FOAL 2004 Proceedings

Foundations of Aspect-Oriented Languages Workshop at AOSD 2004

Curtis Clifton, Ralf Lämmel, and Gary T. Leavens (editors)

TR #04-04
March 2004

Keywords: Aspect-oriented Programming Languages, Aspects, Pointcuts, AspectJ, Static Analysis, Modular Reasoning, Specification, Verification, TinyAspect, Open Modules, Bisimulation, Encapsulation, Formal Semantics, Program Slicing, Implicit Invocation, Model Checking, Horizontal Architectures, Superposition, Reactive Systems, Synchronous Languages

2003 CR Categories: D.3.1 [*Programming Languages*] Formal Definitions and Theory—semantics; D.3.2 [*Programming Languages*] Language classifications—object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs and Features—control structures, data types and structures, abstract data types, polymorphism; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs—assertions, logics of programs, pre- and post-conditions, specification techniques; F.3.2 [*Logics and Meaning of Programs*] Semantics of programming languages—operational semantics, program analysis;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Contents

Preface

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The third Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Third International Conference on Aspect-Oriented Software Development in Lancaster, UK, on March 23, 2004. 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, but not limited to: 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 papers at the workshop, which are included in the proceedings, were selected from papers submitted by researchers worldwide. Due to time limitations at the workshop, not all of the submitted papers were selected for presentation. FOAL also welcomed an invited talk by James Riely (DePaul University), the abstract of which is included below.

The workshop was organized by Gary T. Leavens (Iowa State University), Ralf Lämmel (CWI and Vrije Universiteit, Amsterdam), and Curtis Clifton (Iowa State University). The program committee was chaired by Lämmel and included Lämmel, Leavens, Clifton, Lodewijk Bergmans (University of Twente), John Tang Boyland (University of Wisconsin, Milwaukee), William R. Cook (University of Texas at Austin), Tzilla Elrad (Illinois Institute of Technology), Kathleen Fisher (AT&T Labs–Research), Radha Jagadeesan (DePaul University), Shmuel Katz (Technion–Israel Institute of Technology), Shriram Krishnamurthi (Brown University), Mira Mezini (Darmstadt University of Technology), Todd Millstein (University of California, Los Angeles), Benjamin C. Pierce (University of Pennsylvania), Henny Sipma (Stanford University), Mario Südholt (École des Mines de Nantes), and David Walker (Princeton University). We thank the organizers of AOSD 2004 for hosting the workshop.

Invited Talk—Formal AOP: Opportunities Abound

James Riely (DePaul University)

Aspect-oriented programming (AOP) is a polarizing paradigm—attractive to some and repulsive to others. On the up side, AOP allows for new forms of encapsulation: each concern (e.g., functionality, security) can be coded separately rather than interleaved in the same code base. On the down side, AOP destroys old forms of encapsulation: straight-line code no longer runs in a straight line. Static weaving has proven eminently practical, but many potential applications depend upon dynamic properties which do not lend themselves to this implementation technique. To realize its full potential, AOP must become both more controlled and more expressive. In each case, the issues are complex and difficult to arbitrate without formal models.

Diagnosis of Harmful Aspects Using Regression Verification

Shmuel Katz
Computer Science
The Technion
Haifa Israel
katz@cs.technion.ac.il

ABSTRACT
Aspects are intended to add needed functionality to a system or to treat concerns of the system by augmenting or changing the existing code in a manner that cross-cuts the usual class or process hierarchy. However, sometimes aspects can invalidate some of the already existing desirable properties of the system. This paper shows how to automatically identify such situations. The importance of specifications of the underlying system is emphasized, and shown to clarify the degree of obliviousness appropriate for aspects. The use of regression testing is considered, and regression verification is recommended instead, with possible division into static analysis, deductive proofs, and aspect validation using model checking.

Static analysis of only the aspect code is effective when strongly typed and clearly parameterized aspect languages are used. Spectative aspects can then be identified, and imply absence of harm for all safety and liveness properties involving only the variables and fields of the original system. Deductive proofs can be extended to show inductive invariants are not harmed by an aspect, also by treating only the aspect code. Aspect validation to establish lack of harm is defined and suggested as an optimal approach when the entire augmented system with the aspect woven in must be considered.

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

General Terms
Languages, Verification.

Keywords
Aspects, desired specification properties, noninterference, preventing harm, regression verification, aspect validation.

1. INTRODUCTION
Like all modularity and language concepts, aspects are intended to improve the development of complex systems. On the code level, Aspect-Oriented Programming (AOP) languages provide notations to separately declare and repeatedly apply aspects that cross-cut the usual class structure of object-oriented systems. Using AOP has already been shown in numerous case studies to isolate the treatment of concerns that otherwise are scattered throughout the system, and tangled with code treating a variety of application issues. However, it is clear that sometimes such augmentations of systems can make properties that previously held for the system become untrue in the combination of the system with the aspect.

Such changes in the properties of the system could be a proper outcome of applying the aspect if the property is considered undesirable, such as that the system deadlocked in certain situations, or that messages were visible to any other observer in the computer. On the other hand, in general there is no way to linguistically prevent aspects from invalidating some properties that *are* desirable. This could occur either inadvertently, or maliciously. An example of the former could be when an aspect intended to treat overflow of variables, by mistake also causes the system to deadlock. An example of the latter could be when a system with private fields that guarantee some level of privacy is augmented by an aspect that provides public methods for reading the values of those very fields, in order to expose their contents, thereby violating the desired level of privacy.

In order to identify and treat such situations, the systems to which aspects are woven need to be augmented with *specifications.* These are descriptions of the desirable properties of the system. Note that they do not describe *all* properties of the system, only those seen as important and positive. Such properties should be maintained even if the system is augmented with aspects, or even if an aspect is combined with other aspects. What *can* change are the

properties of the system not seen in the specification. The form of such specifications is described in Section 2.

Treatment of harmful aspects also requires a rethinking of the degree of *obliviousness* needed by an aspect-oriented notation. Obliviousness has traditionally [2][3] been seen as a desirable feature of Aspect-oriented notations. Although several definitions are possible, all imply that the underlying system does not have to prepare any hooks, or in any way depend on the intention to apply an aspect over it. The application of an aspect adds new features to a system, but the system without the aspect has its own specification and is correct relative to that specification, without needing any aspects.

Obliviousness is clearly important in dynamically evolving systems, where the aspects may not even have been thought of when the original system was created. It also is appropriate when a system can have many variants, some with one collection of aspects, and some with another, each configured for a user's particular needs. This is one of the potential uses of aspects to allow more flexible components, configurable on demand.

However, a total obliviousness to aspects prevents treating such malicious aspects as the one that reveals values intended to be kept private. Who prevents the application of such an aspect, on the language/system level (as opposed to locking the source in a safe, and physically preventing access to it)?

If specifications are available, a middle ground is possible, where a system is oblivious to the particular aspects to be applied to it, but still can restrict new aspects to those that do not violate its specification (or at least some parts of its specification). An aspect will be considered harmful if it invalidates any desired properties of the system to which it is applied. This will be more precisely defined and justified in Section 2.

The paths open to diagnosis of harmful aspects are usual testing, static code analysis similar to that done by type-checkers, and use of formal methods, both deductive verification and model checking. We shall consider all possibilities. The type of augmentation or change made by an aspect is another dimension that can determine the best way of preventing harm. The three basic divisions [6] are to *spectative* aspects that only gather information about the system to which they are woven, usually by adding fields and methods, but do not influence the possible underlying computations otherwise, *regulatory* aspects that change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields, and *invasive* aspects that do change values of existing fields (but still should not invalidate desirable properties).

Yet another question is whether only the aspect module itself must be analyzed, independently of any system to which it may be woven, or whether an entire system augmented by an instance of the woven aspect is the object of analysis. In the continuation, the former is called *aspect code* analysis, and the latter *augmented system* analysis. The system before an aspect is woven into it is termed the *original* system.

The focus on preventing harmful effects of aspects is unusual, but as will be shown, does allow a uniform treatment. Such a treatment is more difficult when the new properties to be established by the aspect also need to be taken into consideration. Taking a medical analogy, the basic principle should first be, as in the doctor's Hippocratic oath: "Do no Harm."

2. SPECIFICATIONS OF ASPECTS AND SYSTEMS

A full treatment of aspects and their compositions clearly does deal with the specifications of the aspects themselves, and not just of the underlying system. In a HyperJ view, the entire system is composed of such aspects, or concerns. However, such specifications are often difficult to construct. Aspects on a code level are typically described by defining *joinpoints* where changes are to be made, and *advice*, with code to augment or replace what is done at the original joinpoint. Note that joinpoints may be defined as dynamically determinable events, and not merely locations in code or method calls.

As already defined in earlier works[6], specifications of aspects need to describe both what is assumed about any object or method in the basic system to which the aspect may be applied (and in general, what must be true at each joinpoint identified by the aspect), and, on the other hand, what is required to be true after the advice is applied, if the needed assumption indeed holds at the joinpoint. For each joinpoint and advice segment of code, the advice assumes some property of the system, and guarantees some property when it finishes. Such an assume-guarantee structure for aspects has already been recognized in [1], and [7], and is essential for describing the added value of an aspect. The overall properties added by the aspect can also be globally described. Since many aspects deal with so-called non-functional concerns like availability, fault-tolerance, security, or persistence, providing their specifications is that much more difficult.

Here, however, we concentrate on simply avoiding harm, and thus are not interested in what new properties are promised by the aspect. Only the specification of the system to which the aspect is woven is needed to prove the absence

of harm. Since that specification usually deals with basic functional properties, it is more amenable to a description in standard temporal logic, and/or using precondition/postcondition pairs around methods or functions.

The obliviousness of systems to aspects is reflected in that usually the underlying system does not make assumptions of any kind about the possible aspects that may be applied. The existence of a specification of the desired properties that hold for the basic system provide a way to weaken obliviousness while maintaining the desired characteristics of extensibility and flexibility to add new unanticipated aspects. The specification of the basic system, in addition to restricting the implementation of the system, also can restrict future aspects, either by default —guaranteeing that all the desired properties in the basic specification will be maintained after weaving an aspect--- or in a more restricted version, where only some of the original desired properties are designated as unchangeable. Thus, for purposes of avoiding harm, the only requirement of the aspect is that the desired properties of the basic system expressed in its specification remain true when the aspect code is woven into the basic system and the augmented system is then executed.

Although not essential to the arguments in this paper, temporal logic provides a convenient formal notation for describing properties of execution sequences. In the simplest version G stands for 'globally' meaning from now on in the sequence of states, and F stands for 'in the future', meaning that eventually there is a state. Thus an assertion G(p => Fq) means that in every state, if p is true then eventually there will be another state with q. If p represents "a request has been made", while q is "a response is given", this corresponds to a specification that every request has a later response. Note that a counterexample to such an assertion would involve showing a computation with a state where p is true, but which never has a later state with q true. Whatever specification notation is used, it should not allow expressing assertions about immediately following states (using, for example, the "next-state" temporal modality X), since such assertions are known to be sensitive to any refinements or additions, and will be violated by any aspect that adds computation at problematic points. Thus we require a "stutter-free" version of temporal logic [5].

3. REGRESSION TESTING AND ITS LIMITATIONS

A straightforward approach to detecting harmful aspects would seemingly be the use of *regression testing*. The idea is simply to retest a system every time a new aspect is woven into it, to ensure that the test suite which previously was passed (and presumably captures the desirable outcomes that should be maintained) is still passed. Then the new properties to be added by the aspect could later be validated with new additional tests to be added to the test suite. This is the technique used by Extreme Programming (XP) [8] in place of having specifications, and is intended in XP to be applied to any significant change (e.g., a new version) in the system. However, there are several serious drawbacks to this approach when applied to aspects and their weaving:

First, regression testing is most easily applied to systems to which spectative aspects have been woven, where the aspects do not influence the computations of the underlying system at all. A regression test then could reasonably expect that the fields of the underlying system are unaffected by the augmentation of the aspect, so the results of the tests are unchanged. A violation is then trivially determined by comparing the results of the test, and can be inspected automatically. Yet when spectative aspects are used, it is more efficient to determine such situations using static analysis, as described in the next subsection. When the aspect is regulative or invasive, and thus *does* affect the computation, the results of the test will differ from the same test applied to the original system. They thus are often difficult to evaluate, and any violation cannot be determined automatically simply by detecting changes.

Second, this approach obviously relates to the entire augmented system, and retesting the entire system every time an aspect is applied is often unfeasible due to time or resource constraints. For a complex system, it seems overkill to activate the entire test suite even if an aspect with presumably small changes to only some of the objects and methods is added. Also, when aspects are taken from a library and bound to new systems, such a small investment in coding (binding the aspect to a system) hardly justifies an entire activation of the test suite. Moreover, if aspects are applied and removed dynamically, during run time, retesting is not realistic.

Third, and most significantly, the original tests obviously did not take into account the structure of the aspect or the influence it may have on the basic computation paths. Thus, for example when conditionals appear in aspect code, the original tests may be completely irrelevant, since new paths are generated and followed, and the tests may miss many computation paths. Precisely because of their cross-cutting nature, it is difficult to isolate parts of the test suite that still might be relevant, since many regular modules (e.g., classes) are affected by each aspect.

Therefore, simply using regression testing does not adequately treat harmful aspects. We thus turn to regression verification, which can be based on static type analysis, deductive verification, or model checking using aspect validation.

4. STATIC ANALYSIS

As noted above, when the aspect to be applied is spectative relative to the underlying system, it should often be possible

to establish this fact using static analysis of code. As will be discussed, in some languages the aspect can be analyzed in isolation, while in others the augmented system must be considered. A spectative aspect does not change either the value of any field or the flow of method calls of the underlying system. New fields, methods, and even classes can be added, but the new model of computation has a very particular relation to the underlying one without the aspect. Each computation path has sections of original computation interleaved with sections of new computation. The result is always equivalent to temporarily suspending the underlying system, recording some information about it, computing new values not influencing the underlying system in any way, and then continuing as before.

Such a situation might be difficult to detect directly on the execution graph of the computation, but it is amenable to detection on the code level in some aspect languages, using standard type checking and data-flow techniques. The idea is that the locally defined fields of the aspect are the only ones computed by that aspect, and no assignments are made by aspect code to fields or to parameters that can be bound to fields, variables, or parameters of the basic system. The aspect code also cannot "redirect" the flow of execution, and simply adds to the previous system without skipping any of its computation.

This situation is amenable to syntactic detection by analyzing only the aspect if all bindings between fields or variables of the aspect and the basic system are made through parameters of the aspect. On the other hand, when arbitrary binding is possible, for example by using the same name in both code segments, then only when a specific binding has been made can the augmented system be analyzed to determine which elements are bound, and whether the aspect is spectative. In either case, dataflow techniques, such as the *uses* and the *defined-use* pairs of standard code optimization, can be employed to determine whether there is any influence of fields in an aspect on those of the basic system (the other direction is, of course, not a problem). The possibility of analyzing just the aspect is one argument in favor of clearly identifying parameters for weaving, rather than allowing free bindings that force analysis of the entire augmented system.

Showing that an aspect is spectative is one way to guarantee that all safety and liveness properties involving assertions only about variables, fields, and methods of the underlying system will not be influenced by the aspect (as already explained, without assertions about "next" states). However, it should be noted that properties such as "the value of a field is not visible outside the class" can be violated by spectative aspects, even when they were previously true. The problem is that the assertion of "not visible" involves both the original fields and methods *and* new fields or methods added by the aspect. As already noted in the Introduction, a (hidden) field X could be "made visible" by examining another field Y (added by the aspect) linked to X by an invariant, or by adding new public methods.

Such data-flow and type-safety techniques are always conservative, in that if successful, the spectative nature of the aspect is guaranteed, and the aspect can cause no harm for specification properties as described above. If the analysis does not establish that the aspect is spectative, it remains to be seen whether the aspect is actually harmful.

5. DEDUCTIVE PROOFS OF CLASSES OF TEMPORAL PROPERTIES

It is also possible to establish a lack of harm for either specific properties or entire classes of properties using deductive proofs only over the aspect code. For example, an invariant of the original system can often be shown to also be an invariant of the augmented system, even without analyzing in what situations the aspect code will be applied. This is true when the invariant I is what is known as "inductive," meaning that {I} s {I} can be shown for each individual step s. Note that it is sufficient to show that if the invariant is assumed before a step, it will again be true at the end of the step. In this situation, to establish that I is also an invariant of the augmented system, it is sufficient to check that each aspect action t also satisfies the same assertion {I} t {I}. Since I is already known to be an invariant of the original system, it actually *is* true of the augmented system whenever the aspect is first applied, even without analyzing the joinpoints. By induction, it is easy to see that I will hold whenever some t action is taken, so will be an invariant of the augmented system, even without rechecking the original code.

For example, consider a situation where x>y>0 is an invariant of a system, and an aspect has changes of the form

<complex> ➔ double (x,y),

where <complex> is a complex condition for applicability, and double(x,y) doubles the values of x and of y. Then we easily have {x>y>0} double(x,y) {x>y>0}, extending the invariant to the augmented system, even though only the aspect code was newly analyzed, and when it is applied was ignored.

It is also possible to prove that an aspect is "almost spectative" in that it might only abort an underlying system, but would not otherwise affect the computation of the original statespace. In such a situation, liveness properties of the underlying system might be harmed, but all safety properties are maintained.

Consider an aspect that treats overflow for variables in one part of a system with limited memory. An invariant of the underlying system that is also in its specification could be that x = y. However, this will no longer hold in the augmented system if x is treated for overflow, resulting in new assignments to x, while y is not. In this case the aspect has harmed the system by violating a safety assertion of its specification. On the other hand, if the aspect stops the system when overflow is detected, rather than continuing as above, then safety properties *are* maintained, as long as the system continues.

6. REGRESSION ASPECT VALIDATION FOR INVASIVE ASPECTS

The approach of aspect validation, first suggested in [4], can be specialized to detecting harmful aspects. The idea of validation is to prove that each individual weaving is acceptable, rather than having a single generic proof that the aspect always does no harm. It is effective when each weaving of an aspect triggers automatic generation of verification tasks that themselves are automatically checked, e.g., using a model checking tool. Note that the initial organization and set-up of the validation framework for each application aspect can be non-algorithmic and require human effort and invention. However, the validation associated with each weaving of the aspect does not require such intervention, and must be automatic.

Aspect validation is appropriate when we cannot successfully identify absence of harm syntactically or statically by analyzing the aspect code, and yet concluding about lack of harm for classes of properties and for every possible weaving of the aspect. Thus we are forced to turn to techniques that analyze the augmented system, rather than just the aspect code. Indeed, in general we need to know the binding of the aspect to the basic system, and the properties which are the desired ones of that system, before the absence of harm can be established. Then we need to verify that those properties hold of the augmented system. The automatic verification for each weaving is essential to make this approach feasible.

In many cases a software model checker can be used to generate a model checking task to be executed using a well-known tool such as SMV, Spin, or Java Pathfinder. One practical tool design and implementation for aspect validation was suggested in [4], where Bandera is used to generate input for standard model checkers directly from heavily annotated Java code. The annotations that express the specification of the original system are themselves given as aspects. These so-called *specification aspects* include parametric temporal properties, labels, predicates, and functions intended to annotate a system with its desired properties, as preparation of input for Bandera. The parameterization, and the fact that the annotation is kept as a separate module rather than being built into the original system allows the specification aspect to be applied both to the original system, and to one augmented with application aspects. Since annotating a system in preparation for a Bandera verification is a nontrivial task, using specialized notation and requiring human ingenuity, the reuse of the specification aspect is the key to making the approach practical.

Such an approach of aspect validation is possible when the original and the version augmented with aspects are ultimately given in the same notation. In practice, it has been used with AspectJ in the mode that generates source Java code for the system with its aspects, and could also be done when a Java bytecode verifier is available.

In essence, this is an incremental model checking task, if we assume that the properties in the specification of the orginal system were already verified using model checking. The task to be shown in order to verify that the new aspect causes no harm is simply to reprove the specification of the original system, but this time for the augmented system combining the original one with the aspect code bound to various joint points (including dynamic ones). We can and should reuse elements from the model checking of the original system in the model checking of the augmented one.

In particular, any model checking of the original system usually requires abstraction of the statespace to create a smaller model, in order to avoid the state-explosion problem that often prevents a successful verification, even though the model checking itself is algorithmic. Like the specification aspect, these abstractions, used to make the proof feasible in the available space and time, can be reused for the augmented version. If this should prove insufficient because an extensive new statespace is generated, of course new abstractions might be necessary. However, this would violate our goal of fully automatic validation. In case studies we have carried out, the abstractions needed for the original still lead to sufficient reductions in the augmented system, but further research is needed to determine whether this is generally the case.

As opposed to simple regression testing, this is a full verification, and thus will check the desired properties for whatever new paths might be introduced by the weaving of the aspect. However, using aspect validation for regression verification, and thus model-checking the entire augmented system, is clearly less desirable than proving once and for all (by only analyzing the aspect code) that an aspect cannot harm large classes of easily identifiable systems and specification properties.

7. SUMMARY

The goal of full specification and verification of aspect-oriented systems is still important. But even when specifications of aspects are difficult to express for non-functional concerns, and a full verification may be difficult, showing the absence of harm through regression verification is a valuable first step. A significant improvement in code reliability and quality can be obtained at a relatively low cost, especially when a specification of the underlying system is already available. A combined approach of static dataflow analysis, one-time deductive proofs, and aspect validation shows particular promise. Proper language design for aspects, with local variables and parameterization, can help extend the static analysis of only the aspect code to determine harmfulness or its absence, either for classes of properties and for every possible weaving, or reanalyzing only the aspect for each weaving. When analysis of the full augmented system is required, aspect validation is suggested.

This focus on harmful aspects also allows a weakening of obliviousness in a way that maintains extensibility, but does not allow (or at least diagnoses) malicious or inadvertent corruption of the desired properties of the underlying system.

ACKNOWLEDGMENTS

This work was done while the author was visiting at Lancaster University in the PROBE project, and is supported by UK Engineering and Physical Sciences Research Council (EPSRC) Grant GR/S70159/01. Valuable discussions with Awais Rashid are gratefully acknowledged.

8. REFERENCES

[1] Devereux, B., Compositional Reasoning about Aspects using Alternating-time Logic, FOAL 2003.

[2] Filman, R.E., and Friedman, D.P., Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced separation of Concerns, OOPSLA 2000, October 2000.

[3] Filman, R.E., What is AOP, Revisited, Workshop on Advanced Separation of Concerns, 15th ECOOP, June, 2001.

[4] Katz, S., and Sihman, M., Aspect Validation Using Model Checking , Intl. Symposium on Verification in honor of Zohar Manna, Springer-Verlag, LNCS 2772, pp. 389-411. Also, early version in FOAL2003.

[5] Lamport, L., What Good is Temporal Logic?, IFIP 9th World Congress, 1983, pp. 657-668.

[6] Sihman, M. and Katz, S. Superimposition and Aspect-Oriented Programming, The Computer Journal, 46 (5), 2003, pp. 529-541.

