separately, as part of an

overall system design, or could be derived from information in the aspects making up the system. In either case, it could be thought of as a simple list of events identified as abstract pointcuts, a constrained event specification, a work-flow diagram, or as hinted at in [28], in the form of the sequence diagrams forming the system's design.

```
aspect x;
export {
   pointcut p(int a):
            call(* X.foo(int a));
}
provide {pointcut n(int a, real b);}
class Y {when[2] p(int a) {...}}
```

Figure 1- a symmetric aspect

In addition to the specification of exported and provided pointcuts, the aspect contains behaviour that is attached to the events and is subject to the exports. For an aspect to be attached to a base, its specification of the cooperative calls required must be consonant with those of the base An aspect not in the base has no exported pointcuts. A base with no advice for other aspects has no "provides". While not exploiting this syntax, Continuum's "service" construct explicitly lists the cooperative behaviours provided and describes their dependencies on earlier flow.

3.3 Whose Is The Specification - Join Points By Intention or Injection

In a classic call, the client developer keeps in mind the services needed (e.g. "a hash table into which objects can be put") while finding a suitable implementation for the service. In many object-oriented languages this decision is consolidated by naming the pre-existing class or interface that is associated with the selected implementation. When providing a local implementation, the class or interface created by the developer may be sketchy, or it may be well documented and meet the expected standards for reusable software. But in all cases, the focus is on what the service does or on what the client needs.

In cooperative call, there is more that needs to be said: what the client is doing in a cooperative sense. A call to hash.put(…) may have been written because of the intention "put a book into the library records". It is this intention which is the link that ties the cooperating concerns together. When written with respect to a particular base, a pointcut specification needs to supplement the call, to fill-in just this information about intention.

We say that the purpose of pointcuts, whether injected or intentionally exported, is to distinguish cooperative calls from ones that are hidden from cooperative attachment. But distinguishing the cooperative calls does not suffice to provide the specification of their intention. If we expect the cooperator to be found by a mechanical service-finder rather than by the client developer, it is clear that additional documentation must be available. In fact, the entire issue of malleability of call structures becomes more critical for cooperative calls than for hidden ones. The same method, identified by its name and signature, may be used to support many different intentions. This argues that the intention and characterization information must be separately attached to the name. In the interest of service-finding, we can employ glossary or ontology references associated with methods and their parameters to supply information concerning:

- the actual intention expressed at the cooperative call
- the separate functional expectations of a call so that they might be realised by separate aspects composed later
- the functionality provided by other aspects available
- the roles of parameters of the call in an order-free manner to give greater flexibility in matching server to client

Not all join-points need be originally written as method-calls. Those which are not must to be injected or exported, as mentioned in Section 2.2, using a pointcut to turn them into cooperative method calls. The information about intention can be supplied at the point where the pointcut is defined.

The issue of control of the specification is closely related to a phenomenon that could be termed "function bundling". Function bundling reflects the fact that many methods perform a multiplicity of functions. For example, an analysis of the Unix "sort" command was conducted and it was found to be reasonably represented as the composition of 30 concerns [4]. Bundling of this sort is often signalled by the presence of option parameters or of optional parameters – reference parameters that may be null. The bundling reflects the developer's statement of the specification as a "maximum" for the component being developed. It often contains excessive functionality contributing to the bloat of its clients [8]. An aspect-oriented realization could encourage independent functions to be presented in separate aspects, combined later by the service-finder at run-time rather than by the developer at development time.

3.4 Raising Concurrency with Aspects

3.4.1 Attaching Aspects to Events

The fact that we are reaching fundamental limits in increasing the speed of sequential processors indicates a growing need to increase the parallelism and asynchrony that is potentially available even in the ordinary software we write. One way to go about this is to bring the use of asynchronous events more into the mainstream by simplifying programming language constructs to encourage

[2] "When" is used as an alternative to before, after, event, around, etc. denoting behaviour to be performed sometime between before and after but not needing to be wrapped around other behaviour.

their use. The metaphor of software as aspects floating on a sea of events may offer us an opportunity to do so because it emphasizes the attachment of behaviour to events rather than the construction of sequences of control to arrange for their execution. While it is important to preserve the flexible synchronous combinators, like before, after, with, around, etc., used for aspect behaviour, not all aspect behaviour needs to complete before the cooperative method call that calls for it can continue. For example, the behaviour provided by a logging aspect can often be attached as an event since the continuation of the base does not depend on its early completion. Event attachment is irrevocably concurrent. The originating client can have no expectation about the time of its execution, which may even be deferred until the client completes.