[7] Sipma, H. B., A Formal Model for Cross-cutting Modular Transition Systems, FOAL 2003.

[8] eXtreme Programming homepage, http://www.extremeprogramming.org

Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

This paper makes two contributions to a formal understanding of aspect-oriented programming. First, we define `TinyAspect`, a formal model capturing core AOP concepts. Compared to previous formalizations of AOP constructs, `TinyAspect`is extremely small, models aspects at the source level, and is defined using structured operational semantics and syntax-directed typing rules. In combination, these properties make it easy to investigate aspect-oriented language extensions and prove theorems about them.

Second, we propose Open Modules, a module system for `TinyAspect`that guarantees modular reasoning in the presence of aspects. Modular reasoning can be challenging in AOP systems because advice can change the semantics of a module from the outside. Open Modules are "open" in that external aspects can advise functions and pointcuts in their interface, providing significant aspect-oriented expressiveness that is missing in non-AOP systems. In order to guarantee modular reasoning, however, our system places limits on advice: external aspects may not advise function calls internal to a module, except for calls explicitly exposed through pointcuts in the module's interface.

We use a notion of bisimulation to show that Open Modules enforce Reynolds' abstraction theorem, a strong encapsulation property. This theorem guarantees that clients are unaffected by changes to a module, as long as those changes preserve the semantics of the functions and pointcuts in the module's interface.

1. Outline

This paper makes two contributions: the definition of a new formal model for aspect-oriented programming, and a proposed module system for aspects. In order to cleanly separate these contributions, we motivate each in its own section. We begin in Section 2 with the presentation of `TinyAspect`, a minimal core language for aspect-oriented programming.

Section 3 motivates the need for better module systems in aspect-oriented programming, and provides an overview of Open Modules, our proposed design. Section 4 formalizes the Open Modules proposal as an extension to `TinyAspect`. In Section 5 we use a notion of bisimulation to show that Open Modules enforce Reynolds' abstraction theorem, a strong encapsulation property. Section 6 discusses related work, Section 7 outlines future work, and Section 8 concludes.

Names	n	::=	x
Expressions	e	::=	$n \mid \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e \mid e_1\ e_2 \mid \texttt{()}$
Declarations	d	::=	$\bullet$
		|	$\texttt{val}\ x = e\ \ d$
		|	$\texttt{pointcut}\ x = p\ \ d$
		|	$\texttt{around}\ p(x{:}\tau) = e\ \ d$
Pointcuts	p	::=	$n \mid \texttt{call}(n)$
Types	τ, σ	::=	$\texttt{unit} \mid \tau_1 \rightarrow \tau_2 \mid \texttt{pc}(\tau_1 \rightarrow \tau_2)$

Figure 1: `TinyAspect` Source Syntax

2. The TinyAspect Language

We would like to use a formal model of aspect-oriented programming in order to study language extensions like the module system discussed in the next section. While other researchers have used denotational semantics [22], big-step operational semantics [12], and translation systems [14, 20] to study the semantics of aspect-oriented programming, small-step operational semantics have the advantage of providing a simple and direct semantics that is amenable to syntactic proof techniques.

Jagadeesan et al. have proposed an operational semantics for the core of AspectJ, incorporating several different kinds of pointcuts and advice in an object-oriented setting [10]. These features are ideal for modeling AspectJ, but the complexity of the model makes it tedious to prove properties about the system.

Walker et al. propose a much simpler formal model incorporating just the lambda calculus, labeled join points, and advice [21]. However, their system is not intended to model the source-level constructs of langauges like AspectJ directly; instead, it is a foundational calculus into which source-level AOP constructs can be translated. It is considerably more general than existing languages like AspectJ, and so properties that might be true at the source level of a language may not hold in the foundational calculus. Thus, a source-level formal model would be a more effective way to investigate many source-level properties.

We have developed a new functional core language for aspect-oriented programming called `TinyAspect` that is intended to make proofs of source-level properties as straightforward as possible. As the name suggests, `TinyAspect` is tiny, containing only the lambda calculus with units, declara-

tions, pointcuts, and `around` advice. `TinyAspect` directly models AOP constructs similar to those found in AspectJ, making source-level properties easy to specify and prove using small-step operational semantics and standard syntactic techniques. Although we are working in an aspect-oriented, functional setting, our system's design is inspired by that of Featherweight Java [9], which has been successfully used to study a number of object-oriented language features.

Figure 1 shows the syntax of `TinyAspect`. Our syntax is modeled after ML [15], so that `TinyAspect` programs are easy to read and understand. Names in `TinyAspect` are simple identifiers; we will extend this to paths when we add module constructs to the language. Expressions include the monomorphic lambda calculus – names, functions, and function application. To this core, we add a primitive unit expression, so that we have a base case for types. We could add primitive booleans and integers in a completely standard way. Since these constructs are orthogonal to aspects, we omit them.

In most aspect-oriented programming languages, including AspectJ, the pointcut and advice constructs are second-class and declarative. So as to be an accurate source-level model, a `TinyAspect` program is made up of a sequence of declarations. Each declaration defines a scope that includes the following declarations. A declaration is either the empty declaration, or a value binding, a pointcut binding, or advice. The `val` declaration gives a static name to a value so that it may be used or advised in other declarations.

The `pointcut` declaration names a pointcut in the program text. A pointcut of the form `call`(n) refers to any call to the function value defined at declaration n, while a pointcut of the form n is just an alias for a previous pointcut declaration n. A real language would have more pointcut forms; we include only the most basic possible form in order to keep the language minimal.

The `around` declaration names some pointcut p describing calls to some function, binds the variable x to the argument of the function, and specifies that the advice e should be run in place of the original function. Inside the body of the advice e, the special variable `proceed` is bound to the original value of the function, so that e can choose to invoke the original function if desired.

`TinyAspect` types τ include the `unit` type, function types of the form $\tau_1 \rightarrow \tau_2$, and pointcut types representing calls to a function of type $\tau_1 \rightarrow \tau_2$.

2.1 Fibonacci Caching Example

We illustrate the language by writing the Fibonacci function in it, and writing a simple aspect that caches calls to the function to increase performance. While this is not a compelling example of aspects, it is standard in the literature and simple enough for an introduction to the language.

Figure 2 shows the `TinyAspect` code for the Fibonacci function. We assume integers and booleans have been added to illustrate the example.

`TinyAspect` does not have recursion as a primitive in the language, so the `fib` function includes just the base case of the Fibonacci function definition, returning 1.

We use `around` advice on calls to `fib` to handle the recursive cases. The advice is invoked first whenever a client calls `fib`. The advice is invoked first whenever a client calls `fib`. The body of the advice checks to see if the argument is greater than 2; if so, it returns the sum of `fib(x-1)` and `fib(x-2)`. These recursive calls are intercepted by the advice, rather than the original function, allowing recursion to work properly. In the case when the argument is less than 3, the advice invokes `proceed` with the original number `x`. Within the scope of an advice declaration, the special variable `proceed` refers to the advised definition of the function. Thus, the call to `proceed` is forwarded to the original definition of `fib`, which returns 1.

```
val fib = fn x:int => 1
around call(fib) (x:int) =
    if (x > 2)
        then fib(x-1) + fib(x-2)
        else proceed x

(* advice to cache calls to fib *)
val inCache = fn ...
val lookupCache = fn ...
val updateCache = fn ...

pointcut cacheFunction = call(fib)
around cacheFunction(x:int) =
    if (inCache x)
        then lookupCache x
        else let v = proceed x
            in updateCache x v; v
```

Figure 2: The Fibonacci function written in `TinyAspect`, along with an aspect that caches calls to `fib`.

In the lower half of the figure is an aspect that caches calls to `fib`, thereby allowing the normally exponential function to run in linear time. We assume there is a cache data structure and three functions for checking if a result is in the cache for a given value, looking up an argument in the cache, and storing a new argument-result pair in the cache.

So that we can make the caching code more reusable, we declare a `cacheFunction` pointcut that names the function calls to be cached–in this case, all calls to `fib`. Then we declare `around` advice on the `cacheFunction` pointcut which checks to see if the argument `x` is in the cache. If it is, the advice gets the result from the cache and returns it. If the value is not in the cache, the advice calls `proceed` to calculate the result of the call to `fib`, stores the result in the cache, and then returns the result.

In the semantics of `TinyAspect`, the last advice to be declared on a declaration is invoked first. Thus, if a client calls `fib`, the caching advice will be invoked first. If the caching advice calls `proceed`, then the first advice (which recursively defines `fib`) will be invoked. If that advice in turn calls `proceed`, the original function definition will be invoked. However, if the advice makes a recursive call to `fib`, the call will be intercepted by the caching advice. Thus, the cache works exactly as we would expect–it is invoked on all recursive calls to `fib`, and thus it is able to effectively avoid the exponential cost of executing `fib` in the naïve way.

2.2 Operational Semantics

We define the semantics of `TinyAspect` more precisely as a set of small-step reduction rules. These rules translate a series of source-level declarations into the values shown in Figure 3.

Expression-level values include the unit value and func-

Expression values	v	::=	`()` | `fn` $x{:}\tau$ `=>` e | ℓ
Pointcut values	p_v	::=	`call`(ℓ)
Declaration values	d_v	::=	$\bullet$
		|	`val` $x \equiv v\ \ d_v$
		|	`pointcut` $x \equiv p_v\ \ d_v$
Evaluation contexts	C	::=	$\square\ e_2$ | $v_1\ \square$ | `val` $x = \square\ \ d$
		|	`val` $x \equiv v\ \square$
		|	`pointcut` $x \equiv p_v\ \square$

Figure 3: `TinyAspect` Values and Contexts

tions. In `TinyAspect`, advice applies to declarations, not to functions. We therefore need to keep track of declaration usage in the program text, and so a reference to a declaration is represented by a label ℓ. In the operational semantics, below, an auxiliary environment keeps track of the advice that has been applied to each declaration.

A pointcut value can only take one form: calls to a particular declaration ℓ. In our formal system we model execution of declarations by replacing source-level declarations with "declaration values," which we distinguish by using the $\equiv$ symbol for binding.

Figure 3 also shows the contexts in which reduction may occur. Reduction proceeds first on the left-hand side of an application, then on the right-hand side. Reduction occurs within a value declaration before proceeding to the following declarations. Pointcut declarations are atomic, and so they only define an evaluation context for the declarations that follow.

Figure 4 describes the operational semantics of `TinyAspect`. A machine state is a pair (η, e) of an advice environment η (mapping labels to values) and an expression e. Advice environments are similar to stores, but are used to keep track of a mapping from declaration labels to declaration values, and are modified by advice declarations. We use the $\eta[\ell]$ notation in order to look up the value of a label in η, and we denote the functional update of an environment as $\eta' = [\ell{\mapsto}v]\ \eta$. The reduction judgment is of the form $(\eta, e) \mapsto (\eta', e')$, read, "In advice environment η, expression e reduces to expression e' with a new advice environment η'."

The rule for function application is standard, replacing the application with the body of the function and substituting the argument value v for the formal x. We normally treat labels ℓ as values, because we want to avoid "looking them up" before they are advised. However, when we are in a position to invoke the function represented by a label, we use the rule *r-lookup* to look up the label's value in the current environment.

The next three rules reduce declarations to "declaration values." The `val` declaration binds the value to a fresh label and adds the binding to the current environment. It also substitutes the label for the variable x in the subsequent declaration(s) d. We leave the binding in the reduced expression both to make type preservation easier to prove, and also to make it easy to extend `TinyAspect` with a module system which will need to retain the bindings. The `pointcut` declaration simply substitutes the pointcut value for the variable x in subsequent declaration(s).

The `around` declaration looks up the advised declaration

$$\frac{}{(\eta,\ (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ e)\ v) \mapsto (\eta,\ \{v/x\}e)}\ \textit{r-app}$$

$$\frac{\eta[\ell] = v_1}{(\eta,\ \ell\ v_2) \mapsto (\eta,\ v_1\ v_2)}\ \textit{r-lookup}$$

$$\frac{\ell \notin \mathit{domain}(\eta) \quad \eta' = [\ell{\mapsto}v]\ \eta}{(\eta,\ \texttt{val}\ x = v\ \ d) \mapsto (\eta',\ \texttt{val}\ x \equiv \ell\ \ \{\ell/x\}d)}\ \textit{r-val}$$

$$\frac{}{\begin{array}{c}(\eta,\ \texttt{pointcut}\ x = \texttt{call}(\ell)\ \ d) \mapsto \\ (\eta,\ \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ \{\texttt{call}(\ell)/x\}d)\end{array}}\ \textit{r-pointcut}$$

$$\frac{\begin{array}{c}v' = (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ \{\ell'/\texttt{proceed}\}e) \\ \ell' \notin \mathit{domain}(\eta) \quad \eta' = [\ell{\mapsto}v', \ell'{\mapsto}\eta[\ell]]\ \eta\end{array}}{(\eta,\ \texttt{around}\ \texttt{call}(\ell)(x{:}\tau) = e\ \ d) \mapsto (\eta',\ d)}\ \textit{r-around}$$

$$\frac{(\eta,\ e) \mapsto (\eta',\ e')}{(\eta,\ C[e]) \mapsto \eta',\ C[e'])}\ \textit{r-context}$$

Figure 4: `TinyAspect` Operational Semantics

ℓ in the current environment. It places the old value for the binding in a fresh label ℓ', and then re-binds the original ℓ to the body of the advice. Inside the advice body, any references to the special variable `proceed` are replaced with ℓ', which refers to the original value of the advised declaration. Thus, all references to the original declaration will now be redirected to the advice, while the advice can still invoke the original function by calling `proceed`.

The last rule shows that reduction can proceed under any context as defined in Figure 3.

2.3 Typechecking

Figure 5 describes the typechecking rules for `TinyAspect`. Our typing judgment for expressions is of the form $\Gamma; \Sigma \vdash e : \tau$, read, "In variable context Γ and declaration context Σ expression e has type τ." Here Γ maps variable names to types, while Σ maps labels to types (similar to a store type).

The rules for expressions are standard. We look up the types for variables and labels in Γ and Σ, respectively. Other standard rules give types to the `()` expression, as well as to functions and applications.

The interesting rules are those for declarations. We give declaration signatures β to declarations, where β is a sequence of variable to type bindings. The base case of an empty declaration has an empty signature. For `val` bindings, we ensure that the expression is well-typed at some type τ, and then typecheck subsequent declarations assuming that the bound variable has that type. Pointcuts are similar, but the rule ensures that the expression p is well-typed as a pointcut denoting calls to a function of type $\tau_1 \to \tau_2$. The around advice rule checks that the declared type of x matches the argument type in the pointcut, and checks that the body is well-typed assuming proper types for the variables x and `proceed`.

Finally, the judgment $\Sigma \vdash \eta$ states that η is a well-formed environment with typing Σ whenever all the values in η have the types given in Σ. This judgment is analogous to store

$$\frac{x{:}\tau \in \Gamma}{\Gamma;\Sigma \vdash x : \tau}\ \textit{t-var}$$

$$\frac{\Gamma;\Sigma \vdash n : \tau_1 \to \tau_2}{\Gamma;\Sigma \vdash \texttt{call}(n) : \texttt{pc}(\tau_1 \to \tau_2)}\ \textit{t-pctype}$$

$$\frac{\ell{:}\tau \in \Sigma}{\Gamma;\Sigma \vdash \ell : \tau}\ \textit{t-label}$$

$$\frac{}{\Gamma;\Sigma \vdash \texttt{()} : \texttt{unit}}\ \textit{t-unit}$$

$$\frac{\Gamma, x{:}\tau_1;\Sigma \vdash e : \tau_2}{\Gamma;\Sigma \vdash \texttt{fn}\ x{:}\tau_1\ \texttt{=>}\ e : \tau_1 \to \tau_2}\ \textit{t-fn}$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma;\Sigma \vdash e_2 : \tau_2}{\Gamma;\Sigma \vdash e_1\ e_2 : \tau_1}\ \textit{t-app}$$

$$\frac{}{\Gamma;\Sigma \vdash \bullet : \bullet}\ \textit{t-empty}$$

$$\frac{\Gamma;\Sigma \vdash e : \tau \quad \Gamma, x{:}\tau;\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \texttt{val}\ x = e\ \ d : (x{:}\tau, \beta)}\ \textit{t-val}$$

$$\frac{\Gamma;\Sigma \vdash p : \texttt{pc}(\tau_1 \to \tau_2) \quad \Gamma, x{:}\texttt{pc}(\tau_1 \to \tau_2);\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \texttt{pointcut}\ x = p\ \ d : (x{:}\texttt{pc}(\tau_1 \to \tau_2), \beta)}\ \textit{t-pc}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash p : \texttt{pc}(\tau_1 \to \tau_2) \quad \Gamma;\Sigma \vdash d : \beta \\ \Gamma, x{:}\tau_1, \texttt{proceed}{:}\tau_1 \to \tau_2;\Sigma \vdash e : \tau_2\end{array}}{\Gamma;\Sigma \vdash \texttt{around}\ p(x{:}\tau_1) = e\ \ d : \beta}\ \textit{t-around}$$

$$\frac{\forall \ell.(\Sigma[\ell] = \tau \wedge \eta[\ell] = v \implies \bullet;\Sigma \vdash v : \tau)}{\Sigma \vdash \eta}\ \textit{t-env}$$

Figure 5: `TinyAspect` Typechecking

typings in languages with references.

2.4 Type Soundness

We now state progress and preservation theorems for `TinyAspect`. The theorems quantify over both expressions and declarations using the metavariable E, and quantify over types and declaration signatures using the metavariable T. The progress property states that if an expression is well-typed, then either it is already a value or it will take a step to some new expression.

Theorem 1 (Progress)
If $\bullet;\Sigma \vdash E : T$ and $\Sigma \vdash \eta$, then either E is a value or there exists η' such that $(\eta, E) \mapsto (\eta', E')$.

Proof: By induction on the derivation of $\bullet;\Sigma \vdash E : T$. ■

The type preservation property states that if an expression is well-typed and it reduces to another expression in a new environment, then the new expression and environment are also well-typed.

Theorem 2 (Type Preservation)
If $\bullet;\Sigma \vdash E : T$, $\Sigma \vdash \eta$, and $(\eta, E) \mapsto (\eta', E')$, then there exists some $\Sigma' \supseteq \Sigma$ such that $\bullet;\Sigma' \vdash E' : T$ and $\Sigma' \vdash \eta'$.

Proof: By induction on the derivation of $(\eta, E) \mapsto (\eta', E')$. The proof relies on a standard substitution and weakening lemmas. ■

Together, progress and type preservation imply type soundness. Soundness means that there is no way that a well-typed `TinyAspect` program can get stuck or "go wrong" because it gets into some bad state.

3. Open Modules

In this section, we explore one possible way to define a module system for aspects. The following section models our proposed module system in `TinyAspect`, providing an initial evaluation of the core language design, and gaining insight into the potential benefits of our module system.

3.1 Motivation

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules in order to hide information that is likely to change [17]. Thus if change is anticipated with reasonable accuracy, the system can be evolved with local rather than global system modifications, easing many software maintenance tasks. Furthermore, the correctness of each module can be verified in isolation from other modules, allowing developers to work independently on different sub-problems.

Unfortunately, developers do not always respect the information hiding boundaries of modules–it is often tempting to reach across the boundary for temporary convenience, while causing more serious long-term evolution problems. Thus, encapsulation mechanisms such as Java's packages and public/private data members were developed to give programmers compiler support for enforcing information hiding boundaries.

The central insight behind aspect-oriented programming is that conventional modularity and encapsulation mechanisms are not flexible enough to capture many concerns that are likely to change. These concerns cannot be effectively hidden behind information-hiding boundaries, because they are scattered in many placed throughout the system and tangled together with unrelated code. Aspect-oriented programming systems provide mechanisms for modularizing a more diverse set of concerns. However, few aspect-oriented programming projects have addressed the problem of providing an encapsulation facility for aspect-oriented programming.

3.2 Existing Encapsulation Approaches

The most widely-used AOP system, AspectJ, leaves Java's existing encapsulation mechanisms largely unchanged [11]. AspectJ provides new programming mechanisms that capture concerns which crosscut Java's class and package structure. Because these mechanisms can reach across encapsulation boundaries, AspectJ does not enforce information hiding between aspects and other code.

For example, Figure 6 shows how an aspect can depend on the implementation details of another module. The figure shows two different `Shape` subclasses, one representing points and another representing rectangles. Both classes have a method `moveBy`, which moves the rectangles on the screen. An assurance aspect checks certain invariants of

```
package shape;

public class Point extends Shape {
  public void moveBy(int dx, int dy) {
    x += dx; y += dy;
  ...
}

public class Rectangle extends Shape {
  public void moveBy(int dx, int dy) {
    p1x += dx; p1y += dy;
    p2x += dx; p2y += dy;
  ...
}


package assure;

aspect AssureShapeInvariants {
  pointcut moves():
      call(void shape.Shape+.moveBy(..));

  after(): moves() {
    scene.checkInvariants();
  }
}
```

Figure 6: In this AspectJ code, the correctness of the shape invariants aspect depends on the implementation of the shapes. If the implementation is changed so that `Rectangle` uses `Point` to hold its coordinates, then the invariants will be checked in the middle of a `moveBy` operation, possibly leading to a spurious invariant failure.

the scene every time a shape moves. The aspect is triggered by a pointcut made up of all calls to the `moveBy` function in shapes. We assume the assurance aspect is checking application-level invariants, rather than invariants specific to the `shape` package, and therefore it is defined in a package of its own.

Unfortunately, this aspect is tightly coupled to the implementation details of the `shape` package, and will break if these implementation details are changed. For example, consider what happens if the rectangle is modified to store its coordinates as a pair of points, rather than two pairs of integer values. The body of `Rectangle.moveBy` would be changed to read:

```
p1.moveBy(dx, dy);
p2.moveBy(dx, dy);
```

Now the `moves` pointcut will be invoked not only when the `Rectangle` moves, but also when its constituent points move. Thus, the scene invariants will be checked in the middle of the rectangle's `moveBy` operation. Since the scene invariants need not be true in the intermediate state of motion, this additional checking could lead to spurious invariant failures.

The aspect in Figure 6 violates the information hiding boundary of the `shape` package by placing advice on method calls within the package. This means that the implementor of `shape` cannot freely switch between semantically equivalent implementations of `Rectangle`, because the external aspect may break if the implementation is changed. Because the aspect violates information hiding, evolving the `shape` package becomes more difficult and error prone.

AspectJ is not the only system in which aspects can violate information hiding boundaries. Other aspect-oriented programming systems that support method interception, such as Hyper/J [19] and ComposeJ [23], share the issue. Even recent proposals describing module systems for AOP allow these kinds of violations [13, 6].

Clearly the programmer of the assurance aspect could have written the aspect to be more robust to this kind of change. However, the whole point of an encapsulation system is to protect the programmer from violating information hiding boundaries. In the rest of this paper, we explore a proposed module system that is able to enforce information hiding, while preserving much of the expressiveness of existing aspect-oriented programming systems.

3.3 Overview

We propose Open Modules, a new module system for aspect-oriented programs that is intended to be *open* to aspect-oriented extension but *modular* in that the implementation details of a module are hidden. The goals of openness and modularity are in tension, and so we try to achieve a compromise between them.

The principle behind the design of Open Modules is that interfaces should mediate the interaction between the implementation of a module and its clients, even in the presence of aspects. Our system can capture crosscutting concerns in much the same way as previous aspect-oriented programming systems; the only difference is that some pointcuts may have to be moved from the aspect code to the interface of the module being advised.

For example, the assurance aspect in Figure 6 would be prohibited by our system as written, because the aspect's pointcut potentially includes calls that are within the private implementation of the `shape` package. However, the aspect could be re-written in one of two ways to be compatible with Open Modules.

In the first solution, the pointcut in the aspect would additionally specify that it captures only calls from *outside* the `shape` package:

```
pointcut moves():
    call(void shape.Shape+.moveBy(..))
    && !within(shape.*);
```

This solution fits with Open Modules because it advises only incoming calls to the interface of the package; the aspect is decoupled from the implementation, permitting implementation changes like the one discussed in the previous subsection.

In the second solution, the pointcut in the aspect would be moved to the `shape` module, and then referenced by the aspect:

```
after(): shape.Shape.moves() { ... }
```

In this case, the pointcut becomes part of the interface of the `shape` package, again decoupling the aspect from the package's implementation. If that implementation changes, the maintainer of the module has the responsibility to maintain the semantics of the pointcut so that external aspects are unaffected by the change.

This second solution, called *pointcut interfaces*, was originally proposed by Gudmundson and Kiczales as an engineering technique that can ease software evolution by decoupling an aspect from the code that it advises [8]. It is also related to the Demeter project's use of *traversal strategies* to isolate an aspect from the code that it advises [16].

We now provide a more technical definition for Open Modules, which can be used to distinguish our contribution from previous work:

Definition [Open Modules]: *A module system that:*

- *allows external aspects to advise external calls to functions in the interface of a module*
- *allows external aspects to advise pointcuts in the interface of a module*
- *does not allow external aspects to advise calls from within a module to other functions within the module (including exported functions).*

3.4 Expressiveness

Like the Gudmundson and Kiczales proposal on which they are based [8], Open Modules sacrifice some amount of *obliviousness* [7] in order to support better information hiding. Base code is not completely oblivious to aspects, because the author of a module must expose relevant internal events in pointcuts so that aspects can advise them[1]. However, our design still preserves important cases of obliviousness:

- While a module can expose interesting implementation events in pointcuts, it is oblivious to which aspects might be interested in those events.
- Pointcuts in the interface of a module can be defined *non-invasively* with respect to the rest of the module's implementation, using the same pointcut operations available in other AOP languages.
- A module is completely oblivious to aspects that only advise external calls to its interface.

A possible concern is that the strategy of adding a pointcut to the interface of a base module may be impossible if the source code for that module cannot be changed. In this case, the modularity benefits of Open Modules can be achieved with environmental support for associating an external pointcut with the base module. If the base module is updated, the maintainer of the pointcut is responsible for rechecking the pointcut to ensure that its semantics have not been invalidated by the changes to the base module.

Experiment. In a companion paper, we performed a micro-experiment applying the ideas of Open Modules to SpaceWar, a small demonstration application distributed with AspectJ. The experiment was far too small to provide definitive results. However, we found that Open Modules support nearly all of the aspects in this program with no changes or only minor changes to the code [2].

The only concern our system could not handle was an extremely invasive debugging aspect. Debugging is an inherently non-modular activity, so we view it as a positive sign that our module system does not support it. In a practical system, debugging can be supported either through external tools, or through a compiler flag that makes an exception to the encapsulation rules during debugging activity.

[1]We note that many in the AOP community feel "obliviousness" is too strong a term, preferring a notion of "non-invasiveness" that is compatible with our proposal. See for example posts to the aosd-discuss mailing list by Dean Wempler and Gregor Kiczales in August 2003, available at aosd.net.

Names	n	::=	$\ldots \mid m.x$
Declarations	d	::=	$\ldots \mid \texttt{structure}\ x = m\ d$
Modules	m	::=	n
		$\mid$	$\texttt{struct}\ d\ \texttt{end}$
		$\mid$	$m\ \texttt{:>}\ \sigma$
		$\mid$	$\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m$
		$\mid$	$m_1\ m_2$
Types	τ, σ	::=	$\ldots \mid \texttt{sig}\ \beta\ \texttt{end}$
Decl. values	d_v	::=	$\ldots \mid \texttt{structure}\ x = \square\ d$
Module values	m_v	::=	$\texttt{struct}\ d_v\ \texttt{end}$
		$\mid$	$\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m$
Contexts	C	::=	$\ldots \mid \texttt{structure}\ x = \square\ d$
		$\mid$	$\texttt{structure}\ x \equiv m_v\ \square$
		$\mid$	$\texttt{struct}\ \square\ \texttt{end} \mid \square\ \texttt{:>}\ \sigma$
		$\mid$	$\square\ m_2 \mid m_v\ \square$

Figure 7: Module System Syntax, Values, and Contexts

Comparison to non-AOP techniques. One way to evaluate the expressiveness of Open Modules is to compare them to non-AOP alternatives. One alternative is using wrappers instead of aspects to intercept the incoming calls to a module, and using callbacks instead of pointcuts in the module's interface. The aspect-oriented nature of Open Modules provides several advantages over the wrapper and callback solution:

- Open Modules are compatible with the *quantification* [7] constructs of languages like AspectJ, so that many functions can be advised with a single declaration. Implementing similar functionality with conventional wrappers–which do not support quantification–is far more tedious because a wrapper must be explicitly applied to each function.
- In Open Modules, a single, locally-defined aspect can implement a crosscutting concern by non-locally extending the interface of a number of modules. Wrappers cannot capture these concerns in a modular way, because each target module must be individually wrapped.
- Callbacks are invasive with respect to the implementation of a module because the implementation must explicitly invoke the callback at the appropriate points. In contrast, pointcut interfaces are non-invasive in that the pointcut is defined orthogonally to the rest of the module's implementation, thus providing better support for separation of concerns.

```
structure Cache =
  functor(X : sig f : pc(int->int) end) =>
    struct
        around X.f(x:int) = ...
            (* same definition as before *)
    end

structure Math = struct
    val fib = fn x:int => 1
    around call(fib) (x:int) =
        if (x > 2)
            then fib(x-1) + fib(x-2)
            else proceed x

    structure cacheFib =
        Cache (struct
                   pointcut f = call(fib)
               end)
end :> sig
  fib : int->int
end
```

Figure 8: Fibonacci with Open Modules

These advantages illustrate how the quantification and non-invasive extension provided by Open Modules distinguish our proposal from solutions that do not use aspects [7].

4. Formalization of Open Modules

We now extend `TinyAspect` to model Open Modules. Our module system is modeled closely after that of ML, providing a familiar concrete syntax and benefiting from the design of an already advanced module system.

Figure 7 shows the new syntax for modules. Names include both simple variables x and qualified names $m.x$, where m is a module expression. Declarations can include structure bindings, and types are extended with module signatures of the form `sig` β `end`, where β is the list of variable to type bindings in the module signature.

First-order module expressions include a name, a `struct` with a list of declarations, and an expression m `:>` σ that seals a module with a signature, hiding elements not listed in the signature. The expression `functor`$(x{:}\sigma)$ `=>` m describes a functor that takes a module x with signature σ as an argument, and returns the module m which may depend on x. Functor application is written like function application, using the form $m_1\ m_2$.

Our module system does not include abstract types, and so the abstraction property we enforce is one of implementation independence, not representation independence. The underlying problem is the same in both cases: external aspects should not be able to observe the internal behavior of module functions. Thus, we conjecture that our solution to the implementation independence problem will also enforce representation independence once abstract types are added in standard ways [15].

4.1 Fibonacci Revisited

Figure 8 shows how a more reusable caching aspect could be defined using functors. The `Cache` functor accepts a module that has a single element `f` that is a pointcut of calls to some function with signature `int->int`. The `around` advice then advises the pointcut from the argument module `X`.

The `fib` function is now encapsulated inside the `Math` module. The module implements caching by instantiating the `Cache` module with a structure that binds the pointcut `f` to calls to `fib`. Finally, the `Math` module is sealed with a signature that exposes only the `fib` function to clients.

```
structure shape = struct
    val createShape = fn ...
    val moveBy = fn ...
    val animate = fn ...
    ...
    pointcut moves = call(moveBy)
end :> sig
    createShape : Description -> Shape
    moveBy      : (Shape,Location) -> unit
    animate     : (Shape,Path) -> unit
    ...
    moves       : pc((Shape,Location)->unit)
end
```

Figure 9: A shape library that exposes a position change pointcut

4.2 Sealing

Our module sealing operation has an effect both at the type system level and at the operational level. At the type level, it hides all members of a module that are not in the signature σ–in this respect, it is similar to sealing in ML's module system. However, sealing also has an operational effect, hiding internal calls within the module so that clients cannot advise them unless the module explicitly exports the corresponding pointcut.

For example, in Figure 8, clients of the `Math` module would not be able to tell whether or not caching had been applied, even if they placed advice on `Math.fib`. Because `Math` has been sealed, external advice to `Math.fib` would only be invoked on external calls to the function, not on internal, recursive calls. This ensures that clients cannot be affected if the implementation of the module is changed, for example, by adding or removing caching.

The strategy used to protect information hiding in our formal system is slightly different from the informal strategy presented in Section 3. There we were assuming the semantics of AspectJ, and so in order to avoid capturing internal calls we had to explicitly say `!within(shape.*)` in the pointcut. In the formal system, we provide a cleaner solution, where once a module is sealed, externally defined pointcuts *automatically* include the limitation to external calls.

4.3 Exposing Semantic Events with Pointcuts

Figure 9 shows how the shape example described above could be modeled in `TinyAspect`. Clients of the shape library cannot advise internal functions, because the module is sealed. To allow clients to observe internal but semantically important events like the motion of animated shapes, the module exposes these events in its signature as the `moves` pointcut. Clients can advise this pointcut without depending on the internals of the shape module. If the module's implementation is later changed, the `moves` pointcut must

$$\frac{bind\ x \equiv v \in d_v}{(\eta,\ \texttt{struct}\ d_v\ \texttt{end}.x) \mapsto (\eta,\ v)}\ \textit{r-path}$$

$$\frac{}{(\eta,\ \texttt{structure}\ x = m_v\ \ d) \mapsto (\eta,\ \texttt{structure}\ x \equiv m_v\ \ \{m_v/x\}d)}\ \textit{r-structure}$$

$$\frac{}{(\eta,\ (\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m_1)\ m_2) \mapsto (\eta,\ \{m_2/x\}m_1)}\ \textit{r-fapp}$$

$$\frac{seal(\eta, d_v, \beta) = (\eta', d_{seal})}{(\eta,\ \texttt{struct}\ d_v\ \texttt{end}\ \texttt{:>}\ \texttt{sig}\ \beta\ \texttt{end}) \mapsto (\eta',\ \texttt{struct}\ d_{seal}\ \texttt{end})}\ \textit{r-seal}$$

$$\frac{}{seal(\eta, \bullet, \bullet) = (\eta, \bullet)}\ \textit{s-empty}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{seal(\eta, bind\ x \equiv v\ \ d, \beta) = (\eta', d')}\ \textit{s-omit}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d') \quad \eta'' = [\ell \mapsto v]\,\eta' \quad \ell \notin domain(\eta')}{seal(\eta, \texttt{val}\ x \equiv v\ \ d, (x{:}\tau, \beta)) = (\eta'', \texttt{val}\ x \equiv \ell\ \ d')}\ \textit{s-v}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{seal(\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d, (x{:}\texttt{pc}(\tau), \beta)) = (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d')}\ \textit{s-p}$$

$$\frac{seal(\eta, d_s, \beta_s) = (\eta', d_s') \quad seal(\eta', d, \beta) = (\eta'', d')}{seal(\eta, \texttt{structure}\ x \equiv \texttt{struct}\ d_s\ \texttt{end}\ \ d, (x{:}\texttt{sig}\ \beta_s\ \texttt{end}, \beta)) = (\eta'', \texttt{structure}\ x \equiv \texttt{struct}\ d_s'\ \texttt{end}\ \ d')}\ \textit{s-s}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{seal(\eta, \texttt{structure}\ x \equiv \texttt{functor}(y{:}\sigma_y)\ \texttt{=>}\ m\ \ d, (x{:}\sigma, \beta)) = (\eta', \texttt{structure}\ x \equiv \texttt{functor}(y{:}\sigma_y)\ \texttt{=>}\ m\ \ d')}\ \textit{s-f}$$

Figure 10: Module System Operational Semantics

also be changed to ensure that client aspects are not affected.

Thus, sealing enforces the abstraction boundary between a module and its clients, allowing programmers to reason about and change them independently. However, our system still allows a module to export semantically important internal events, allowing clients to extend or observe the module's behavior in a principled way.

4.4 Open Modules Operational Semantics

Figure 10 shows the operational semantics for Open Modules. In the rules, module values m_v mean either a struct with declaration values d_v or a functor. The path lookup rule finds the selected binding within the declarations of the module. We assume that bound names are distinct in this rule; it is easy to ensure this by renaming variables appropriately. Because modules cannot be advised, there is no need to create labels for structure declarations; we can just substitute the structure value for the variable in subsequent declarations. The rule for functor application also uses substitution.

The rule for sealing uses an auxiliary judgment, *seal*, to generate a fresh set of labels for the bindings exposed in the signature. This fresh set of labels insures that clients can affect external calls to module functions by advising the new labels, but cannot advise calls that are internal to the sealed module.

$$\frac{\Gamma; \Sigma \vdash m : \texttt{sig}\ \beta\ \texttt{end} \quad x{:}\tau \in \beta}{\Gamma; \Sigma \vdash m.x : \tau}\ \textit{t-name}$$

$$\frac{\Gamma; \Sigma \vdash m : \sigma \quad \Gamma, x{:}\sigma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{structure}\ x = m\ \ d : (x{:}\sigma, \beta)}\ \textit{t-structure}$$

$$\frac{\Gamma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{struct}\ d\ \texttt{end} : \texttt{sig}\ \beta\ \texttt{end}}\ \textit{t-struct}$$

$$\frac{\Gamma; \Sigma \vdash m : \sigma_m \quad \sigma_m\ \texttt{<:}\ \sigma}{\Gamma; \Sigma \vdash m\ \texttt{:>}\ \sigma : \sigma}\ \textit{t-seal}$$

$$\frac{\Gamma, x{:}\sigma_1; \Sigma \vdash m : \sigma_2}{\Gamma; \Sigma \vdash \texttt{functor}(x{:}\sigma_1)\ \texttt{=>}\ m : \sigma_1 \to \sigma_2}\ \textit{t-functor}$$

$$\frac{\Gamma; \Sigma \vdash m_1 : \sigma_1 \to \sigma \quad \Gamma; \Sigma \vdash m_2 : \sigma_2 \quad \sigma_2\ \texttt{<:}\ \sigma_1}{\Gamma; \Sigma \vdash m_1\ m_2 : \sigma}\ \textit{t-fapp}$$

Figure 11: Open Modules Typechecking

At the bottom of the diagram are the rules defining the sealing operation. The operation accepts an old environment η, a list of declarations d, and the sealing declaration signature β. The operation computes a new environment η' and new list of declarations d'. The rules are structured according to the first declaration in the list; each rule handles the first declaration and appeals recursively to the definition of sealing to handle the remaining declarations.

An empty list of declarations can be sealed with the empty signature, resulting in another empty list of declarations and an unchanged environment η. The second rule allows a declaration $bind\ x \equiv v$ (where *bind* represents one of `val`, `pointcut`, or `struct`) to be omitted from the signature, so that clients cannot see it at all. The rule for sealing a value declaration generates a fresh label ℓ, maps that to the old value of the variable binding in η, and returns a declaration mapping the variable to ℓ. Client advice to the new label ℓ will affect only external calls, since internal references still refer to the old label which clients cannot change. The rule for pointcuts passes the pointcut value through to clients unchanged, allowing clients to advise the label referred to in the pointcut. Finally, the rules for structure declarations recursively seal any internal struct declarations, but leave functors unchanged.

4.5 Typechecking

The typechecking rules, shown in Figure 11, are largely standard. Qualified names are typed based on the binding in the signature of the module m. Structure bindings are given a declaration signature based on the signature σ of the bound module. The rule for `struct` simply puts a `sig` wrapper around the declaration signature. The rules for sealing and functor application allow a module to be passed into a context where a supertype of its signature is expected.

Figure 12 shows the definition of signature subtyping. Subtyping is reflexive and transitive. Subtype signatures may have additional bindings, and the signatures of constituent bindings are covariant. Finally, the subtyping rule

$$\frac{}{\sigma <: \sigma} \textit{ sub-reflex}$$

$$\frac{\sigma <: \sigma' \quad \sigma' <: \sigma''}{\sigma <: \sigma''} \textit{ sub-trans}$$

$$\frac{\beta <: \beta'}{\texttt{sig}\ \beta\ \texttt{end} <: \texttt{sig}\ \beta'\ \texttt{end}} \textit{ sub-sig}$$

$$\frac{\beta <: \beta'}{x : \tau, \beta\ <:\ \beta'} \textit{ sub-omit}$$

$$\frac{\beta <: \beta' \quad \tau <: \tau'}{x : \tau, \beta\ <:\ x : \tau', \beta'} \textit{ sub-decl}$$

$$\frac{\sigma_1' <: \sigma_1 \quad \sigma_2 <: \sigma_2'}{\sigma_1 \to \sigma_2\ <:\ \sigma_1' \to \sigma_2'} \textit{ sub-contra}$$

Figure 12: Signature Subtyping

for functor types is contravariant.

4.6 Type Soundness

When extended with Open Modules, `TinyAspect` enjoys the same type soundness property that the base system has. The theorems and proofs are similar, and so we omit them.

5. Abstraction

The example programs in Section 3 are helpful for understanding the benefits of `TinyAspect`'s module system at an intuitive level. However, we would like to be able to point to a concrete property that enables separate reasoning about the clients and implementation of a module.

Reynolds' *abstraction* property [18] fits these requirements in a natural way. Intuitively, the abstraction property states that if two module implementations are semantically equivalent, no client can tell the difference between the two. This property has two important benefits for software engineering. First of all, it enables reasoning about the properties of a module in isolation. For example, if one implementation of a module is known to be correct, we can prove that a second implementation is correct by showing that it is semantically equivalent to the first implementation. Second, the abstraction property ensures that the implementation of a module can be changed to a semantically equivalent one without affecting clients. Thus, the abstraction property helps programmers to more effectively hide information that is likely to change, as suggested in Parnas' classic paper [17].

In `TinyAspect`, we can state the abstraction property as follows. If two modules m and m' are observationally equivalent and have module signature σ, then for all client declarations d that are well-typed assuming that some variable x has type σ, the client behaves identically when executed with either module.

Intuitively, two modules are observationally equivalent if all of the bound functions and values in the module are equivalent. Two functions are equivalent if they always produce equivalent results given equivalent arguments, *even if a client advises other functions exported by the module*. This illustrates the importance of using sealing to limit the scope of client advice. If two modules are sealed, then they can be proved equivalent assuming that clients can only advise the exported pointcuts. In this sense, module sealing enables separate reasoning that would be impossible otherwise.

$$\frac{\Lambda \vdash (\eta, V) \simeq (\eta', V') : T}{\Lambda \vdash (\eta, V) \cong (\eta', V') : T}$$

$$\frac{\begin{array}{c}(\eta_1, E_1) \overset{\Lambda}{\mapsto}{}^{*} (\eta_1', E_1') \qquad (\eta_2, E_2) \overset{\Lambda}{\mapsto}{}^{*} (\eta_2', E_2') \\ \Lambda' \vdash (\eta_1', E_1') \cong (\eta_2', E_2') : T \\ (\Lambda' - \Lambda) \cap \mathit{domain}(\eta_1 \cup \eta_2) = \emptyset\end{array}}{\Lambda \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{\Lambda \vdash (\eta_1, C_1[\eta_1[\ell]\ v_1]) \cong (\eta_2, C_2[\eta_2[\ell]\ v_2]) : T}{\Lambda \vdash (\eta_1, C_1[\ell\ v_1]) \cong (\eta_2, C_2[\ell\ v_2]) : T}$$

$$\frac{(\eta, E)\ \mathit{and}\ (\eta', E')\ \mathit{diverge\ following\ the\ rules\ above}}{\Lambda \vdash (\eta, E) \cong (\eta', E') : T}$$

Figure 14: `TinyAspect` Observational Equivalence for Expressions

5.1 Formalizing Abstraction

We can define abstraction formally using judgments for observational equivalence of values, written $\Lambda \vdash (\eta, V) \simeq (\eta', V') : T$ and read, "In the context of a set of visible labels Λ, value V in environment η is observationally equivalent to value V' in environment η' at type T. A similar judgment of the form $\Lambda \vdash (\eta, E) \cong (\eta', E') : T$ is used for observationally equivalent expressions. The judgments depend on the set of labels Λ that are visible and thus capable of being advised; in order for two values to be observationally equivalent, they must use these labels in the same way.

The main rules for observational equivalence of values are defined in Figure 13. Most of the rules are straightforward–for example, there is only one unit value, so all values of type unit are equivalent.

The most interesting rule is the one for function values. Two function values are equivalent if for any observationally equivalent argument values v_1 and v_2, they produce equivalent results. Note that the rule for observational equivalence for function values includes both syntactic functions and labels that denote functions. A similar rule is used for observational equivalence of functors.

Two `val` declarations are equivalent if they bind the same variable to the same label (since labels are generated fresh for each declaration we can always choose them to be equal when we are proving equivalence), and the label is equivalent in the two environments η and η'. Since the label exposed by the `val` declaration is visible, it must be in Λ. Pointcut and structure declarations just check the equality of their components. All three declaration forms ensure that subsequent declarations are also equivalent; we assume that the empty declaration $\bullet$ is equivalent to itself. Finally, two first-order modules are equivalent if the declarations inside them are also equivalent.