3.4.2 Sending Events and Passing Commitments

Attachment of aspects as events is a helpful first step, but does little to encourage more concurrency within the base itself. A second step forward is to permit cooperative method calls to be sent as explicit events. All recipients run concurrently with the continuation of the base. The declaration of method one in Figure 2 suggests how a commitment for the eventual invocation of an event can be passed declaratively from one method to another. The "sends" clause in the declaration of method one indicates that it is committed to the ultimate sending of event two, either on its own or by passing the commitment forward. In Figure 2 the commitment might be passed to method three (assuming its unshown declaration has a similar "sends" clause).

3.4.3 Future Event Handling

People describe problem solutions sequentially, although they can break off chunks described to be done concurrently. And they seldom think of subtasks as subject to long potential delays. The ability to send method calls asynchronously is not a new construct, and like asynchronous aspect attachment yields only a small improvement in the overall concurrency behaviour of software. Both require the software developer to break the train of sequential thought, and both require the high intellectual overhead of creating new classes, methods, etc. We need to provide developers with a construct that allows them to think sequentially about activities that can be deferred or executed concurrently. We can build on the concept of passing commitment to provide it. In Figure 2, imagine that the developer knows that after doing method four, some other tasks must be performed. Perhaps method four makes a bank transfer, and a receipt must then be presented. This is a sequential thought that would generally be represented by invoking method four and then performing the receipt processing, shown as "…". We want to allow the developer to express the sequential dependency without holding up the return from method one. (If we imagine method one is called inside a loop, then allowing it to return without waiting for method four to execute means that many executions of method four are started by the loop, and all can run concurrently.) But syntactically, we avoid interrupting the developer's train of expression, by allowing the receipt processing to be written as part of the call to method four using a commitment that it will eventually send message five.

```
void one(Object x,int y)
     sends two(Object m, real z) {
  send three(this,"hello");
  send four(this, 6)
  expect five(MyClass this, int a) {...}
}
```

Figure 2 – preserving sequence without synchrony

Eventually, the commitment is met and message five is sent. Its implementation is as specified in the "expect" clause, and the receipt is printed. Note that method five cannot access any of the local state of method one, which may be long gone. But it can use its parameters to access object state as usual. The mechanisms for doing this and the meaning of the "`MyClass this`" parameter declaration are part of with the service model of aspects employed in Continuum and with its model for dynamically extending knowledge about the methods supported by classes. These are described briefly in Sections 3.5 and 3.6.

3.4.4 Exceptions

Any construct like "send", that decouples future execution but still provides for satisfaction of commitments, must address the problem of exceptions and failures. Since a sent message may commit the future invocation of another event, we must define what happens if it fails to do so. This can be addressed at two levels. On the static level, a method declared to send some event must do so on all execution paths. This can be checked at compile and load time. On the dynamic level, a logic error may still prevent the future "send" from taking place by causing an exception to be thrown. When an exception is thrown, potentially unsatisfied commitments to send an event are satisfied by sending the event in such a way as to trigger the exception immediately on entry to the called method.

3.5 Generalizing Aspects as Services

Ever since the programming language community adopted the target-directed method invocation model for the dominant languages, software developers have been compelled to escape from it by moving dispatch from the language to the middleware. We see this escape in the all the emerging technologies mentioned above: Aspect-Oriented, Service-Oriented, Grid, Ubiquitous, and Complex Event Processing. In the infrastructure for all of these, method calls are directed to objects other than the "target" presented by the client. Widening the concept of "aspect" provides the opportunity to address this need. The

programming language Continuum makes the escape explicit by integrating with the language and VM a dispatcher whose mechanisms are constrained to assure delivery and freedom from ambiguity but are otherwise explicitly unspecified [14]. This is a step beyond the flexibility introduced by aspects, most approaches to which still hold the definition of dispatch closely.