Figure 14 shows the rules for observational equivalence of expressions. Two expressions are equivalent if they are

$$\Lambda \vdash (\eta_1, v) \simeq (\eta_2, v) : \texttt{unit}$$

$$\Lambda \vdash (\eta_1, v_1) \simeq (\eta_2, v_2) : \tau' \to \tau \quad \text{iff} \quad \begin{array}{l}\text{for all } v_1', v_2' \\ \text{such that } \Lambda \vdash (\eta_1, v_1') \simeq (\eta_2, v_2') : \tau' \\ \text{we have } \Lambda \vdash (\eta_1, v_1\ v_1') \cong (\eta_2, v_2\ v_2') : \tau\end{array}$$

$$\Lambda \vdash (\eta, \texttt{val}\ x \equiv \ell\ \ d_v) \simeq (\eta', \texttt{val}\ x \equiv \ell\ \ d_v') : (x{:}\tau, \beta) \quad \text{iff} \quad \begin{array}{l}\ell \in \Lambda, \Lambda \vdash (\eta, \ell) \simeq (\eta', \ell) : \tau, \\ \text{and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta\end{array}$$

$$\begin{array}{c}\Lambda \vdash (\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v) \simeq \\ (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v') : (x{:}\texttt{pc}(\tau), \beta)\end{array} \quad \text{iff} \quad \ell \in \Lambda \text{ and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta$$

$$\begin{array}{c}\Lambda \vdash (\eta, \texttt{structure}\ x \equiv m_v\ \ d_v) \simeq \\ (\eta', \texttt{structure}\ x \equiv m_v'\ \ d_v') : (x{:}\sigma, \beta)\end{array} \quad \text{iff} \quad \begin{array}{l}\Lambda \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma, \\ \text{and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta\end{array}$$

$$\Lambda \vdash (\eta, \texttt{struct}\ d_v\ \texttt{end}) \simeq (\eta', \texttt{struct}\ d_v'\ \texttt{end}) : \texttt{sig}\ \beta\ \texttt{end} \quad \text{iff} \quad \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta$$

$$\Lambda \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma_1 \to \sigma_2 \quad \text{iff} \quad \begin{array}{l}\text{for all } m_v^1, m_v^2 \text{ such that } \Lambda \vdash (\eta, m_v^1) \simeq (\eta', m_v^2) : \sigma_1 \\ \text{we have } \Lambda \vdash (\eta_1, m_v\ m_v^1) \cong (\eta_2, m_v'\ m_v^2) : \sigma_2\end{array}$$

Figure 13: `TinyAspect` Observational Equivalence for Values

equivalent values. Otherwise, the expressions must be *bisimilar* with respect to the set of labels in Λ. That is, they must look up the same sequence of labels in Λ while either diverging or reducing to observationally equivalent values (since client aspects can use advice to observe lookups to labels in Λ).

We formalize this with three rules. The first allows two expressions to take any number of steps that does not include looking up a label in Λ (using the evaluation relation $\overset{\Lambda}{\mapsto}{}^*$ which is identical to $\mapsto^*$ except that the rule *r-lookup* may not be applied to any label in Λ). The second allows two expressions to lookup the same label in Λ. The third allows computation to diverge according to the first two rules, rather than terminating with a value.

Now that we have defined observational equivalence, we can state the abstraction theorem:

Theorem 3 (Abstraction)
If $\Lambda \vdash (\bullet, m_v) \cong (\bullet, m_v') : \sigma$*, then for all* d *such that* $x{:}\sigma; \bullet \vdash d : \beta$ *we have* $\Lambda \vdash (\bullet, \texttt{structure}\ x = m_v\ \ d) \cong (\bullet, \texttt{structure}\ x = m_v'\ \ d) : (x{:}\sigma, \beta)$

For space reasons, we give only a brief sketch of the proof of abstraction. More details are available in a companion technical report [1]. The proof uses a structural congruence property: the expressions are structurally equal except for closed values, which are observationally equivalent. A key lemma states that structural congruence is preserved by reduction.

We then observe that the two programs being compared are initially structurally congruent. By the lemma, they either remain structurally congruent indefinitely, corresponding to the divergence case of observational equivalence, or else they eventually reduce to values which are observationally equivalent.

5.2 Applying Abstraction

The abstraction theorem can be used to show that two different implementations of a module are equivalent and thus interchangeable. For example, Figure 15 shows two definitions of the Fibonacci function. The first one uses recursion

```
structure Fib1 = struct
    val fib = fn x:int => 1
    around call(fib) (x:int) =
        if (x > 2)
            then fib(x-1) + fib(x-2)
            else proceed x
end :> sig
    fib : int->int
end

structure Fib2 = struct
    val helper = fn x:int => 1
    around call(helper) (x:int) =
        if (x > 2)
            then helper(x-1) + helper(x-2)
            else proceed x
    val fib = fn x:int => helper x
end :> sig
    fib : int->int
end
```

Figure 15: Two equivalent modules that define the Fibonacci function

directly to compute the result, while the second one invokes a helper function. Since we have sealed both modules, it is easy to prove that they are equivalent. Clients can only advise the fresh label exported by the sealed modules, which doesn't affect the internal semantics of the module at all. Therefore, we can prove that the modules are equivalent by showing that the `fib` functions always return the same value when passed the same argument. A simple proof by induction on the argument value will suffice.

However, if we did not use `TinyAspect`'s sealing operation on these modules but instead used a more conventional module system to hide the `helper` function in `Fib2`, we would be unable to prove the modules equivalent. In this case, a client could advice `fib`, which would capture the recursive calls in module `Fib1` but not in module `Fib2`. Thus, the client's behavior would depend on the module's imple-

mentation.

This example shows that the properties of the module sealing operation are crucial for formal reasoning about aspect-oriented systems. Sealing is also important for more informal kinds of reasoning, for example allowing engineers to change the internals of a module with some assurance that clients will not be affected.

The Fibonacci example is simplistic in that it does not export any pointcuts to clients. However, similar equivalence properties can be proven in the presence of pointcuts, if it can be shown that two modules always treat their exported pointcut labels in an identical way, as defined by the observational equivalence relation.

6. Related Work

Formal Models. Walker et al. model aspects using an expression-oriented functional language that includes the lambda calculus, labeled join points, and advice [21]. They show that their model is type-safe, but they model `around` advice using a low-level exception construct and so their soundness theorem includes the possibility that the program could terminate with an uncaught exception error. `TinyAspect` guarantees both type safety and a lack of runtime errors because it models advice with high-level constructs similar to those in existing aspect-oriented programming languages. In addition, the declarative, source-level nature of `TinyAspect` allows us to easily explore modularity and prove an abstraction result.

Jagadeesan et al. describe an object-oriented aspect calculus modeling many of the features of AspectJ [10]. Their formal model is much richer than ours, capturing complex pointcuts and different forms of advice in a rich subset of Java. `TinyAspect` is intentionally much more minimal than their aspect calculus, so that it is easy to investigate language extensions such as a module system and prove properties such as abstraction.

In other work on formal systems for aspect-oriented programming, Lämmel provides a big-step semantics for a method-call interception extension to object-oriented languages [12]. Wand et al. give an untyped, denotational semantics for advice advice and dynamic join points [22]. Masuhara and Kiczales describe a general model of crosscutting structure, using implementations in Scheme to give semantics to the model [14]. Tucker and Krishnamurthi show how scoped continuation marks can be used in untyped higher-order functional languages to provide static and dynamic aspects [20].

Aspects and Modules. Dantas and Walker are currently extending the calculus of Walker et al. to support a module system [6]. Their type system includes a novel feature for controlling whether advice can read or change the arguments and results of advised functions. In their design, pointcuts are first-class, providing more flexibility compared to the second-class pointcuts in `TinyAspect`. This design choice breaks abstraction and thus separate reasoning, however, because it means that a pointcut can escape from a module even if it is not explicitly exported in the module's interface. In their system, functions can only be advised if this is planned in advance; in contrast, `TinyAspect` allows advice on all function declarations, providing unplanned extensibility without compromising abstraction.

Lieberherr et al. describe Aspectual Collaborations, a construct that allows programmers to write aspects and code in separate modules and then compose them together into a third module [13]. Since they propose a full aspect-oriented language, their system is much richer and more flexible than ours, but its semantics are not formally defined. Their module system does not encapsulate internal calls to exported functions, and thus does not enforce the abstraction property.

Other researchers have studies ways of achieving modular reasoning without the use of explicit module systems. For example, the Eclipse plugin for AspectJ includes a view showing which aspects affect each line of source code. Clifton and Leavens propose engineering techniques that reduce dependencies between concerns in aspect-oriented code [4].

Our module system is based on that of standard ML [15]. `TinyAspect`'s sealing construct is similar to the freeze operator in the module calculus of Ancona and Zucca, which closes a module to extension [3].

The name Open Modules indicates that modules are open to advice on functions and pointcuts exposed in their interface. Open Classes is a related term indicating that classes are open to the addition of new methods [5].

7. Future Work

In future work, we plan to extend the module system presented here to support recursive modules and abstract data types, as well as supporting modules that can be loaded and instantiated at run time. We would like to extend the base language with polymorphism, references, and objects; enforcing abstraction in the context of these features is an open problem. Based on this foundation, we intend to design and implement a user-level language with aspect-oriented features, including richer mechanisms for pointcuts and advice.

8. Conclusion

This paper described `TinyAspect`, a minimal core language for reasoning about aspect-oriented programming systems. `TinyAspect` is a source-level language that supports declarative aspects. We have given a small-step operational semantics to the language and proven that its type system is sound. We have described a proposed module system for aspects, formalized the module system as an extension to `TinyAspect`, and proved that the module system enforces abstraction. Abstraction ensures that clients cannot affect or depend on the internal implementation details of a module. As a result, programmers can both separate concerns in their code and reason about those concerns separately.

9. Acknowledgments

I thank Ralf Lämmel, Karl Lieberherr, David Walker, Curtis Clifton, Derek Dreyer, Todd Millstein, Robert Harper, and the anonymous reviewers for comments and conversations on an earlier draft of this paper.

10. REFERENCES

[1] J. Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. Carnegie Mellon Technical Report CMU-ISRI-04-108, available at http://www.cs.cmu.edu/~aldrich/aosd/, March 2004.

[2] J. Aldrich. Open Modules: Reconciling Extensibility and Information Hiding. In *AOSD workshop on Software*

Engineering Properties of Languages for Aspect Technologies (SPLAT '04), March 2004.

[3] D. Ancona and E. Zucca. A Calculus of Module Systems. *Journal of Functional Programming*, 12(2):91–132, March 2002.

[4] C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages*, April 2002.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[6] D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Unpublished manuscript, 2003.

[7] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Advanced Separation of Concerns*, October 2000.

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

[9] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[10] R. Jagadeesan, A. Jeffrey, and J. Riely. An Untyped Calculus of Aspect-Oriented Programs. In *European Conference on Object-Oriented Programming*, July 2003.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, June 2001.

[12] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Aspect-Oriented Software Development*, Apr. 2002.

[13] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[14] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *European Conference on Object-Oriented Programming*, July 2003.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[16] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, September 2001.

[17] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[18] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*, 1983.

[19] P. Tarr, H. Ossher, W. Herrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Interntional Conference on Software Engineering*, May 1999.

[20] D. B. Tucker and S. Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Aspect-Oriented Software Development*, March 2003.

[21] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *International Conference on Functional Programming*, 2003.

[22] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *Transactions on Programming Languages and Systems*, To Appear 2003.

[23] J. C. Wichman. ComposeJ - The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Masters Thesis, University of Twente, 1999.

Call and Execution Semantics in AspectJ

Ohad Barzilay
School of Computer Science
Tel Aviv University
ohadbr@cs.tau.ac.il

Yishai A. Feldman
Efi Arazi School of Computer Science
The Interdisciplinary Center, Herzliya
yishai@idc.ac.il

Shmuel Tyszberowicz
Department of Computer Science
The Academic College of Tel Aviv Yaffo
tyshbe@mta.ac.il

Amiram Yehudai
School of Computer Science
Tel Aviv University
amiramy@post.tau.ac.il

ABSTRACT
The Aspect-Oriented Programming methodology provides a means of encapsulation of crosscuting concerns in software. AspectJ is a general-purpose aspect-oriented programming language that extends Java. This paper investigates the semantics of call and execution pointcuts in AspectJ, and their interaction with inheritance. We present a semantic model manifested by the current (1.1.1) release of AspectJ, point out its shortcomings, and present alternative models.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms
Languages

Keywords
Aspect-oriented programming, AspectJ

1. INTRODUCTION
Many papers and books have been written about Aspect-Oriented Programming (AOP) in general, and about AspectJ in particular (e.g., [1, 2, 4]), as well as several papers giving formal semantics of simple aspect-oriented languages (e.g., [3, 5, 6, 8–10]), but none of them provides a precise (even if not completely formal) semantics of AspectJ. Such a semantics is necessary for language users to express their intent, and is crucial for tools that compile into AspectJ. For example, we are developing a design-by-contract [7] tool for Java. The main purpose of such a tool is to instrument the code to check assertions (method pre- and postconditions and class invariants) at run time. Existing tools we have examined perform this instrumentation in various ways, all of which have subtle errors. Our tool uses AspectJ instead of ad-hoc methods. While working on the tool, we discovered that some pointcuts we wrote did not yield the sets of join points that we expected. This has led us to conduct the study that we report on here.

We believe that a close examination of the semantics of AspectJ as manifested by the current implementation, and a discussion of the desired or "correct" semantics, is important to the AOP community. We hope that studies of the semantics of other parts of the language will follow. This paper investigates one of the subtle parts of AspectJ, namely, call and execution pointcuts and their interaction with inheritance. We present a semantic model manifested by the current (1.1.1) release of AspectJ, point out its shortcomings, and present alternative models. We note that Jagadeesan et al. [3] mention a few of these shortcomings, but do not discuss their deficiencies.

We follow the approach taken by authors of the AspectJ documentation and books by ignoring implementation issues. For the purpose of this paper, we are not interested in how code instrumentation is carried out, and in the practical constraints on which classes may or may not be instrumented. We similarly ignore the implementation of the matching between pointcuts and join points in AspectJ. Instead, we treat AspectJ as a black box, and examine its performance on carefully-chosen test cases.

2. CURRENT SEMANTICS OF ASPECTJ
The semantics of the wildcard operators ("`*`" and "`..`") inside call and execution pointcuts are easily specified by considering them to be an abbreviation for the (infinite) union of all possible expansions. We will therefore ignore wildcards in the sequel. Also, in order to simplify the presentation, we will deal only with void functions of no arguments. This will entail no loss of generality. Since static methods are not inherited, we will also ignore those in the sequel.

2.1 Call Semantics

Consider the pointcut specified by `call(void A1.f())`. This should capture all calls to the method `f` defined in class `A1`. Indeed it does, but that is due to the careful wording of the previous sentence. What happens if `f` is inherited from another class? In order to answer this question, we will consider the following hierarchy of classes:

```
public class A1
{
  public void f() {}
  public void g() {}
}

public class A2 extends A1
{
  public void h() {}
}

public class A3 extends A2
{
  public void f() {}
}
```

We then consider the following three variable definitions, in which the name of the variable indicates its static type and, if different, also its dynamic type:

```
A1 s1 = new A1();
A3 s3 = new A3();
A1 s1d3 = new A3();
```

It turns out that the pointcut `call(void A1.f())` captures the calls `s1.f()`, `s3.f()`, and `s1d3.f()`. Similarly, the pointcut `call(void A1.g())` captures the calls `s1.g()`, `s3.g()`, and `s1d3.g()`. It seems that even without the `+` subtype pattern modifier, which specifies subclasses, these pointcuts capture calls to the same method in subclasses, whether inherited or overridden. This may be a little surprising—what is the `+` modifier for, then?—but is consistent with the dynamic binding mechanism of Java. (We shall have more to say about the `+` modifier later.)

The pointcut `call(void A3.f())` captures the call `s3.f()` but not `s1d3.f()`. This implies that matching of call pointcuts is based on the static type of the variable, which is *not* consistent with the dynamic binding principle, but may perhaps be justified based on the information available at the calling point. However, the real surprise is that the pointcut `call(void A3.g())` does not capture *any* join points in our example, not even `s3.g()`! The only difference between `f` and `g` in `A3` is that `f` is overridden whereas `g` is only inherited. Thus, it seems that for matching to succeed, it is necessary for the method to be lexically defined within the specified class—inheritance is not enough. We use the term "lexically defined" to indicate that a definition (first or overriding) of the method appears inside the definition of the class.

Thus we are led to the following model. The semantics of a pointcut will be given as a set of join points, formalized as a predicate specifying which join points are captured by the pointcut. Consider the following definitions:

- a pointcut $pc_c = \texttt{call(void } C.f\texttt{())}$,
- a variable defined as $S\ x$ `= new` D`()`, and
- a join point $jp = x.f\texttt{()}$.

That is, the pointcut specifies a class C, and the target of the join point has the static type S and the dynamic type D. (Obviously, D must be a descendant of S for this to compile correctly. We will denote this relationship by $S \subseteq D$.) Then:

$$jp \in pc_c \iff S \subseteq C \ \wedge\ f \text{ is lexically defined in } C.$$

2.2 Execution Semantics

Continuing with our example, we find that call and execution pointcuts capture exactly the same join points for `s1` and `s3` (we are ignoring other features of pointcuts, such as `this` and `target`). The only difference is in the treatment of `s1d3.f()`, which is captured by `execution(A1.f())` and `execution(A3.f())` but not by `call(A3.f())`. However, `execution(void A3.g())`, like the corresponding call pointcut, captures none of our join points. Thus, the rule for an execution pointcut

$$pc_e = \texttt{execution(void } C.f\texttt{())}$$

seems to be:

$$jp \in pc_e \iff D \subseteq C \ \wedge\ f \text{ is lexically defined in } C.$$

That is, the static type is replaced by the dynamic type. Again, this can be justified by the different type information available at execution join points, but is nevertheless an inconsistency in the semantics.

2.3 Subtype Pattern Semantics

The semantics of a subtype pattern such as `call(A1+.f())` should naturally be equivalent to the union of all possible expansions where `A1` is replaced by any of its descendants. This is indeed the case in AspectJ. However, because of the surprising semantics described above, this has a subtle interpretation. If

$$pc_c^+ = \texttt{call(void } C\texttt{+.}f\texttt{())}$$

is a call pointcut using subtypes, the matching rule is:

$$jp \in pc_c^+ \iff S \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } S \subseteq F \subseteq C.$$

In particular, the pointcut `call(A1+.h())` captures `s3.h()`, because `h` is defined in `A2`, but the same join point is *not* captured by `call(A3+.h())`, even though `A3` has this method. This violates our expectation that `call(A3+.h())` should be a subset of `call(A1+.h())` that is identical for all join points in classes under `A3`.

Similarly, for

$$pc_e^+ = \texttt{execution(void } C\texttt{+.}f\texttt{())},$$

the matching rule is:

$$jp \in pc_e^+ \iff D \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } D \subseteq F \subseteq C.$$

Variable definition:	$S\ x$ `= new` D`()`
Join point:	$jp = x.f$`()`
Pointcuts:	$pc_c =$ `call(void` $C.f$`())`
	$pc_e =$ `execution(void` $C.f$`())`
	$pc_c^+ =$ `call(void` C`+.`f`())`
	$pc_e^+ =$ `execution(void` C`+.`f`())`

$$
\begin{array}{lcl}
jp \in pc_c & \iff & S \subseteq C \ \wedge\ f \text{ is lexically defined in } C \\
jp \in pc_e & \iff & D \subseteq C \ \wedge\ f \text{ is lexically defined in } C \\
jp \in pc_c^+ & \iff & S \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } S \subseteq F \subseteq C \\
jp \in pc_e^+ & \iff & D \subseteq C \ \wedge\ f \text{ is lexically defined in some } F \text{ s.t. } D \subseteq F \subseteq C
\end{array}
$$

Figure 1: Semantics of current (1.1.1) AspectJ implementation.

2.4 Summary

The current semantics of AspectJ is summarized in Figure 1. It satisfies some of our intuitive expectations but violates others. The points on which AspectJ is consistent with the intuitive semantics are:

- Pointcuts with wildcards are equivalent to the union of all possible expansions.
- Pointcuts with subtype patterns are equivalent to the union of all pointcuts with subtypes substituted for the given type.
- The semantics of execution pointcuts is based on the dynamic type of the target.

On the following points the semantics of AspectJ deviates from our intuition:

- The semantics of call pointcuts is different from that of execution pointcuts, and depends on the static type of the target.
- Call and execution pointcuts only capture join points for classes where the given method is lexically defined.
- As a result of this, the difference between pointcuts with or without subtype patterns is subtle and unintuitive.

It is arguable whether pointcuts without subtype patterns should capture join points in subclasses at all. On the one hand, an instance of a class is *ipso facto* considered to belong to all its superclasses; this is reflected in the syntactic restrictions on assignment and parameter passing, and in the semantics of the `instanceof` operator. On the other hand, the existence of the subtype pattern modifier seems to imply the intention that a pointcut that does not use it refer only to instances of the specified class.

We believe that the lexical restrictions shown in these semantic definitions were unintended; their removal would greatly simplify the semantics. Some evidence that this is not the intended semantics comes from the following quote from one of the AspectJ gurus [4, p. 79]: "The [`call(* Account.* (..))` pointcut] will pick up all the instance and static methods defined in the `Account` class *and all the parent classes in the inheritance hierarchy*" (emphasis added). This is not true in AspectJ, but is intuitively appealing.

Another interesting clue is the fact (pointed out to us by one of the anonymous reviewers) is that when the AspectJ compiler is invoked with the `-1.4` switch, the set of joinpoints defined by call pointcuts changes, and the restriction on the lexical definition of the method in the designated class is removed. Curiosly, the behavior of execution pointcuts does *not* change even with this switch.

3. ALTERNATIVE SEMANTICS

If the current AspectJ semantics is inappropriate, we should propose one or more alternatives. As mentioned above, such alternatives should not restrict methods to be lexically defined in the designated class. Two questions remain:

1. should subclasses be included when the subtype pattern modifier does not appear in the pointcut; and
2. should call and execution pointcuts capture different join points.

These issues lead to four possible definitions of the semantics (see Figure 2). In these definitions we use the term "f exists in C" to denote the fact that the method f exists in class C, whether or not it is lexically defined (or overridden) in it. We use the term "broad" for those semantics that include subclasses even when subtypes are not indicated, and "narrow" for those that do not. The term "static" denotes semantics that use the static type for call pointcuts, and "dynamic" denotes those that use the dynamic type. It is important to note that although the join points captured by call and execution pointcuts are the same in the dynamic semantics, their properties (e.g., `this` and `target`) are different.

Each of the four semantics is consistent and reasonable. Perhaps the broad–dynamic semantics best reflects object-oriented principles, in that a reference to a class includes its subclasses, and the type that determines matching is the dynamic rather than static type of the variable. However,

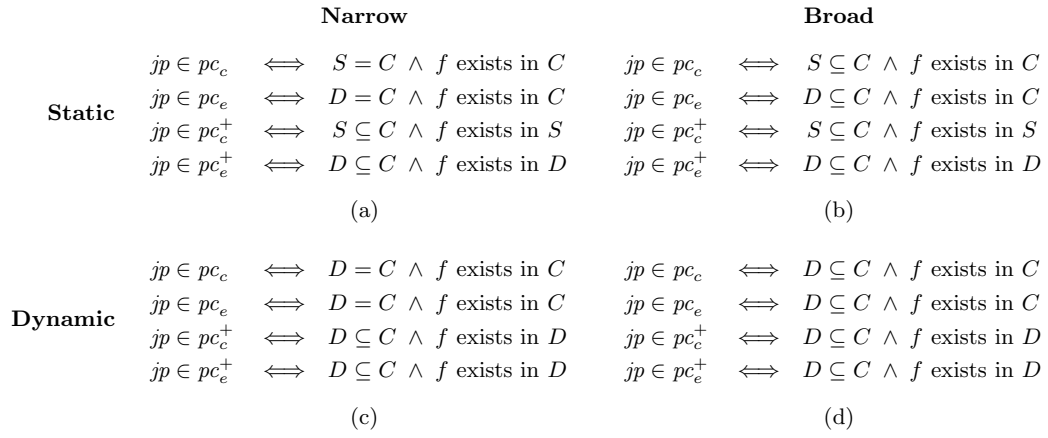

Figure 2: Four possible semantics: (a) narrow–static; (b) broad–static; (c) narrow–dynamic; (d) broad–dynamic.

other semantics may be easier to use if they more closely reflect the intent of AspectJ programmers.

4. EXPRESSIVE POWER

The five semantic models presented above (current AspectJ semantics and four alternatives) are able to describe different sets of join points. However, AspectJ has additional pointcut designators, which may be used to modify the meaning of a pointcut. The question now is, what is the expressive power of each of the given semantics definitions? Are there meaningful sets of join points that can only be expressed by some of them?

The answer is, of course, positive. For example, a narrow semantics is easily expressed in the corresponding broad semantics. The pointcut `call(void C.f())`, whose meaning in the narrow–static semantics is "$S = C \wedge f$ exists in C" can be expressed in the broad–static semantics by the following pointcut:

```
call(void C.f()) && target(x) &&
   if(x.getClass() == C.class)
```

However, the reverse is not true: in order to get a subset relation in the narrow semantics, we must use the subtype pattern modifier, but then there is no way to enforce the requirement that the method already exists in class C. So each broad semantics is strictly more expressive than the corresponding narrow semantics.

The static and dynamic semantics are incomparable. The dynamic semantics have no way of referring to the static type (S), and the static semantics have no way of referring to the dynamic type (D) in call pointcuts.

The semantics of `execution(void C+.f())` in either of the dynamic semantics is easily expressed in the current AspectJ semantics by the expression

```
execution(void f()) && this(C).
```

The corresponding expression for `call(void C+.f())` is

```
call(void f()) && target(C).
```

(Note that `target` in call pointcuts corresponds to `this` in execution pointcuts.) In order to understand the semantics of this expression under AspectJ, note that the call pointcut `call(void f())` without a class designator is equivalent to `call(Object+.f())`, so when applying the semantics of Figure 1, the class inclusion condition is trivial, and we obtain simply that f exists in S. Together with the additional requirement, `target(C)`, we get that the semantics of the above expression in AspectJ is

$$D \subseteq C \wedge f \text{ exists in } S,$$

which is a little different from the dynamic semantics.

Under the current semantics, AspectJ has no way of requiring that f exist in C without being lexically defined in it. The alternative, going to the top of the inheritance hierarchy, then prevents the possibility of referring to the static type. On the other hand, the new proposed semantics have no way of requiring the lexical definition of a method in some class. (Note that the `within` and `withincode` constructs are too restrictive, because they do not capture overriding definitions. Also, these do not help with call pointcuts, because they refer to the caller code rather than the method implementation.)

Of course, the fact that one semantics is more expressive than another does not mean it is better. The question is what programmers (and automatic tools) really need to say. Furthermore, the cost of a complex semantics should be weighed against the convenience of the language. Common patterns of usage should be expressed concisely. It might be better to adopt a simpler semantics for call and execution pointcuts, and add another construct to capture lexical definitions, if this is indeed necessary.

5. CONCLUSIONS

The current semantics of AspectJ has some unintuitive aspects. We have presented a number of alternative semantics, and compared their expressive power. The "right" semantics for AspectJ needs to be worked out with the user community, since it ultimately depends on how AspectJ is used in

practice. We hope that this paper will start a fruitful and constructive discussion on this question.

6. REFERENCES

[1] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Comm. ACM*, 44(10):29–32, 2001.

[2] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, 2003.

[3] R. Jagadeesan, A. Jeffrey, and J. Reily. A calculus of untyped aspect-oriented programs. In Luca Cardelli, editor, *ECOOP 2003, European Conference on Object-Oriented Programming, Darmstadt, Germany*, volume 2743 of *Lecture Notes in Computer Science*, pages 54–73. Springer-Verlag, NY, 2003.

[4] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[5] R. Lämmel. A semantical approach to method-call interception. In *Proc. First Int'l Conf. Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, April 2002.

[6] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Ninth Int'l Workshop on Foundations of Object-Oriented Languages*, 2002.

[7] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[8] D. B. Tucker and S. Krishnamurthi. A semantics for pointcuts and advice in higher-order languages. Technical Report CS-02-13, Brown University, 2003.

[9] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, pages 127–139. ACM Press, August 2003.

[10] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Ninth Int'l Workshop on Foundations of Object-Oriented Languages*, 2002.

Using Program Slicing to Analyze Aspect Oriented Composition

Davide Balzarotti
Dip. Elettronical e Informatica
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano, Italy
balzarot@elet.polimi.it

Mattia Monga
Dip. Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41
20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

AspectJ language was proposed to make cross-cutting concerns clearly identifiable with special linguistic constructs called aspects. In order to analyze the properties of an aspect one should consider the aspect itself and the part of the system it affects. This part is just a slice of the entire system and can be extracted by exploiting program slicing algorithms. However, the expressive power of AspectJ constructs forces slicers to take into account big portions of programs. We suggest that AspectJ should regulate more formally the interaction among code units, by defining some stricter boundaries around aspect influence, otherwise the separation turns out to be just syntactic sugar.

1. INTRODUCTION

Aspect oriented languages claim to be able to provide linguistic support to make cross-cutting concerns isolated in proper code units. Currently the most successful aspect-oriented language is probably AspectJ [17]. Designed and implemented at Xerox PARC, it is aimed at managing tangled concerns in Java programs.

AspectJ provides first-class entities called **aspect**s that, in a strong analogy to regular Java **class**es, can define fragments of code called *advices* that will be woven at run time **before**, **after**, or **around** interesting points (*join points*) in the whole program. AspectJ showed up to be very convenient to express cross-cutting concerns. A typical AspectJ advice can be something like "before any call to the division function, check if the divisor is not zero"; in a very economical way it is possible to affect all the divisions in the code, even without knowing where these divisions will occur.

The problem we want to discuss in this paper is if the aspect oriented computation is actually separated from the rest of the program by using the linguistic construct provided by AspectJ. In fact, on the one hand a "no division by zero" aspect would be a isolated code unit. However, on the other hand it might be difficult to figure out the behavior of the whole system: every time the division function is called, one has to consider that also the aspect oriented code is executed. In order to asses the resulting complexity of an aspect oriented program, we tried to apply well known techniques of program comprehension, namely static analysis and program slicing, to AspectJ.

In the rest of the paper we describe the results of our preliminary experiments. The discussion is organized as follows: in Section 2 we briefly introduce program slicing techniques, in Section 3 we propose our approach to slice aspect oriented programs, in Section 4 we examine the problems we found, and finally in Section 5 we draw some conclusions.

2. ASPECT ORIENTED PROGRAM SLICING

Program slicing was proposed by Weiser [16] in the early 80's. It is a technique aimed at extracting program elements related to a particular computation. A *slice* of a program is the set of statements which affect a given point in a executable program (*slicing criterion*). One can compute statically the set of statements that potentially affect the slicing criterion for every possible program execution (static slicing), or one can consider the information about a particular execution of the program and derive a dynamic slice [3] of a program.

Different slicing algorithms and different type of slice have been proposed by many authors [14, 5]. Nowadays a widely adopted approach to compute static program slicing consists in re-formulating the problem as a reachability problem on a particular graph representation of the program, which, for inter-procedural slicing, is the so called *system dependencies graph* (SDG) [7]. It is worth noting that, while producing the minimal slice is known to be uncomputable, it is possible to compute non-minimal slices with fairly efficient algorithms.

Program slicing was initially studied for procedural programming language but has then be extended to cope with the object oriented paradigm by Larsen and Harrold [9, 10] (specific solutions for Java language are proposed in [11, 8, 18]). Notwithstanding the rich theoretical work done in the

field, the only publicly available tool we are aware of which is able to compute a program slice of a Java program is Bandera [4].

The application of program slicing techniques to aspect oriented software is a novel research topic.

A preliminary work in this area has been done by Zhao [19]. He proposed an aspect-oriented SDG that is a further extension of the object-oriented SDG. The aspect-oriented system dependence graph (ASDG) consists of an SDG for the traditional code enriched with a set of dependence graphs that represent the aspect code. Graphs are connected through special edges that model introductions and advice execution. He focused on AspectJ, however he did not consider that aspect advices might apply to the aspect oriented code itself.

We start analyzing the problem of slicing aspect oriented in [2]. We initially proposed an approach based on the concept of *conjugated class* of an aspect to allow the application of existing object oriented algorithms. A conjugated class contains all members and methods of the originating aspect and it has a method for each advice. We did not propose a complete solution of the problem, since we did not describe how conjugated class should be connected to the rest of the program and we were not able to cope with introductions and other subtilties of AspectJ syntax. In the next Section, we describe the new approach we think is most suitable to implement a real tool able to slice a larger family of AspectJ programs.

3. SLICING JAVA BYTE-CODE

Since AspectJ programs are eventually woven in Java byte-code binaries, which are executed by a Java Virtual Machine, in order to slice them two different approaches are possible: (1) one can consider methods and advices as first-class entities and try to extend SDGs to take them into account [2, 19, 13]; otherwise, (2) one can try to analyze the woven program by applying existing techniques and map the results on the original structure of the program.

The high-level approach is conceptually more appealing, since it does not depend on the actual implementation of the AspectJ weaver, and, more fundamentally, it enables the use of aspects as first-class entities in the resulting model. However, building a working tool is far from trivial, because it needs to be able to manage a several AspectJ syntax details. In particular, the AspectJ pointcut definition language allows programmers to characterize pointcuts on a wide range of abstraction levels:

- Lexical (**withincode**, regular expression on identifiers, etc.)
- Statically known interfaces (**void** ∗.`func`(**int**), etc.)
- Run time events (**call**, **execution**, **set**, **if**, etc.)

In order to build as quick as possible a tool for experimenting with and slicing real world programs, we adopted a more pragmatic strategy:

1. Compile classes and aspects using the AspectJ compiler.
2. Weave aspects into an executable program.
3. Apply existing slicing algorithms (we built upon the Soot static analysis framework [15]) to the resulting byte-code.
4. Obtain a slice, as a set of byte-code statements.
5. Map the results onto the original aspect oriented source code.

Working at the level of Java byte-code could appear not appropriate because any distinction among classes and aspects may seem to be lost. The AspectJ weaver translates aspects in classes, advices in methods, and join points in methods invocation. Thanks to this approach, it is not difficult to map every statements to its original aspect (or class). However, a tool based on byte-code slicing has to be changed when the AspectJ weaver modifies its implementation strategy.

Figure 1 shows an example that contains only a trivial class `C` and an aspect `A`. The aspect introduces a public field into the class and it defines an advice that print a value when `method2`() is called. The resulting woven classes are shown on the right hand side. The aspect has been translated in a class and the advice in an equivalent method. A decompilation of `C.method`() produces:

```
...
 3 invokevirtual #17 <Method void method2()>
 6 invokestatic #31 <Method A aspectOf()>
 9 aload_1
10 invokevirtual #34
      <Method void ajc$afterReturning$A$21(C)>
...
```

It is easy to identify the call to after-returning advice after the invocation of `method2`().

Thus, by using dedicated libraries it is fairly easy to build a tool for inspecting the Java byte-code, obtaining the call graph, and performing the def-use analysis. It is then possible to implement existing algorithms to construct the system dependence graph and to calculate static or dynamic slices.

4. ANALYSIS OF ASPECT INTERACTION

The final goal of our work is to be able to analysis interactions among aspects. An aspect oriented program is composed by weaving aspect and classes together. An aspect is conceptually *a posteriori* with respect to the rest of a system. When a programmer writes a new aspect, s/he assumes that the rest of the system is working correctly and s/he hopes to add the new cross-cutting functionality *without breaking the system*. How one can check that the new aspect does not interfere with existing aspects and classes?

Let a *code unit* be an aspect or a class of a system. We say that an aspect A does not interfere with a code unit C if and only if every interesting predicate on the state manipulated

```
class C{
 void method() {
    method2();
 }
 void method2(){}
}

aspect A {
 public int C.x = 10;
 after(C c) returning:
     target(c) &&
     call(void C.method2())
 {
   System.out.println(c.x);
 }
}
```

```
Compiled from C.java
public class C extends java.lang.Object {
    public int x;
    C();
    void method();
    void method2();
}

Compiled from A.java
public class A extends java.lang.Object {
    public static final A ajc$perSingletonInstance;
    static {};
    A();
    public static void ajc$interFieldInit$A$C$x(C);
    public static int ajc$interFieldGetDispatch$A$C$x(C);
    public static void ajc$interFieldSetDispatch$A$C$x(C, int);
    public void ajc$afterReturning$A$21(C);
    public static A aspectOf();
    public static boolean hasAspect();
}
```

Figure 1: A simple class before and after the weaving of an aspect

by C is not changed by the application of A. For example, if an object x manipulated by C exists such that the predicate $x \leq 0$ must hold for the correctness of the system, A does not interfere with C only if C woven with A preserves $x \leq 0$.

In [2] we proposed the following sufficient condition to check non-interference between aspects:

> Let A_1 and A_2 be two aspects and S_1 and S_2 the corresponding backward and forward slices obtained by using the pointcuts declarations defined in A_1 and A_2 as slicing criteria. A_1 does not interfere with A_2 if
>
> $$S_1 \cap S_2 = \emptyset$$

This condition may be too strong: in fact, two aspects may not interfere also if their slices share some statements.

A weaker and more practical condition is

> Let A_1 and A_2 be two aspects and S_1 and S_2 the corresponding backward slices obtained by using all the statements defined in A_1 and A_2 as slicing criteria. A_1 does not interfere with A_2 if
>
> $$A_1 \cap S_2 = \emptyset$$

S_2 contains all statements that affect the slicing criterion (which contain all the statements of A_2). It is worth noting that the interference relation is not symmetric. In fact, it is possible that A_1 interferes with A_2 but A_2 does not interfere with A_1. For example, if A_2 is a tracing aspect and A_1 is an aspect that change the order in which procedures are called, the application of A_2 does not change an existing A_1, but the application of A_1 onto A_2 change its behavior.

Moreover, modifications in the type hierarchy can be difficult to deal with. Consider the following program:

```
class ClassA{
  void method() {}
  public static void main(String[] args) {
   ClassA a = new ClassA();
   a.method();
  }
}

class ClassB {}

aspect A1 {
  declare parents: ClassA extends ClassB;
}

aspect A2 {
  before(): call(* ClassB+.*()) {
    // ....
  }
}
```

If we apply the aspect `A2` but not the aspect `A1` the advice is never executed. When we add to the program the aspect `A1` the advice is executed before the invocation of `ClassA.method()`. By using **declare parents**: we forced the members of `ClassB` to become part of `ClassA` also.

4.1 On the Precision of Slicing

A well-known result asserts that, in general, the problem of finding the minimum static slice is incomputable [16]. This means that when we compute a slice, the result may contains some unnecessary nodes. This is not a problem in most cases but sometime can lead to an incorrect result.

For interference analysis, Figure 2(a) shows how a non-minimum slice might cause a decision error when we analyze the interference between aspects. The result is always correct if the algorithm does not find any intersection between the two sets, but could be wrong if an intersection occurs.

Another problem that could have a significant effect on slice precision is the resolution of aliases. An alias occurs when two or more different variables refer to the same memory location. The computation of slices relays on the construction of the SDG, that in turn requires to calculate control

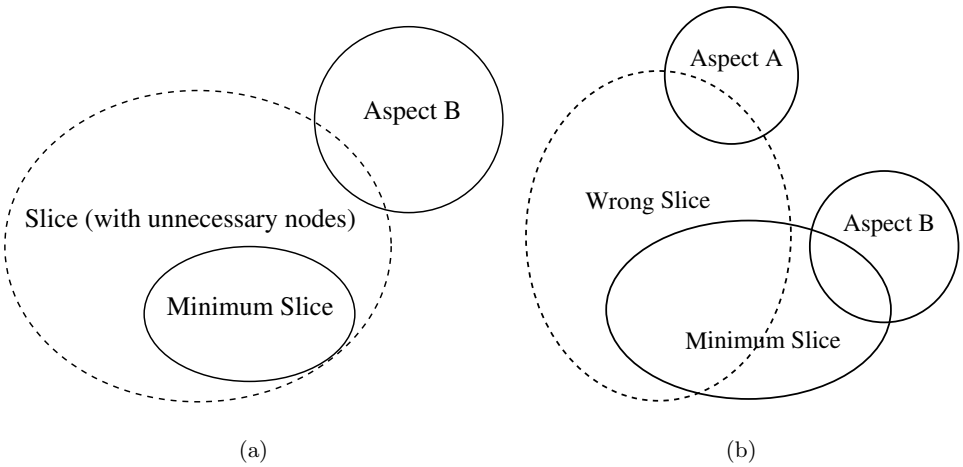

Figure 2: Incorrect results in presence of non-correct slice

and data dependencies among every program statement. In presence of aliasing the exact computation of data dependency becomes a very difficult task [12]. In order to mitigate the impact of aliases it is necessary to adopt a generalization of the notion of data dependence [1]. If the algorithm does not correctly take into consideration all may-alias variables (note that this is possible only assuming a closed world hypothesis), the resulting slice may omit some required statement. Thus, in general we might find a slice that contains statements that are unnecessary (since we cannot compute the minimum slice) and omits others (since pragmatic issues could force us to use approximate algorithms). Figure 2(b) shows an example in which the slice identifies an incorrect interference with the aspect A and does not identify the correct interference with the aspect B. Therefore, in order to keep significant our non-interference criterion, we have to adopt a conservative approach that guarantees that every possible may-alias are take in consideration. This resolve the problem of false negative (when the algorithm does not find an existing interference) but it tends to make the slice bigger, increasing the number of false positive (when the algorithm finds a false interference).

Moreover, some AspectJ join points are not statically determinable. AspectJ provides some primitives to declare pointcuts that discriminates based on the dynamic context (i.e. **cflow**, **cflowbelow**, **if**, **this**, ...). A conservative approach requires to consider every possible execution trace, but this may further increase the number of false-positive response.

Adding together all the previous considerations, the final precision of a hypothetic tool that analyze aspect interaction using static slicing techniques may become quite low. In general, program slicing can be used to automatically build an abstraction (i.e., a simplified model) of a program. Often the goal is to reduce the dimension of a program in order to reduce the complexity of algorithms that are exponential in the number of program statements, for example in order to be able to apply a model checking tool [4]. In this case, the size of slice is a minor issue, because every discarded statement is a benefit anyway. However, when one is interested in deciding if a given statement belongs to a slice, the dimension of the slice becomes a major issue, because the accuracy of the decision depends on it.

5. CONCLUSIONS AND FUTURE WORK

In object oriented programs one can define *composition invariants* which are properties of classes that are preserved in every possible composition of objects. Software engineers leverage on this in order to reason on the properties of whole object oriented systems without considering all the details.

Unfortunately, in general no composition invariants are guaranteed to hold anymore, when AspectJ is in use. We tried to derive the slice of a system affected by an aspect, but the loosely regulated expressiveness of AspectJ constructs causes a turbulent *ripple effect* that in general forces to take into account most of the statements of a system. Moreover, a whole system analysis is needed, because arbitrary introductions and modifications of type hierarchy require a closed world assumption to be resolvable. This is in part due to the *obliviousness* that Filman identifies as an intrinsic characteristic of aspect orientation [6], but it is amplified by the undisciplined power of AspectJ constructs. The ultimate goal of aspect-oriented programming is the separation of otherwise cross-cutting concerns. However, these benefits are lost if the comprehension of aspect properties entails the analysis of the whole program. Instead, if we would be able to define some boundaries around aspect influence, the separation turns out to be not just syntactic sugar but a true aid in dealing with program complexity.

Currently we are improving our tool for slicing Java bytecode that we want to apply to AspectJ code. We are interested in evaluating the actual impact of AspectJ constructs in real world aspect oriented system. Our final goal is to define patterns of use and/or new AspectJ constructs to improve the comprehensibility and maintainability of AspectJ programs.

6. REFERENCES

[1] D. W. Binkley and K. B. Gallagher. Program slicing. *Advances of Computing*, 43:1–50, 1996.

[2] L. Blair and M. Monga. Reasoning on AspectJ programmes. In *Proceedings of Workshop on Aspect-Oriented Software Development*, pages 45–50, Essen, Germany, Mar. 2003. German Informatics Society.

[3] J. W. L. Bogdan Korel. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[5] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[6] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of OOPSLA 2000 workshop on Advanced Separation of Concerns*, 2000.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[8] G. Kovcs, F. Magyar, and T. Gyimthy. Static slicing of Java programs.

[9] L. Larsen and M. Harrold. Slicing object-oriented software. In *In Proceedings of the 18th International Conference on Software Engineering*, pages 45–50. Association for Computer Machinery, Mar. 1996.

[10] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[11] M. W. Neil Walkinshaw, Marc Roper. The Java system dependence graph. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, page 55, Sept. 2003.

[12] A. Orso, S. Sinha, and M. J. Harrold. Effects of pointers on data dependences. Technical Report GIT-CC-00-33, College of Computing, Georgia Institute of Technology, Dec. 2000.

[13] M. Stoerzer. Analysis of AspectJ programs. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, Mar. 2003.

[14] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[15] R. Vall, e Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay. Soot - a Java bytecode optimization framework. 1999.

[16] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, Mar. 1981.

[17] XEROX Palo Alto Research Center. *AspectJ: User's Guide and Primer*, 1999.

[18] J. Zhao. Applying program dependence analysis to Java software. In *Proceedings of Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, December 1998.

[19] J. Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pages 251–260, June 2002.

Aspect Reasoning by Reduction to Implicit Invocation

Jia Xu
Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2296
jx9n@cs.virginia.edu

Hridesh Rajan
Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2296
hr2j@cs.virginia.edu

Kevin Sullivan
Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2206
sullivan@cs.virginia.edu

ABSTRACT
Aspect-oriented programming constructs complicate reasoning about program behavior. Our position is that we can *reduce* key elements of aspect programming to implicit invocation (II) and then use existing work on reasoning about II to reason formally about aspect programs. We map aspect-oriented programs to equivalent programs with join points and advice replaced by event notifications and observers; use existing techniques for reasoning about programs that use implicit invocation; and then interpret the results in the context of the original aspect-oriented program.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Measurement, Design, Experimentation, Languages, Verification.

Keywords
AOP, Implicit Invocation, Reasoning, Model Checking

1. INTRODUCTION
The ability of aspect-oriented [15][16] approaches to enable modular representation of crosscutting concern by implicit behavioral modifications at join points specified by predicates on program elements is a bane for reasoning. In the presence of aspects, the behavior of a module at runtime can not be determined by just looking at the module code. One is required to understand the possible effects of each aspect in the system. The actual behavior is determined by composing the base and aspect behaviors. It is widely understood that reasoning about AOP remains a challenge [5]. Our position is that a reduction from the space of aspect-oriented programs to the space of programs using implicit invocation has the potential to enable formal reasoning about properties of aspect-oriented programs using existing methods for reasoning about implicit invocation systems.