Continuum introduces the "service" construct into the language. Its role is to provide methods with access to the state of one or more the objects they are passed as parameters. This process is called decapsulation and can only be applied to parameters of the method's class – the same ones that govern dynamic dispatch to the method itself. Like advices, the methods provided by services are added to the behaviour performed when the cooperative method is invoked. Services have several other roles. They act as encapsulation boundaries, with explicit specification of the cooperative methods and events they support and require, They also draw boundaries that contain the propagation of ambiguity so that it remains within a service. This limits the scope of the checking that must be performed when a class is added to a service. Services each have an independent representation of objects' states, so that references to objects may be represented as other than opaque pointers without losing their ability to convey information assuring support for methods. Non-opaque references can be passed between services that are distributed around the network.

3.6 Overcoming the Drawbacks of Obliviousness

Much work has been done on the dynamic introduction of aspects, but if the base is intended to be "oblivious," it is hard to extend this work to use aspects as general dynamic service providers – service calls are obvious rather than oblivious in the client. Dynamic service provision suggests that the client knows about the methods and expects to call them even though the methods' implementations are not available when the client is started. The service model described above provides for the dynamic introduction of services. To complement it, Continuum's assurance model is also dynamic. A class does not specify or limit the interfaces that it supports, allowing services to add to growing knowledge about which methods are supported [14].

```
void meth(Store{put(Store,Item)} store1,
          Store store2);
Store{put(Store,Item),
      boolean inStock(Item,Store)} more;
more =
 ({boolean inStock(Item,Store)}) store1;
more = store2;
boolean t = inStock(item,store2);
```

Figure 3 – dynamic knowledge about classes

In Figure 3, the first assignment statement to "more" adds the knowledge that "inStock" is assured safe to call with ***any*** object in the class Store, allowing this knowledge to be transferred later to store2. Each service may provide methods and implementations for various classes, based on the state information the service possesses. Services can even provide methods that do not require explicit knowledge of state, but simply depend on access to that state provided by other services.

Clients do not know which service provides a method, nor even which class the method is "in". This is because continuum's assurance model is symmetric, which means that a service can provide a method dispatched on a parameter other than some designated target. The assurance that the method can be safely called is passed through a generalized concept of interface, shown in Figure 3, even though the implementation need not lie in the object to whose reference the interface is attached. Hence, even though the knowledge that inStock is assured is associated with "store2", the implementation may actually reside in "item". This helps make the client structure less dependent on the implementation structure because the client does not need to know in which objects methods are implemented.

4. CONCLUSIONS AND DIRECTIONS

Aspect-oriented software is broadly focused on the language and support for separating the design and implementation work required for differently-motivated concerns. In conventional object-oriented software, these are often tangled within a single class or method. While the language features of some aspect-oriented approaches emphasize its use for post-facto attachment of functionality obliviously, others employ it to achieve planned separation for flexibility, as within product lines, or to achieve concurrency and event-processing. We are entering a computing environment that is radically changing to address the needs of mobility and the challenges of physical limits on processor speed, and in which broadband services are encroaching on the traditional point-to-point structures. In such an environment, there is much room for growth and exploration in the use of aspect-like constructs to adapt software to changing environments. We have outlined some issues that need to be addressed and some approaches that may address them in the hope that the community can widen the scope of its thinking about the applicability of aspects and concern-separated software as we move forward.

5. REFERENCES

[1] Aksit, M., Bergmans, L., Vural, S., An object-oriented language-database integration model: The composition filters approach. In *Proc. ECOOP'92, Springer Verlag, LNCS 615*

[2] Bodkin, R., Furlong, J., Gathering Feedback on User Behaviour using AspectJ. in *AOSD 2006 - Industry Track Proceedings*, Chapman, M., Vasseur, A., Kniesel, G. (Eds.), Technical Report IAI-TR-2006-3,

ISSN 0944-8535, Computer Science Department III, University of Bonn, March 2006

[3] Breu, S., Moonen, L., Bruntink, M., and Krinke, J. (Eds.), *Proceedings First International Workshop Towards Evaluation of Aspect Mining*. July 4, 2006, Nantes, France, Delft University of Technology Software Engineering Research Group Technical Report TUD-SERG-2006-012

[4] Carver, L., *Building Real-World Applications with Aspect-Oriented Modules and Hyper/J*. Master's thesis, Univeristy of Califorina, San Diego, Department of Computer Science and Engineering, June 2002

[5] Colyer, A., and Clement, C., Large-Scale AOSD for Middleware. *Proceedings of the 3rd international Conference on Aspect-Oriented Software Development* (AOSD 2004), Lancaster, UK, March 22-26, 2004, ACM, New York (2004), pp. 56-65