The problem is well known. Several approaches have been proposed to enable automated reasoning about aspect-oriented programs. Some of these approaches try to apply model checking to verify properties of aspect-oriented programs [3][19][24], while others try to reduce aspect-oriented programming model to simpler models which can be easier to reason about [1][5][6][7].

Our contribution is in seeing how to exploit the relationship between join points and events in implicit invocation systems [12]. In such systems, modules expose events, with which other modules register procedures. Registered procedures are invoked when modules announce events, extending the modules' behaviors implicitly. Implicit invocation is widely used for complex system design. Sullivan and Notkin [23] showed how implicit invocation enables separation of integration concerns to ease the design and evolution of integrated systems and how it poses AOP-like problems in reasoning about II systems.

The problem of reasoning about implicit invocation (II) has generated significant interest over the last decade. In particular, Garlan et al. [11] proposed an event model to describe the behavior of the II systems. They then use model checker to check property assertions on this event model. Bradbury et al [4] further refined Garlan et al.'s approach and evaluated their approach in real world software systems, demonstrating the feasibility of applying formal reasoning techniques to real II systems.

Our position is that reducing the join point and advice model of aspect programming to II is possible, as shown by Eos [18], and that this reduction permits formal reasoning techniques for II systems to be applied to aspect programs. We first map an aspect-oriented program that uses join point and advice to a semantically equivalent implicit invocation program; we reason about it using existing techniques; we then map the results from the II space back to the AOP space. In our earlier work [18], we showed that the implicit invocation space can be mapped to the aspect space. (In particular, support for instance-level aspects and first class aspect instances enables a mapping of aspect programs to mediator-based design structures [21][23], which use implicit invocation extensively to separate integration concerns.) In this work, we make the reduction concrete and present preliminary evidence supporting our hypothesis.

In the rest of this paper, we will be using an example system from our previous work [18], [22] to illustrate various approaches. The example is extremely simple, but it is known to capture essential issues in a way that scales up. Our example system consists of two objects *b1* and *b2*, instances of the *Bit* type. A *Bit* can be Set and Cleared by *Set* and *Clear* and its current state can be read by the *Get* method. In our example system *b1* and *b2* are required to work together as follows: if any client *Sets* (respectively *Clears*) either *Bit*, the other must be *Set* (*Cleared*). In other words, the behaviors of the *Bits* have to be integrated by a behavioral relationship, which we will call *Equality*, which maintains a bit-equality constraint.

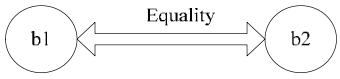

Figure 1. A simple example: Bit

The rest of this paper is organized as follows. Section 2 describes aspect oriented programming and the challenges in reasoning about it. Section 3 describes the effort in reasoning about implicit invocation space. Section 4 describes our approach. Section 5 presents related work. Section 6 concludes.

2. ASPECT-ORIENTED PROGRAMMING

Aspect-oriented programming constructs are meant to enable the modular representation of otherwise scattered and tangled code. The key mechanisms of aspect-oriented programming in the tradition of AspectJ [2] are join points, pointcut expressions, advice code, and aspect modules. A *join point* is a point in the execution of a program (such as just before a method body executes) exposed by the language design for behavioral modification by aspect modules. The join points exposed by current AspectJ-like languages include method calls and execution, field get and set operations, exceptions, and object initialization. A *pointcut* is an expression—a predicate—that serves to select a subset of program join points. *Advice* is code that is effectively to be executed at each join point selected by a pointcut. An *aspect* is a module that aggregates pointcut expressions and associated advice code along with other information typically found in class definitions. Weaving is the process by which advice code is composed with the base program code at selected join points to yield an executable.

Eos [13] [18] is an AspectJ-like extension of C# [17] that supports first-class aspect instances and instance-level advising. By first class aspect instance we mean that the aspects are class-like constructs that can be instantiated, passed as arguments, returned as value, etc. By instance-level advising we mean ability to select specific *instances* of a type that will be affected by aspect advice. (In AspectJ, aspects advise types and thus all instances.) The code for the Bit example in Eos is as follows:

```
1    public class Bit {
2      bool value;
3      public Bit() { value = false; }
4      public void Set() { value = true; }
5      public bool Get (){ return value; }
6      public void Clear() {value= false; }
7    }
```

The following code implements Equality as an instance-level aspect:

```
1  public instancelevel aspect Equality  {
2    Bit b1, b2;
3    bool busy;
4    public Equality(Bit b1, Bit b2)   {
5              addObject(b1);  addObject(b2);
6              this.b1 = b1;    this.b2 = b2;
7              busy = false;
8   }
9   after():execution(public void Bit.Set ())   {
10       if(!busy) {
11             busy = true;
12             Bit m = (Bit) thisJoinPoint.getTarget();
13             if(b == m1)b2.Set(); else b1.Set();
14             busy = false;
15             }
16   }
17   after():execution(public void Bit.Clear ())   {
18       if(!busy) {
19             busy = true;
20             Bit b = (Bit) thisJoinPoint.getTarget();
21             if(b == b1)b2.Set(); else b1.Set();
22             busy = false;
23             }
24   }
25  }
```

Figure 2. Eos code for Bit example

The purpose of the aspect is to ensure that b1 and b2 always have the same state at quiescent points (i.e., except during execution of a Set operation). We thus need to verify that the aspect module behaves in such a way. It is, however, generally difficult to reason about AOP for the following reasons:

1. The primitive constructs in aspect-oriented languages need to be rigorously defined.
2. It could be very hard to reason about an aspect program automatically. There has been a fair amount of research on the possibility of applying model checking on reasoning about AOP, although there is hardly a working example.
3. Since the behavioral modifications by aspects can cut across the entire code base, it's very hard for us to understand an aspect-oriented program in a modular way. That is, we can no longer analyze modules separately then combine results. An aspect can influence the semantics of the whole system. This issue as the most difficult part of reasoning about AOP.

3. IMPLICIT INVOCATION

Implicit invocation [23] [12] is a mechanism for managing how *invocation* relations are represented as *names* relations. If component *A* needs to *invoke* component *B* at a certain point, *A* can do so either by explicitly calling *B*, in which case *A names* B, or *B* can *register* with *A* to be invoked implicitly by event announcement, which case, *B* names *A*. Because the *names* relation is a key determinant of compile, link, and runtime dependencies, having means to structure it properly is important. Implicit invocation and join points and advice provide such means.

II is also known as publish-subscribe system, since generally it is implemented in such a way. A component (the subscriber) registers interest in particular events that the other component (the publisher) announces. The II mechanism then guarantees the invocation of subscriber. The publisher is not aware of the existence of the subscribers. II has been used widely in system-level development and message-passing applications. For example, a user can define and register a callback procedure that is invoked when a particular signal is raised by the OS kernel.

Such systems make modular reasoning harder, since we need to decouple the verification of one component from the verification of the rest of system that communicate with the given component by event bindings. Dingel et al. [8] proposed a formal model for II systems and proposed a three-phase reasoning methodology: decomposition, local reasoning and global reasoning. By the decomposition process, we can formalize the event/handler semantics and model the system, the environment and the event dispatch mechanism in a modular way. Applying the three-phase reasoning then can be expected to achieve the effect of modular reasoning about the whole system.

It is, however, often not easy to decompose the system into separate groups and prove their independence. An alternative approach proposed by Garlan et al. [11] uses model checking

instead of formal modular reasoning. Application of model checking to software encounters two problems. First, an appropriate state model for the system being checked needs to be created. This state model of a reasonable size system has a huge state space. To check this huge state space using a model checker is time consuming, if not infeasible. Second problem is thus to find the means to reduce this state space into manageable size so that it can be supplied as an input to the model checker.

The architecture of an II system in [11] models the following features:

- Components: functional objects with well-defined interfaces
- Events: the primary communication method between components
- Event-Method Bindings: the correspondence between announced events and the methods that are invoked in response as event handlers
- Event Delivery Policy: rules about event announcement and delivery
- Shared State: another communication method between elements of the II system
- Concurrency Model: determines if the system has a single thread or multiple threads of control

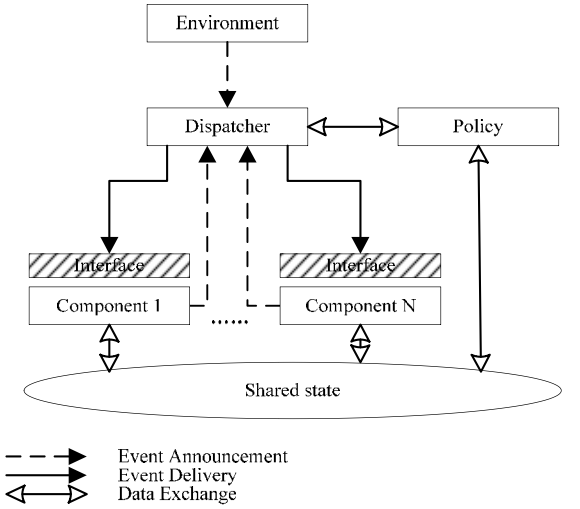

Figure 3. II system architecture

The system structure is in Figure 3. Besides those functional components, the dispatcher and policy modules are another important part of the approach. They are responsible for event binding and dispatching. Also the environment represents external elements that could affect the system.

The run time state model of an II system has to model the following in addition:

- Event announcement by the system components
- Storage of event announcements before dispatching
- Event delivery to the system components
- Invocation of methods bound to the delivered events
- Invocation acknowledgement

For the *Bit* example, *b1* and *b2* are considered separate components. Calling the state change methods *Set*/*Clear* on the component *b1* results in the component announcing an event representing the change in its state, which is then captured by the dispatcher. The dispatcher will consult the policy module to determine what event will be delivered to which component without causing a propagation cycle. The state change in b2 will also result in announcement of an event representing its state change and the same actions as above. Figure 4 depicts the simplest II state model that models the *Bit* example, in which we omit the environment module and the details of b2's event exchange since it's the same with b1. For each event of interest, a notify message is delivered from the component to the dispatcher, which results in the delivery of an invoke message from the dispatcher to the other component.

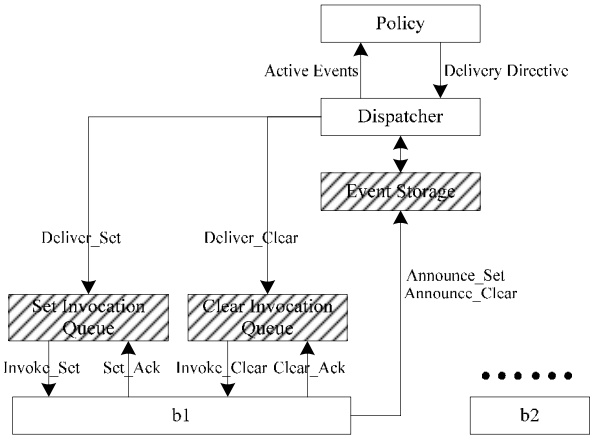

Figure 4. II run time state model for the Bit example

To reason about the system, we are interested in verifying some properties of the system. These properties are expressed as assertions. The *Equality* between *b1* and *b2* will be represented as the following assertion:

Equality:
assert(F (b1.state = b2.state));

"F", a logical notation of LTL, represents "eventually". The SMV model checker takes these assertions and the state machine constructed before as input and produces validity of these assertions as output.

Bradbury et al. [4] extended Garlan et al.'s model to support dynamic event model. They use XML to represent the event model which is later translated into SMV input. They have applied their method on real world implicit invocation systems such as Active Badge Location System (ABLS) [26] and Unmanned Vehicle Control System (UVCS) [20] demonstrating the capabilities of their approach.

4. Reduction from AOP to II

We would like to assure that the aspect performs its intended behavioral modifications without producing any undesirable side effect. Existing approaches can be used to reason about the plain object-oriented systems. Our approach therefore focuses on reasoning about aspects, the modular representation of crosscutting concerns, and its interaction with the component part. To enable this reasoning, we propose to reduce an element of aspect-oriented programming space (an aspect-oriented program)

to an element in implicit invocation space (an implicit invocation based program). We can then reason about the element in the implicit invocation space using the II reasoning techniques described in the last section. The reasoning results will then be reduced back from II space to the AOP space thus effectively enabling reasoning about the aspect-oriented program.

First, let us revisit the concepts of aspect-oriented programming as embodied in asymmetric languages such as AspectJ. A *join point* is a point in the execution of a program (such as just before a method body executes) exposed by the language design for behavioral modification by aspect modules. The join points exposed by current AspectJ-like languages include method calls and execution, field get and set operations, exceptions, and object initialization. A *pointcut* is an expression—a predicate—that serves to select a subset of program join points. A *pointcut* is then defined to be a predicate expression over a set of join points.

We observe that every join point can be viewed as a set of semantic events. These events will be announced when the control hits the join point during program execution. AspectJ-like languages can advise a join point in three different ways: *before*, *after* and *around*. The before and after advice are semantically clean, however, the around advice is a bit more involved. To keep the model simple we will not be discussing around advices. To differentiate between these ways of advising we map each join point to a 2-tuple of events: <before the join point, after the join point>. We treat each of these elements differently in the corresponding event model. An advice before a join point is mapped to the event "before a join point" and similarly for after advice.

A pointcut selects a subset of program join points. Each pointcut is mapped to an enumeration of a set of 2-tuple of events where each 2-tuple in the set corresponds to a join point in the subset of program join points selected by the pointcut. A named pointcut is mapped to a named set of 2-tuples. The subset of join points that are matched by the pointcut expressions that rely on run-time information cannot be obtained statically. The control flow (*cflow*) and control flow below (*cflowbelow*) are examples of some of these pointcuts. These pointcuts cannot be mapped statically to a simple event model like the one used by Garlan [11]. In summary, the mapping of join points and pointcuts to II concepts can be shown below:

f: {Joinpoint} -> {Event} X {Event}

g: {Pointcut} -> {Predicate} X {A set of events}

A pointcut can be denoted by a pair *<predicate, {joinpoint}>*, which means a predicate expression over a set of joinpoints.

For example, the mapping applied to a joinpoint *a* is:

f(a) = <after_a, before_a>

in which *after_a* and *before_a* are two events.

The mapping applied to a pointcut <p, {a, b, c}> is:

g(<p, {a, b, c}>) = <g(p), {f(a), f(b), f(c)}>

in which *p* is a predicate over the set of joinpoints *a, b, c*, while *g(p)* denotes the mapped predicate over the set of events.

Events picked out for our Bit example are shown below:

Events exposed by the Bit component =
{ [before_Bit.Bit, after_Bit.Bit],
[before_Bit.Set, after_Bit.Set],
[before_Bit.Clear, after_Bit.Clear] }

Events picked out by the pointcut expression "execution(public void Bit.Set ())" = {[before_Bit.Set, after_Bit.Set]}

Events picked out by the pointcut expression "execution(public void Bit.Clear ())" = {[before_Bit.Clear, after_Bit.Clear]}

The mapping of pointcut *after(): execution(Bit.Set())* is:

g(<after execution, {Bit.Set()}> =

<only after_* events, {<before_Bit.Set, after_Bit.Set>}>

The mapping of pointcut *after(): execution(Bit.Clear())* is:

g(<after execution, {Bit.Clear()}>) =

<only after_* events, {<before_Bit.Clear, after_Bit.Clear>}>

An advice is mapped to an event handler. The role of an advice, with respect to the advised pointcut, is the same as the event handler with respect to the captured event. In the Bit example, there are two event handlers in the *Equality* aspect, one corresponds to the after():execution(Bit.Set()) advice, the other corresponds to the after():execution(Bit.Clear()) advice.

Third, there should be a dispatcher in the mapped system, as well as a dispatch policy module. The dispatcher is responsible for event storage, event binding, event delivery and interacting with the dispatch policy module. The policy module implements event delivery policy. In the context of mapping AOP, it should be able to choose event handlers according to a predicate expression over a set of events, just like the pointcut definition.

As for the Bit example, the aspect *Equality* actually can be mapped to part of the policy module in Figure 3. When the dispatcher receives an event, it will inquire the policy module to decide the actions it will take. In this case, the *Equality* policy will determine which message (Set or Clear) to deliver to which component.

The assertions we can check over this II system like *assert(F (b1.state = b2.state))* is now mapped back to the behavior constraint between the two objects b1 and b2 in the Eos program (Figure 2). This constraint is therefore checked by the reduction process. Thus, we demonstrate a simple example of our approach to use II reasoning technique to reason about an aspect-oriented program's behavior.

5. RELATED WORK

Dingel et al. [8], Garlan et al. [11], and Bradbury et al.'s [4] work on model checking implicit invocation systems is closely related to ours—and is, in fact, used as a subroutine. They proposed an event model to describe the behavior of the II systems. Bradbury et al.'s approach translates this model written in XML format to the SMV language and applies the SMV model checker. These approaches are applicable to II systems, but not directly to aspect-oriented programs. Our approach supplements these approaches by providing a reduction from the AOP space to II space, thus enabling the use of these approaches in the AOP space.

Mapping AOP to event model is not a completely new idea. Filman and Havelund [10] briefly proposed an event language for aspects. The event language has primitive events and a set of relationships between events, which include abstracted temporal relationships, abstract temporal quantifiers, concrete temporal relationship referring to clock time, cardinality relationships and aggregation relationships for describing sets of events. Walker and Murphy [25] employed their implicit context concept to map join points to ordered events. By such mapping, they showed a close relationship between AOP and implicit context. Our work makes the reduction from AOP to II explicit.

As for reasoning about AOP, there has been significant research on this topic. Ubayashi et al. [24] claimed to apply model checking using aspects. They write an aspect for every property to check, and then weave these aspects and the source program into a new program and then execute this weaved program. This approach works only for plain java programs. It can only check properties that can be represented by aspects. It also uses a dynamic approach, so presence of property violation can only be discovered if that execution path is taken.

Blair and Monga [3] view every pointcut declaration as a *slicing criterion* that can be used to compute an associated slice. They then envision that this sliced program could be fed into Bandera model checker, but the expressiveness of aspects is difficult to be captured by any slicing technique.

Instead of reasoning about the entire program, Clifton and Leavens [5][6] give two concepts for AspectJ: Observer (Spectator) and Assistant. Assistants are aspects that could change the behavior of other parts, while observers do not. They also propose an *accept* notation to be added into AspectJ, to make aspect invocation explicit, for facilitating modular reasoning. By categorizing aspects into observers and assistants, and explicitly exposing the join point, they expect to be able to reason AOP in a modular way, however, it remains unclear how can we differentiate assistants from observers in real programs. The *accept* notation compromises the obliviousness [9] properties of aspect-oriented programs. Our approach on the other hand, does not impose any restriction on the language model of aspect-oriented programming languages.

Devereux [7] tries to transfer aspect programs to alternating-time logic. Then program properties can be expressed by assertions in alternating-time logic. It supports two concepts, imposition and preservation similar to assistant and observer. The development of a reduction similar to ours from aspect-oriented space to alternative-time logic is possible; however, the lack of tool support for automated reasoning in alternating-time logic makes the reduction less attractive.

Recently there has been increasing research interests on exploiting type systems to enable reasoning about aspect-oriented programs. Aldrich [1] presented a simple aspect language called TinyAspect. Module sealing and explicit declaration of exported join points is the core of TinyAspect. The idea is to enforce abstraction by prohibiting clients, viz., aspects, from exploiting implementation details, such as calls from within a component to its own public methods. There is a set of type inference rules for TinyAspect by which one can reason about the behavior of aspects. Type checking in the TinyAspect model, however, does not allow one to reason about the kinds of behavioral properties that we address.

6. CONCLUSION

Aspect-oriented programming imposes many new challenges on program understanding and reasoning. In fact, how to reason about AOP in a modular way has been an open question for years. In this paper, we reduce the join point and pointcut mechanisms of AOP to the events of implicit invocation systems, and we show that this reduction has the potential to improve our ability to reason formally about the aspect program behavior. Forthcoming work will formalize the reduction, develop and evaluate the approach, and investigate the possibility of automated tool support for such reductions and formal property verifications.

7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant ITR-0086003.

8. REFERENCES

[1] Aldrich, J., “A Typed, Modular Foundation for Aspect-Oriented Programming”, 2003.

[2] AspectJ Homepage: http://www.eclipse.org/aspectj.

[3] Blair, L., Monga, M. “Reasoning on AspectJ Programmes”, GI-AOSDG 2003 Essen, Germany

[4] Bradbury, J. S., Dingel, J., “Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems,” In Proc. of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), Helsinki, Finland, Sept. 2003.

[5] Clifton, C., and Leavens, G. T., “Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning.”, Technical Report TR#02-04, Department of Computer Science, Iowa State University, March 2002.

[6] Clifton, C., and Leavens, G. T., “Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy.”, Technical Report TR#03-01A, Department of Computer Science, Iowa State University, March 2002.

[7] Devereux, B., “Compositional Reasoning About Aspects Using Alternating-time Logic”, FOAL2003

[8] Dingel, J., Garlan, D., Jha, S., Notkin, D., “Reasoning about implicit invocation”, Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, November 1998

[9] Filman R., and Friedman, D., “Aspect-oriented programming is quantification and obliviousness”, In Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[10] Filman, R.E., Havelund, K.. “Realizing Aspects by Transforming for Events.” In Automated Software Engineering 2002 (ASE'02). Edinburgh, Scottland, 23-27 September 2002. IEEE Computer Society

[11] Garlan, D., Khersonsky, S., and Kim, J. S., ”Model Checking Publish-Subscribe Systems”, Proceedings of The 10th International SPIN Workshop on Model Checking of Software (SPIN 03), Portland, Oregon, May 2003.

[12] Garlan, D., and Notkin, D., “Formalizing Design Spaces: Implicit Invocation Mechanisms”. *VDM '91: Formal Software Development Methods*, pp. 31--44 (October 1991).

[13] Eos Homepage: http://www.cs.virginia.edu/~eos

[14] Java: http://java.sun.com

[15] Kiczales, G.., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., “Aspect-oriented programming,” in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlang, Lecture Notes on Computer Science 1241, June 1997.

[16] Masuhara, H., and Kiczales G., “Modular Crosscutting in Aspect-Oriented Mechanisms”, ECOOP 2003, Darmstadt, Germany, July 2003.

[17] Microsoft. C# Specification Homepage. http://msdn.microsoft.com/net/ecma

[18] Rajan, H. and Sullivan, K., “Eos: Instance-Level Aspects for Integrated System Design”, *2003 Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 03)*, (Helsinki, Finland, Sept 2003).

[19] Sihman, M. and Katz, S.. “Model Checking Applications of Aspects and Superimpositions”, FOAL 2003

[20] Stuurman, S., and Katwijk, J.van, “On-line change mechanisms: the software architectural level”, In Proc. Of the ACM SIGSOFT FSE, pages 80-86, Nov. 1998

[21] Sullivan, K., “Mediators: Easing the Design and Evolution of Integrated Systems”, Ph.D. dissertation, University of Washington, 1994.

[22] Sullivan, K., Gu, L., Cai, Y., “Non-modularity in Aspect-Oriented Languages: Integration as a crosscutting concern for AspectJ,” Proceedings of Aspect-Oriented Software Design, 2002

[23] Sullivan, K. and Notkin, D., “Reconciling environment integration and software evolution,” ACM Transactions on Software Engineering and Methodology 1, 3, July 1992, pp. 229–268 (short form: Proceedings of the 4th SIGSOFT Symposium on Software Development Environments, 1990, pp. 22–33).

[24] Ubayashi, N., Tamai, T., “Aspect-oriented programming with model checking”, Proceedings of the 1st international conference on Aspect-oriented software development, April 22-26, 2002, Enschede, The Netherlands

[25] Walker, R. J. and Murphy, G. C., “Joinpoints as ordered events: towards applying implicit context to aspect-orientation”, Workshop on Advanced Separation of Concerns at the 23nd ICSE, 2001.

[26] Want, R., Hopper, A., Falcao, V., and Gibbons, J., “The active badge location system.” ACM Trans. on software engineering and methodology, 10(1):91-102, Jan. 1992

On the Horizontal Dimension of Software Architecture in Formal Specifications of Reactive Systems

Mika Katara
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
mika.katara@tut.fi

Reino Kurki-Suonio
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
reino.kurki-suonio@tut.fi

Tommi Mikkonen
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
tommi.mikkonen@tut.fi

ABSTRACT

In order to provide better alignment between conceptual requirements and aspect-oriented implementations, formal specification methods should enable the encapsulation of *logical abstractions* of systems. In this paper we argue that *horizontal architectures*, consisting of such logical abstractions, can provide better separation of concerns over conventional ones while supporting incremental development for more common units of modularity such as classes. We base our arguments on our experiences with the DisCo method, where logical abstractions are composed using the *superposition* principle.

1. INTRODUCTION

Post-object programming (POP) mechanisms, like those developed in aspect-oriented programming [6], provide means to modularize crosscutting concerns, which are in some sense orthogonal to conventional modularity. The background of this paper is in the observation that the same goal has been pursued also at the level of formal specifications of reactive systems, and that the results of this research are relevant for the theoretical understanding of POP-related architectures and of the associated specification and design methods.

Unlike conventional software modules, units of modularity that are suited for a structured description of the intended logical meaning of a system can be understood as aspects in the sense of aspect-oriented programming. We call such units *horizontal* in contrast to conventional *vertical* units of modularity, such as classes and processes. While the vertical dimension remains dominant because of the available implementation techniques, the horizontal dimension can provide better separation of concerns over the vertical one improving, for example, traceability of requirements.

In this paper, our experiences with the DisCo method are used as the basis for discussion. The rest of the paper is structured as follows. First, in Section 2, we present the idea of structuring specifications using horizontal units capturing logical rather than structural abstractions of the system. In Section 3 the DisCo method is presented which utilizes such components as primary units of modularity. Section 4 concludes the paper by discussing the approach in the light of related work.

2. TWO DIMENSIONS OF SOFTWARE ARCHITECTURE

Describing an architecture means construction of an abstract *model* that exhibits certain kinds of intended properties. In the following we consider *operational* models, which formalize executions as state sequences, as illustrated in Figure 1, where all variables in the model have unique values in each state s_i. In algorithmic models these state sequences are finite, whereas in reactive models they are nonterminating, in general. Message sequence charts are a well-known operational formalism for describing state sequences where states s_i consist only of the control points of the communicating processes.

2.1 Vertical Units

The algorithmic meaning of software, as formalized by Dijkstra [4], has the desirable property that it can be composed in a natural manner from the meanings of the components in a conventional architecture. To see what this means in terms of executions in operational models, consider state sequences that implement a required predicate transformation. Independently of the design principles applied, a conventional architecture imposes a "vertical" slicing on these sequences, so that each unit is responsible for certain *subsequences* of states. This is illustrated in Figure 2, where the satisfaction of the precondition-postcondition pair (P, Q) for the whole sequence relies on the assumption that a subsequence V, generated by an architectural unit, satisfies its precondition-postcondition pair (P_V, Q_V).

More generally, an architecture that consists of conventional units imposes a nested structure of such vertical slices on each state sequence. In the generation of these sequences, the two basic operations between architectural units can

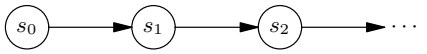

Figure 1: Execution as a state sequence.

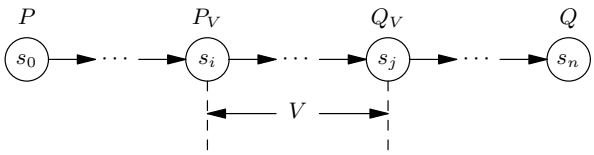

Figure 2: A vertical slice V in an execution.

be characterized as *sequential composition* and *invocation*. The former concatenates state sequences generated by component units; the latter embeds in longer sequences some state sequences that are generated by a component unit. In both cases, the resulting state sequences have *subsequences* for which the components are responsible. In current software engineering approaches, this view has been adopted as the basis for designing behaviors of object-oriented systems, leading the focus to interface operations that are to be invoked, and to the associated local precondition-postcondition pairs.

The architectural dimension represented by this kind of modularity will be called *vertical* in the following.

2.2 Horizontal Units

The meaning of a system can also be modeled by how the values of its variables, denoted by set X, behave in nonterminating state sequences. In order to have modularity that is natural for such a reactive meaning, the meanings of the components must be of the same form. In other words, each component must also generate nonterminating state sequences, but the associated set of variables can be a subset of X. An architecture of reactive units therefore imposes a "horizontal" slicing of state sequences, so that each unit is responsible for some subset X_H of variables in all states s_i, as illustrated in Figure 3.

In the generation of state sequences, only one basic operation is needed. *Superposition* uses state sequences that are generated by a horizontal slice embedding them in sequences that involve a larger set of variables. The state sequences of the resulting vertical architecture have *projections* for which the horizontal components are responsible. Properties of horizontal slices then emphasize collaboration between different vertical units, and the relationships between their internal states.

The two dimensions of architecture are in some sense dual to each other. On the one hand, from the viewpoint of vertical architecture, the behaviors generated by horizontal units represent crosscutting concerns. From the horizontal viewpoint, on the other hand, vertical units emerge incrementally.

2.3 Architecting Horizontal Abstractions

To illustrate the nature of horizontal units, consider a simple modeling example of an idealized doctors' office, where

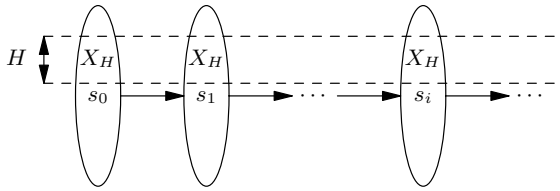

Figure 3: A horizontal slice H in an execution.

ill patients are healed by doctors.[1] The natural vertical units in such a model would include patients, doctors and receptionists. Horizontal units, on the other hand, would model their cooperation as specific projections of the total system, and the whole specification could be built incrementally from these.

The specification process can start with a trivial model of the simple aspect that people get ill, and ill patients eventually get well. The "illness bits" of the patients are the only variables that are needed in this horizontal unit. Next, this unit can be embedded in a larger model where a patient gets well only when healed by a doctor. This extended model has events where a doctor starts inspecting a patient, and participation of a doctor is also added to the events where a patient gets well. Finally, a further superposition step can add the aspect that also receptionists are needed in the model, to organize patients to meet doctors, and to make sure that they pay their bills. This aspect is truly crosscutting in the sense that it affects all the vertical units, i.e., patients, doctors and receptionists.

Each unit in this kind of a horizontal architecture is an *abstraction of the meaning* of the total system. The first horizontal unit in this example is an abstraction where all other behavioral properties have been abstracted away except those that concern the "illness bits" of patients. In terms of Temporal Logic of Actions (TLA) [18], (the meaning of) the total system always implies (the meaning of) each horizontal unit in it. As for variables, each component in the horizontal structure focuses on some variables that will become "secrets" encapsulated in the vertical components in an eventual implementation. This can be related with the observation of [19], where such secrets are considered more important than the interfaces associated with them in early phases of design.

This gives a formal basis for specifying a reactive system – i.e., for expressing its intended meaning – incrementally in terms of operational abstractions that can be formally reasoned about. Since it is unrealistic to formulate any complex specification in one piece, this is a major advantage for using horizontal architectures in the specification process. A classical example of using horizontal slices is the separation of correctness and termination detection in a distributed computation [5]. This is also the earliest known use of superposition in the literature – its close relationship with aspect-orientation was first reported in [13].

For comparison, consider how stepwise refinement proceeds

[1]This is an outline of a simplified version of an example that was used to illustrate the ideas of DisCo in [16].

with vertical architectural units. In terms of executions, each operational abstraction generates state sequences that lead from an initial state to a final state, so that the required precondition-postcondition pairs are satisfied. At the highest level of abstraction there may be only one state change, and each refinement step replaces some state changes by state sequences that are generated by more refined architectural units. This leads to a design where the early focus is on interfaces and their use, whereas the "secrets" inside the vertical components may become available only towards the end of the design, when the level of implementable interface operations is achieved.

Due to the above, the abstractions that a vertical architecture provides are not abstractions of the meaning: the complete meaning is assumed to be available already at the highest level, and it remains the same throughout the design process. Instead, at each level of refinement, a vertical architecture gives an *abstraction of the structure* of an implementable operational model.

3. EXPERIENCES WITH DISCO

The above views have been stimulated by the experiences gained with the DisCo[2] method [10, 23]. DisCo is a formal specification method for reactive system, whose semantics are in TLA [18].

3.1 Horizontal Architectures in DisCo

In DisCo, the horizontal dimension, as discussed above, is used as the primary dimension for modularity. The internal structure of horizontal units consists of partial classes that reflect the vertical dimension. For instance, each of the attributes of a class can be introduced in different horizontal units.

Behavioral modeling is DisCo's bread and butter. The design usually advances so that first the high-level behaviors are included in the model. Based on this abstract behavioral model it is then possible to include more details, even to the level where direct mapping to available implementation techniques becomes an option [17].

In more detail, horizontal components correspond to superposition steps referred to as *layers*. Formally, each layer is a mapping from a more abstract vertical architecture to a more detailed one. As the design decisions are encapsulated inside the layers, they become first-class design elements. Because layers represent logical, rather than structural aspects of the system, they serve in capturing concepts of the problem domain.

Ideally, each layer contains only those details that pertain to a particular logical aspect. Thus, better alignment between the requirements and design can be achieved. In cases where a layered structure is aligned with an aspect-oriented implementation [1], this results in improved traceability concerning the entire design flow.

Dependencies between layers arise if, for instance, a class is defined in one layer and its attribute in another layer. In this case the latter is said to depend on the former. These dependencies impose a partial order between the layers of the specification.

[2]Acronym for Distributed Co-operation.

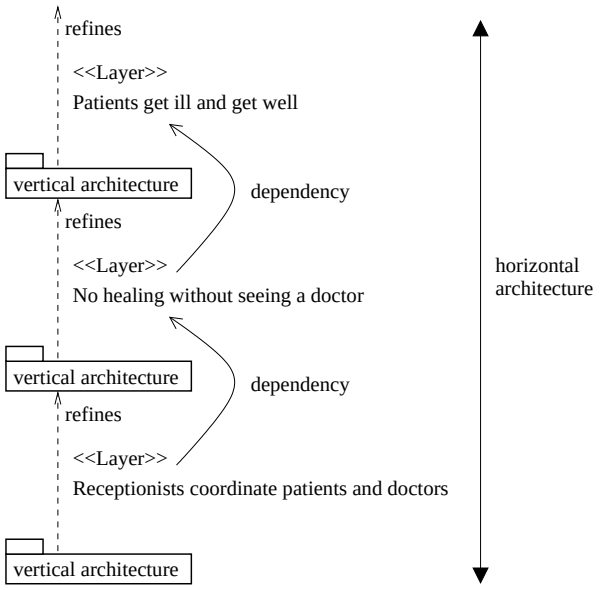

Figure 4: Horizontal architecture consisting of layers.

In Figure 4 the situation is illustrated in the simple case of the example outlined above. The horizontal architecture consists of three layers. Each layer refines a vertical architecture to a more detailed one by adding a new piece of "meaning" to the system. The layer introducing doctors depends on the one introducing patients, and the one introducing receptionists depends on the one introducing doctors (and transitively on the patients layer).

Different concerns can be partially treated in common layers. A situation of this kind arises when a common design decision is made to cater for two different features, for instance. This means that, generally, concerns are treated in one or more layers. Moreover, there can be overlap between the sets of layers treating different concerns.

3.2 Example: Mobile Robot

As a more concrete example, a simplified DisCo specification of a mobile robot (toy car) control software specification is presented. We have omitted the parts dealing with real time (and fairness) which can be found in [11] in a slightly different form.

The mobile robot is a small microcontroller-based car. The objective is to keep the car on a track marked by optical tape. From the viewpoint of the control software the system has two inputs and two outputs. The inputs are readings from an A/D converter connected to infra-red censors, and from an odometer. The outputs are two servo motors controlling the steering and the movement. The servos are driven by PWM (Pulse Width Modulation) signals. There

is also a switch, which is used to start the car and to stop it.

There are two concern that need to be addressed: the basic functionality of the car including starting and stopping and the control part including the control algorithms. These concerns are treated in three separate layers, one of which is common to both concerns, i.e. the concerns are overlapping. The details of the layers are described next.

3.2.1 Basic Actions

Layer Basic_Actions contains the basic functionality of the system. It introduces two *classes* and three *multi-object actions*. Actions are symmetric with respect to participants; there are no callers nor callees. Class Data holds internal *variables* and class Output variables that model the outputs. The variables r_dist and r_tape of type real represent the distance covered between the last two readings of the odometer and the location of the car relative to the tape, respectively. Variables c_engine and c_steer model the current lengths of the servo pulses. If both equal zero, the car is stationary with its wheels straight. In a class definition, the number of instances is indicated by placing the number in parentheses after a class name. The classes are shown below:

```
class Data (1) is
        r_dist: real := 0.0; r_tape: real := 0.0;
end Data;

class Output (1) is
        c_engine: real := 0.0; c_steer: real := 0.0;
end Output;
```

Action Clear clears all the variables given in this layer. This is implemented by a parallel assignment in its *body*. Action Read, which has a *participant* of class Data, models the reading of the odometer and the A/D converter. The actual new readings are modeled by two *parameters* r_x and r_y, which have nondeterministic values and do not refer to any objects. In action Control, parameters c_x and c_y are used to model the new values given by the control algorithm. They are assigned to the variables c_engine and c_steer, respectively. The actions are given below:

```
action Clear (D: Data; O: Output) is
when true do
        D.r_dist := 0.0 || D.r_tape := 0.0 ||
        O.c_engine := 0.0 || O.c_steer := 0.0;
end Clear;

action Read (r_x, r_y: real; D: Data) is
when true do
        D.r_dist := r_x || D.r_tape := r_y;
end Read;

action Control (c_x, c_y: real; O: Output) is
when true do
        O.c_engine := c_x || O.c_steer := c_y:
end Control;
```

Because the *guards* of all three actions are identically true, the actions are continually *enabled*. The behavior of the system consists of clearing the variables, reading the inputs and writing the outputs. The order in which these actions are executed is nondeterministic.

3.2.2 Drive States

Layer Drive_States introduces the start/stop switch and specifies the order in which the actions are executed. Class Data is *extended* to hold a state machine d_state, which indicates the actions that are allowed to be executed. The state machine has states start, read and control, the first of which is the default state.

The switch is modeled by variable switch, which has two states, on and off. The state of the switch is changed in action Toggle. When the switch is on in state start, action Start is enabled. It changes the state to read.

The extensions of class Data and the new actions are shown below:

```
extend Data by
        d_state: (start, read, control);
        switch: (off, on);
end Data;

action Toggle (D: Data) is
when true do
        if D.switch'off then
                D.switch → on();
        else
                D.switch → off();
        end if;
end Toggle;

action Start (D: Data) is
when D.switch'on and D.d_state'start do
        D.d_state → read();
end Start;
```

The car is fully operational when the switch is on in states read and control. Likewise, actions Read and Control are refined so that they are enabled correspondingly. Furthermore, by addition of state transition statements D.d_state → control() and D.d_state → read() to Read and Control, respectively, they are executed by turns. The *refined* action Read is given below, where ellipses denote the guard and the body of the original action:

```
refined Read (r_x, r_y : real; D: Data) is
        when ... D.switch'on and D.d_state'read do
        ...
        D.d_state → control();
end Read;
```

Furthermore, action Clear is refined to change the state back to start when the switch is turned off. In this case it implicitly stops the engine and straightens the wheels by clearing all the variables.

3.2.3 Control Algorithms

Layer Control_Algorithms treats the controlling of the movement and the steering. The layer defines ten constants, extends class Data, introduces two functions and refines all three actions given in Basic_Actions. The constants and the variables are added due to the control algorithms. Variable e_state represents the state of the engine. The three states power_up, moves and normal are needed since, because of friction, it is necessary to power up until movement is sensed and after that power down slightly to prevent slipping.

The P and PID algorithms are used to compute new values for the engine and steering, respectively. Function PID is shown below (s_P, s_I and s_D are constants):

```
function PID(D: Data) : real is
        return –s_P*D.r_tape + s_I*D.r_tape_ma
                + s_D*(D.r_tape_old – D.r_tape);
end PID;
```

Action Read is refined to include the statements needed by the control algorithm (comments begin with double hyphens):

```
refined Read (r_x, r_y: real; D: Data) is
when ... do             - - the guard is unchanged
        ...
        if (r_x = 0.0) and (D.r_dist = 0.0) then
                D.e_state → power_up();      - - not moving yet
        elsif (r_x > 0.0) and (D.r_dist = 0.0) then
                D.e_state → moves();
        else
                D.e_state → normal();
        end if ||

        - - moving average:
        D.r_tape_ma := ((n – 1)*D.r_tape_ma – D.r_tape)/n ||
        D.r_tape_old := D.r_tape;
end Read;
```

In the guard of action Control, parameters c_x and c_y are bound to the return values of functions P and PID, respectively. If the car has lost the track, it should stop. This is the situation if the absolute value of r_tape is greater than limit, which is treated as a special case in the guard of action Control. The refined action Control is shown below:

```
refined Control (c_x, c_y: real; O: Output; D: Data)
of Control(c_x,c_y,O) is
when ... (c_x = if((abs D.r_tape) > limit)
                then 0.0 else P(D,O) end if)
                and c_y = PID(D) do
        ...
end Control;
```

Furthermore, action Clear is refined to clear variables introduced in this layer.

As already mentioned, the vertical units emerge incrementally while specifying the horizontal layers. In the composed specification, class Data, for instance, consist of all variables added in different layers:

```
class Data (1) is
        - - Basic_Actions:
        r_dist: real := 0.0;
        r_tape: real := 0.0;
        - - Drive_States:
        d_state: (start, read, control);
        switch: (off, on);
        - - Control_Algorithms:
        e_state: (power_up, moves, normal);
        d.r_tape_ma: real := 0.0;
        d.r_tape_old: real := 0.0;
end Data;
```

Obviously, it would have been difficult to come up with such variables directly.

The horizontal architecture of the specification is illustrated in Figure 5. Layer Basic_Actions is common to both concerns, and layers Drive_States and Control_Algorithms treat the functional and control concerns, respectively. The latter layers can be applied in any order on top the former when composing the specification as long as the dependencies are respected, i.e. either Control_Algorithms on top of Drive_States or the other way around.

3.3 Specifying with DisCo

As used in DisCo, superposition is strictly additive, i.e. nothing is removed from the refined specification. This means that all assignments to a variable must be given in the layer introducing the variable. With this restriction, safety properties can be preserved by construction.

Concerning support for specifying non-trivial systems, the DisCo toolset [2] provides animated simulation of DisCo

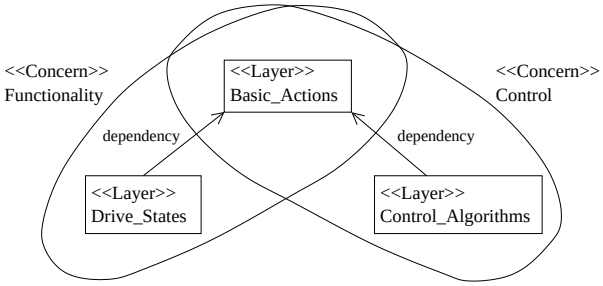

Figure 5: Architecture of the mobile robot specification.

specifications for validation purposes at all levels of refinement. Moreover, assembly of horizontal architectures can be supported by reusable layers as described in [15]. These layers can be seen as behavioral templates that should be formally verified [14].

Related to common software engineering practices, the advantages of better alignment between requirements and specifications are emphasized in the maintenance phase [3]. It should be also noted, that the ideas underlying the approach are insensitive to the notation used, thus offering a foundation for aspect-oriented specification languages. In [12] this was shown using UML.

Model-Driven Architecture (MDA) [8, 7] by Object Management Group (OMG) also includes similar elements to the DisCo approach. In MDA, however, only three pre-defined levels are used, including *computation independent*, *platform independent*, and *platform specific* models. By allowing individual levels of abstraction of MDA to consist of several DisCo layers, one can create a development approach that fits in the guidelines of MDA without compromising rigorousness [20].

4. DISCUSSION

We have shown how two different dimensions of software architecture, which we call vertical and horizontal, can be used for constructing reactive systems. These dimensions are in some sense dual, and they are also incompatible with each other in the sense that it does not seem useful to combine them in a single system of modules. Therefore, the horizontal dimension, which is currently visible in design patterns and aspect-oriented programming, for instance, remains as a crosscutting dimension with respect to conventional vertical units of implementation.

The two dimensions can be combined in different ways. As already discussed, the DisCo approach uses the horizontal dimension as the primary dimension, and provides an incremental specification method for composing specifications. A vertical implementation architecture is, however, anticipated in terms of a high-level view of objects and their cooperation. In contrast, most aspect-oriented approaches, in particular those influenced by AspectJ [22], have taken a more pragmatic approach. There, the primary architectural dimension is based on conventional vertical modularity, and crosscutting aspects are added to it by an auxiliary mechanism for horizontal modularity [6]. Other approaches include problem frames, where the horizontal dimension is used for decomposing a problem into subproblems whose sizes are more manageable [9].

Since layers provide abstractions of the total system, their explicit use seems natural in a structured approach to specification, and also in incremental design of systems. At the programming language level it is, however, difficult to develop general-purpose support for horizontal architectures. This means that a well-designed horizontal structure may be lost in an implementation, or entangled in a basically vertical architecture. However, newer implementation techniques, including aspect-oriented ones in particular, have enabled a wider range of options [1, 21].

We conclude that much work is still needed for making horizontal architectures easier to use in practice. Besides issues concerning implementation, commercial tools for utilizing the horizontal dimension in software design are not available. Thus, it remains a topic of future study to build a tool set where a commonly used notation, such as UML, is used to support the approach in a rigorous way. Accomplishing this may require the introduction of new concepts and terminology, such as those proposed by OMG's Model-Driven Architecture initiative.