[6] Elrad, T, Filman, R. E., Bader, A. Aspect-oriented programming: Introduction. *Communications of the ACM, Volume 44 Issue 10, October 2001*

[7] Filman, R.E. and Friedman, D.P, Aspect-Oriented Programming is Quantification and Obliviousness. *In Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications,* Minneapolis, Minnesota, October 2000

[8] Garlan, D., Allen, R., Ockerbloom, J., Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, vol. 12, no. 6, Nov., 1995

[9] Gudmundson, S., and Kiczales, G., Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Proc. ECOOP 2001 Workshop on Advanced Separation of Concerns*, July 2001.

[10] Harrison, W. and Ossher, H., Subject-Oriented Programming - A Critique of Pure Objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993

[11] Harrison, W. and Ossher, H., *Structure-bound Messages*. IBM Research Report RC 15539, March, 1990.

[12] Harrison, W., Ossher, H., Tarr, P., General Composition of Software Artifacts. *Proceedings of Software Composition Workshop 2006*, March 2006, Springer-Verlag, LNCS 4089

[13] Harrison, W., Ossher, H., Tarr, P., *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. Research Report RC22685, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December, 2002

[14] Harrrison, W., Lievens, D., Walsh, T., *Using Recombinance to Improve Modularity*. Software Structures Group Report #104, Computer Science Department, Trinity College, Dublin, March, 2007

[15] Jansen, A., Smedinga, R., van Gurp, J. and Bosch, J., First class feature abstractions for product derivation. *IEE Proceedings on Software*, Volume: 151, Issue: 4,Aug. 2004

[16] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, Jeffrey Palm and William G. Griswold, An Overview of AspectJ. *Proc. 15th European Conference on Object-Oriented Programming*, 327-353 (2001).

[17] Kiczales, G., Lamping, J., Lopes, C., Hugunin, J., Hilsdale, E., Boyapati, C., *Aspect-oriented programming*. United States Patent 6,467,086, October 15, 2002

[18] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., *Aspect-Oriented Programming*. In Proc. ECOOP'97 (Finland, June 1997) Springer-Verlag

[19] Lamping, J., The Role of the Base in Aspect Oriented Programming. In *Proceedings ECOOP Workshops* 1999: 289-291

[20] Lieberherr, K. J., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996

[21] Luckham, D., *The Power of Events*. Addison Wesley Professional, May 2002, ISBN: 0201727897

[22] Mendonpa, N.C.; Silva, C.F., Aspectual services: unifying service- and aspect-oriented software development. *International Conference on Next Generation Web Services Practices*, Aug. 2005

[23] Ossher, H. and Tarr, P., Operation-level composition: A case in (join) point. In *ECOOP '98 Workshop Reader*, 406–409, July 1998. Springer Verlag. LNCS 1543

[24] Rashid, A., Chitchyan, R., *Persistence as an Aspect*. Proc. of International Conference on Aspect-Oriented Software Development (AOSD 2003), March 2003, Boston, MA

[25] Richardson, J. and Schwarz, P., Aspects: Extending Objects to Support Multiple, Independent Roles. *Proc. ACM-SIGMOD Conf.,* Denver, Colorado, May 1991

[26] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M., "N degrees of separation: Multi-dimensional separation of concerns." In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99),* 107–119, IEEE, May 1999

[27] Steimann, F., The paradoxical success of aspect-oriented programming, In *SIGPLAN Notices*, Vol. 41, No. 10. (October 2006), pp. 481-497.

[28] Walker, R. and Murphy, G., Joinpoints as Ordered Events: Towards Applying Implicit Context to Aspect Orientation. *Proc. ASOC Workshop at ICSE* 2001

[29] Walker, R. and Murphy, G.. Implicit context: Easing software evolution and reuse. In *8th International Symposium on the Foundations of Software Engineering*, San Diego, CA, USA, November 2000

[30] Continuum Draft Language Specification available from *https://www.cs.tcd.ie/research_groups/ssg*

[31] Grid Computing overview available at *http://en.wikipedia.org/wiki/Grid_computing*

[32] Service Oriented Architecture overview available at *http://en.wikipedia.org/wiki/Service-oriented_architecture*

[33] Ubiquitous Computing overview available at *http://en.wikipedia.org/wiki/Ubiquitous_computing*