5. ACKNOWLEDGMENTS

The authors would like to thank the other members of the DisCo team and Shmuel Katz for the discussions on the subject. This research has been partially funded by Academy of Finland, project ABESIS grant number 100005.

6. REFERENCES

[1] Timo Aaltonen, Joni Helin, Mika Katara, Pertti Kellomäki, and Tommi Mikkonen. Coordinating objects and aspects. *Electronic Notes in Theoretical Computer Science*, 68(3), March 2003.

[2] Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. `http://www.jucs.org`.

[3] Timo Aaltonen and Tommi Mikkonen. Managing software evolution with a formalized abstraction hierarchy. In *Proc. Eight IEEE International Conference on Engineering of Complex Computer Systems*, Greenbelt, MD, USA, December 2002. IEEE Computer Society Press.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[5] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[6] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[7] Object Management Group. MDA guide version 1.0.1. OMG Document Number omg/2003-06-01, June 2003. At URL `http://www.omg.org/mda/specs.htm`.

[8] Object Management Group. MDA homepage. At URL `http://www.omg.org/mda/`.

[9] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison–Wesley, 2001.

[10] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proc. 12th International Conference on Software Engineering*, pages 63–71, 1990.

[11] Mika Katara. Composing DisCo specifications using generic real-time events – a mobile robot case study. In Jaan Penjam, editor, *Software Technology, Proc. Fenno–Ugric Symposium*, pages 75–86, Sagadi, Estonia, August 1999. Institute of Cybernetics at Tallinn Technical University (Technical Report CS 104/99). At URL `http://disco.cs.tut.fi`.

[12] Mika Katara and Shmuel Katz. Architectural views of aspects. In *Proc. 2nd International Conference on Aspect-Oriented Software Development*, pages 1–10, Boston, MA, USA, March 2003. ACM Press.

[13] Shmuel Katz and Joseph Gil. Aspects and superimpositions. Position paper in ECOOP 1999 workshop on Aspect-Oriented Programming, Lisbon, Portugal, June 1999.

[14] Pertti Kellomäki. A structural embedding of Ocsid in PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLS2001*, number 2152 in Lecture Notes in Computer Science, pages 281–296. Springer Verlag, 2001.

[15] Pertti Kellomäki and Tommi Mikkonen. Design templates for collective behavior. In Elisa Bertino, editor, *Proc. ECOOP 2000, 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 277–295. Springer-Verlag, 2000.

[16] R. Kurki-Suonio. Modular modeling of temporal behaviors. In S. Ohsuga, H. Kangassalo, H. Jaakkola, K. Hori, and N. Yonezaki, editors, *Information Modelling and Knowledge Bases III*, pages 283–300. IOS Press, 1992.

[17] Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In Bernd Krämer, Naoshi Uchihira, Peter Croll, and Stefano Russ, editors, *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems, PDSE'98*, pages 94–102. IEEE Computer Society, 1998.

[18] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[19] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[20] Risto Pitkänen. Formal model driven approach to developing enterprise systems. Technical report, Institute of Software Systems, Tampere University of Technology, 2004. At URL `http://disco.cs.tut.fi/reports/formalmdd.pdf`.

[21] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[22] AspectJ home page at `http://aspectj.org`.

[23] DisCo home page at `http://disco.cs.tut.fi`.

Exploring Aspects in the Context of Reactive Systems

Karine Altisen
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
Karine.Altisen@imag.fr

Florence Maraninchi
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
Florence.Maraninchi@imag.fr

David Stauch
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
David.Stauch@imag.fr

ABSTRACT

We explore the semantics of aspect oriented programming languages in the context of embedded reactive systems. For reactive systems, there are a lot of simple and expressive formal models that can be used, based on traces and automata. Moreover, the main construct in the programming languages of the domain is parallel composition, and the notion of *transverse* modification is quite natural. We propose: 1) a semantical definition of an aspect, allowing one to study its impact on the original program; 2) some operational constructs that can constitute the basis of a weaving mechanism.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Theory, Verification

Keywords

reactive systems, aspect-oriented programming, formal semantics, synchronous languages

1. INTRODUCTION

1.1 Generalities

Whatever be the structuring mechanisms offered by a programming language, it is always possible to find a program P and some functionality F that P should provide (or a property P should ensure), in such a way that F cannot be implemented only by some modifications of P that would "respect" its existing structure. F is then called an *aspect* [6]. For instance, adding debugging facilities to a program is an aspect because its implementation is likely to need small modifications everywhere.

FOAL '04 Lancaster, GB

Aspect programming studies how aspects can be *specified*, and how the additional code needed to implement F can be automatically *woven* into P, statically or dynamically. All these definitions are quite informal and, for a given programming language, it is not always clear whether a given functionality is indeed an aspect. This shows that any formal definition of aspects and weaving has to provide precise definitions for, at least: 1) the structure of programs (this is the simplest, but we have to choose between a concrete syntax and a very abstract semantic structure); 2) the notion of a functionality to be implemented in a given program; 3) an aspect specification language; 4) a program transformation (not necessarily static) for the weaving process.

There are very few attempts at defining the notion of aspect formally, independently of any language. It is probably too early, and we need to understand the notion of aspect better, concentrating on specific application domains.

1.2 Objectives of the paper

We would like to question the notion of aspect a bit further, in a formally defined context that allows one to give unambiguous definitions for all the important notions related to aspects. We choose reactive systems because they provide a very clear notion of the input/output interface of a program, their design relies on a parallel composition and a communication mechanism, and the definition of their sequential behavior is simple. Additionally, we will consider *static* weaving mechanisms only, because reactive systems are most of the time *embedded*: a monitoring system that may stop the program when its behavior diverges from the expected one is useless, because the system cannot be repaired on-line. See section 5 for more comments on the relationship with so-called *property-enforcing* techniques that can sometimes be considered as aspect-weaving.

Moreover, we choose a formalism made of synchronous compositions, because the synchronous broadcast and the compiled synchronous parallel composition are already very powerful (for instance, rendez-vous can be implemented by the so-called "instantaneous dialogue"). A a lot of things can be implemented by adding components synchronized with an existing program. Something that cannot be implemented this way really deserves the name of aspect. There exist several synchronous languages with very distinct constructs, that all have a semantics in terms of communicating Mealy machines. Considering aspects at the level of this model guarantees that the definitions are not too particular to a

given language.

We try to formalize the following observations, questions and requirements:

(α) What is the semantic impact of weaving? Can the behavior of the modified program be completely distinct from the behavior of the original program? We would like the new program to be somewhat comparable to the old one, at least on a subset of its inputs, and/or on a subset of its instants.

(β) An aspect specification language has to talk about P. How detailed can it be? Can it talk about the interface only, or about the internals of the program?

(γ) How can we be sure that F cannot be implemented in P with simple modifications that do not require the aspect point of view?

The contribution of the paper is a simple formal framework in which aspects of reactive systems can be specified and studied. It is a first approach to this question, and this work can be continued in various ways.

Section 2 defines a formal model for reactive systems; Section 3 is a non exhaustive list of functionalities that constitute good candidates for the notion of aspect in reactive systems; Section 4 is our proposal: a notion of aspects and how to specify them, formal basis for the weaving process; Section 5 is a (probably) non exhaustive list of related work; Section 6 is the conclusion, mainly a list of perspectives.

2. THE MODEL

The formal model for reactive systems is given here in two levels: 1) a trace semantics that is common to a wide variety of languages, and 2) a set of operators on reactive and deterministic Mealy machines that can be considered as an ideal view of a synchronous language. All questions and problems that can be formulated in the first level are therefore general to a lot of languages; the second level will be used whenever we need a *structure* in the programming language.

2.1 Traces and trace semantics

Definition 1 (Traces) *Let $\mathcal{I}$, $\mathcal{O}$ be sets of Boolean input and output variables representing signals from and to the environment. A* trace on $\mathcal{I} \cup \mathcal{O}$, *$t$, is a function: $t : \mathbb{N} \longrightarrow [(\mathcal{I} \cup \mathcal{O}) \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$. At each instant $n \in \mathbb{N}$, the given trace t provides the valuations of every input and output.*

A set of traces $st = \{t \mid t \text{ is a trace on } \mathcal{I} \cup \mathcal{O}\}$ is deterministic *iff*

$$\forall t, t' \in st.\big(\forall n \in \mathbb{N}.\forall i \in \mathcal{I}.t(n)[i] = t'(n)[i]\big) \implies \big(\forall o \in \mathcal{O}.t(n)[o] = t'(n)[o]\big).$$

We write $\{i_1, i_2, ..., i_{\|\mathcal{I}\|}\}$ for the set of inputs $\mathcal{I}$. A set of traces $st = \{t \mid t \text{ is a trace on } \mathcal{I} \cup \mathcal{O}\}$ is reactive *iff*

(i) $\forall (v_1, v_2, ..., v_{\|\mathcal{I}\|}) \in \{\mathtt{true}, \mathtt{false}\}^{\|\mathcal{I}\|}.$
$\exists t \in st.\forall k.t(0)[i_k] = v_k$

(ii) $\forall t \in st.\forall n \in \mathbb{N}.\forall (v_1, v_2, ..., v_{\|\mathcal{I}\|}) \in \{\mathtt{true}, \mathtt{false}\}^{\|\mathcal{I}\|}.$
$\exists t' \in st.(\forall m \leq n.\forall i \in \mathcal{I}.t(m)[i] = t'(m)[i])$
$\wedge\ (\forall k.t'(n+1)[i_k] = v_k)$

A set of traces is a way to define the semantics of a program P, given its inputs and outputs. From the above definitions, a program P is *deterministic* if from the same sequence of inputs it always computes the same sequence of outputs. It is *reactive* whenever it allows every sequences of every eligible valuations of inputs to be computed. Determinism is related to the fact that the program is indeed written with a programming language (which has deterministic execution); reactivity is an intrinsic property of the program that has to react forever, to every possible inputs without any blocking.

Definitions 2 and 3 below define transformations on traces that will be used to characterize the impact of weaving on the semantics of a program.

Definition 2 (Masking traces) *Given a trace t on $\mathcal{I} \cup \mathcal{O}$ and a subset $S \subseteq (\mathcal{I} \cup \mathcal{O})$ the* masking *of t by $(\mathcal{I} \cup \mathcal{O}) \setminus S$ is a trace $t_{|S}$ on S such that:*

$$\forall e \in S.\forall n \in \mathbb{N}.t_{|S}(n)[e] = t(n)[e].$$

Definition 3 (Clocking traces) *Given a trace t on $\mathcal{I} \cup \mathcal{O}$ and a subset $M \in \mathbb{N}$ the* clocking *of t by M is a trace $t_{\|M}$ on $\mathcal{I} \cup \mathcal{O}$ such that:*

$$\forall e \in (\mathcal{I} \cup \mathcal{O}).\forall n \in \mathbb{N} \setminus M.t_{\|M}(n)[e] = t(n)[e].$$

Notice that given a trace t and a subset M of natural, this defines a set of clocking including t itself.

Those definitions are naturally extended to sets of traces and programs.

2.2 Elements of language definition

The core of a synchronous language is made of input/output automata, the synchronous product, and the encapsulation operation.

Definition 4 (Automaton) *An* automaton $\mathcal{A}$ *is a tuple $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{Q}$ is the set of states, $s_{init} \in \mathcal{Q}$ is the initial state, $\mathcal{I}$ and $\mathcal{O}$ are the sets of Boolean input and output variables respectively, $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $\mathcal{B}ool(\mathcal{I})$ denotes the set of Boolean formulas with variables in $\mathcal{I}$. For $t = (s, \ell, O, s') \in \mathcal{T}$, $s, s' \in \mathcal{Q}$ are the source and target states, $\ell \in \mathcal{B}ool(\mathcal{I})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.*

The semantics of the automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is given in terms of a set of traces on $\mathcal{I} \cup \mathcal{O}$. A trace t of $\mathcal{A}$ is inductively built as follows:

- at time 0, the automaton is in state s_{init};
- if at time $n \in \mathbb{N}$, the automaton is in state $s \in \mathcal{Q}$ and receives input valuations v_i for $i \in \mathcal{I}$, then at time $n+1$ it reaches state s', emitting O:

$$\big(\forall i \in \mathcal{I}.t(n)[i] = v_i \in \{\texttt{true}, \texttt{false}\} \wedge$$
$$\forall o \in \mathcal{O}.t(n)[o] = w_o \in \{\texttt{true}, \texttt{false}\}\big) \implies$$
$$\big(\exists (s, \ell, O, s') \in \mathcal{T} \wedge O = \{o \in \mathcal{O}, w_o = \texttt{true}\} \wedge$$
$$\ell \text{ valuates to true w.r.t. the } v_i\text{'s}\big).$$

We note *Trace*($\mathcal{A}$) the set of all traces built following this scheme: *Trace*($\mathcal{A}$) defines the semantics of $\mathcal{A}$. The automaton $\mathcal{A}$ is said to be *deterministic* (resp. *reactive*) iff its set of traces *Trace*($\mathcal{A}$) is deterministic (resp. reactive) (see def. 1).

Definition 5 (Synchronous Product) *Let* $\mathcal{A}_1 = (\mathcal{Q}_1, s_{init1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ *and* $\mathcal{A}_2 = (\mathcal{Q}_2, s_{init2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ *be automata. The* synchronous product *of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *is the automaton* $\mathcal{A}_1 || \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{init1} s_{init2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ *where* $\mathcal{T}$ *is defined by:*

$$(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \iff$$
$$(s_1 s_2, \ell_1 \wedge \ell_2, O_1 \cup O_2, s_1' s' 2) \in \mathcal{T}.$$

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and reactivity.

Definition 6 (Encapsulation) *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ *be a set of inputs and outputs of* $\mathcal{A}$. *The* encapsulation *of* $\mathcal{A}$ *w.r.t.* Γ *is the automaton* $\mathcal{A} \setminus \Gamma = (\mathcal{Q}, s_{init}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ *where* $\mathcal{T}'$ *is defined by:*

$$(s, \ell, O, s') \in \mathcal{T} \wedge \ell^+ \cap \Gamma \subseteq O \wedge \ell^- \cap \Gamma \cap O = \emptyset \iff$$
$$(s, \exists \Gamma.\ell, O \setminus \Gamma, s') \in \mathcal{T}'$$

ℓ^+ is the set of variables that appear as positive elements in the monomial ℓ (i.e. $\ell^+ = \{x \in \mathcal{I} \mid (x \wedge \ell) = \ell\}$). ℓ^- is the set of variables that appear as negative elements in the monomial l (i.e. $\ell^- = \{x \in \mathcal{I} \mid (\neg x \wedge \ell) = \ell\}$).

Intuitively, a transition $(s, \ell, O, s') \in \mathcal{T}$ is still present in the result of the encapsulation operation if its label satisfies a local criterion made of two parts: $\ell^+ \cap \Gamma \subseteq O$ means that a local variable which needs to be true has to be emitted by the same transition; $\ell^- \cap \Gamma \cap O = \emptyset$ means that a local variable that needs to be false should *not* be emitted in the transition.

If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by $\exists \Gamma.\ell$ for the input part, and by $O \setminus \Gamma$ for the output part.

In general, the encapsulation operation does not preserve determinism nor reactivity. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [2]).

An example

Figure 1-(i) shows a 3-bits counter. Dashed lines denote parallel compositions and the overall box denotes the encapsulation of the three parallel components, hiding signals `b` and `c`. The idea is the following: the first component on the right receives `a` from the environment, and sends `b` to the second one, every two `a`'s. Similarly, the second one sends `c` to the third one, every two `b`'s. `b` and `c` are the carry signals. The global system has `a` as input and `d` as output; it counts `a`'s modulo 8, and emits `d` every 8 `a`'s. Applying the semantics of the operator (first the product of the three automata, then the encapsulation) yields the simple flat automaton with 8 states (Figure 1-(ii)).

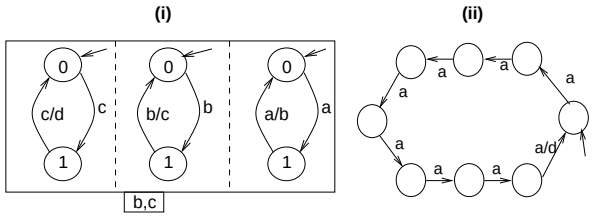

Figure 1: A 3-bits counter. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering cond. / outputs emitted`", e.g. the transition labelled by "`a/b`" is triggered when `a` is true and emits `b`.

3. EXAMPLES OF ASPECTS FOR REACTIVE SYSTEMS

Among the traditional programming examples in the domain of reactive systems, there are some typical cases in which one could think of introducing aspect programming. The candidate "aspects" for reactive systems sometimes depend on the programming paradigm of the language used, but not always. We give several examples below.

3.1 Conditional reinitialization

Being able to re-initialize a reactive system on the occurrence of a special signal is useful in several contexts. For instance, if we program two different systems S1 and S2, in isolation, and then want to design a more complex one in which the two of them have to be used, it is likely that one of the systems is active in a given period of time, then the other one, then the first one again. When the first system restarts, it should have the same behavior as if it were started for the first time. Making a reactive program reinitialiazable means adding reactions to a special signal r, that leads the system to its initial configuration, from any other state it may have reached.

In some languages, there is a dedicated construct for this. In Esterel [2], if `S` is the program of the original system, and `r` the additional signal, an expression similar to "`loop S each r`" is the reinitializable program. This is true for a main program, and this is also true for a subprogram in a complex context: `S` can be replaced by `loop S each r` without changing the context.

In Lustre [1], which is not based upon the imperative paradigm,

reiniatializing a program needs modifications everywhere in the code. Indeed, there is a special conditional structure whose meaning is: *if (this is the beginning of time) then ... else ....* When a program has to be made reinitializable by a signal `r`, all occurrences of this special construct have to be modified to give: *if ((this is the beginning of time) OR r is present) then ... else ....*

In the language of parallel automata we presented in section 2.2, making an automaton reinitializable means adding transitions from any state to the initial one, with the condition `r`, and no emitted signal; moreover, all the existing transitions have their condition reinforced by `not r`. Notice that, in a more sophisticated language based on explicit automata, one would introduce hierarchy of states à-la Statecharts, and reinitialization would become trivial (see, for instance [7]).

3.2 Conditional inhibition

Conditional inhibition means the following: when an additional signal `ck` is present, the new system behaves as the old one. When this signal is not present, then the system does not evolve, it keeps its current state, until `ck` is present again.

In dataflow languages for reactive systems like Lustre or Signal, this kind of behavior is a special case of a system with multiple *clocks*. There are dedicated constructs that allow to manipulate clocks. In Lustre, if `S` is the original program, then something like "`current (S when ck)`" would do the job.

In our language of automata, adding conditional inhibition means (1) enforcing the triggering condition of each transition by $\wedge\neg ck$ and (2) adding to each state s a self-loop transition $(s, ck, \emptyset, s)$ with triggering condition ck and emitting nothing.

3.3 Adding a validity bit

This last example is very common in fault-tolerant systems. Consider a reactive system with an input `i` and an output `o`. `i` comes from an external physical device, that may "fail" temporarily. Some physical properties make it possible to produce the information: *the value transmitted on input* `i`*, in the current instant, is not valid.*

The problem is to rewrite the program, adding a *validity bit* `v` to the input `i`. The new program should behave as the original one when its input is valid (i.e. when `v` is true). When `v` is false, the value of `i` should not be "*taken into account*". This quite informal specification rises two questions.

First, forbidding the use of the input at a given instant in time means that: a) we have to give a default value for the outputs that depend on it; b) we also have to make sure that the input does not serve for updating the memory.

Second, in all cases, one has to determine whether the value computed for the outputs, or the way memory is updated, really *depends* on the input `i` in the current instant. If it is possible to write an additional parallel component that observes `i` and each output `o`, and delivers a Boolean signal `o_depends_on_i`, then the program can be modified quite easily, with transformations similar to the ones we described for reinitialization or inhibition (this is similar for memory updates). But writing this additional component is difficult: `o_depends_on_i` means *if we change* `i`*, then* `o` *changes*, and it cannot be computed by observing *a single* input/output sequence, but the whole set of traces. So it cannot be implemented by an additional parallel component.

All transformations that mention such a dependency notion between inputs and outputs are good examples for point γ in the introduction: we do not know how to program them with the usual constructs of our languages, and it should even be possible to prove that we cannot do so. See more comments in section 6.

4. OUR PROPOSAL

Our proposal is made of two parts: first, we characterize the notion of aspect in a very declarative setting. Second, we propose a set of elementary transformations on automata that may be the basis of a weaving mechanism.

4.1 Characterizing the semantical influence of aspects

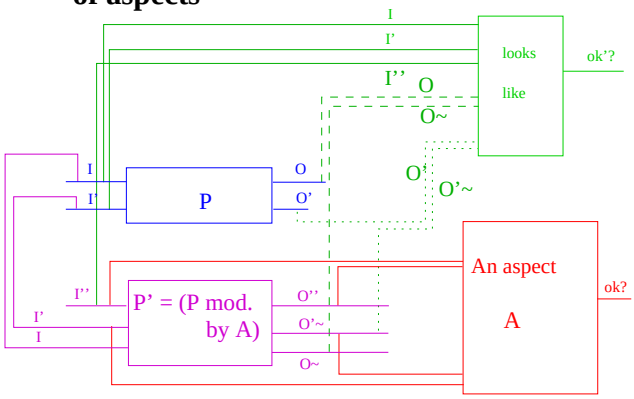

Figure 2: The semantic scheme is a data flow block diagram using the same notations as in the text; the inputs of a box enter the box at its left and the outputs goes out at its right; the box for the aspect *Asp* and the one for the predicate *looks_like* are given in terms of observers, i.e. they only have an output *ok* (resp. ok') that is expected to be always true. $\mathcal{O}$ (resp. $\mathcal{O}'$) denotes a syntactic copy of the set of outputs O (resp. O'): P emits $O \cup O'$ while P'emits $\mathcal{O} \cup \mathcal{O}'$ which may not have the same values.

Figure 2 summarizes the semantic framework in which we define aspects: P is a program (a set of traces on its inputs and outputs). *Asp* is an aspect, allowed to deal with some inputs $I' \subseteq \mathcal{I}$ and some outputs $O' \subseteq \mathcal{O}$ of P; it is also allowed to add some inputs I'' and outputs O'' to P. *Asp* is given as a set of traces on $I' \cup I'' \cup O' \cup O''$. Weaving *Asp* into P yields a modified program $P' = P \lhd Asp$ such that:

- P' has $\mathcal{I} \cup I''$ as inputs and $\mathcal{O} \cup O''$ as outputs;

- P' is *consistent* with Asp, i.e.,

  $$P'_{|I' \cup I'' \cup O' \cup O''} \subseteq Asp;$$

  the set $I' \cup I'' \cup O' \cup O''$ on which P' and Asp are compared is exactly the set of inputs/outputs of Asp; this means that P' on this set must be some of the traces defined by Asp.

The two items above define P' w.r.t. Asp but they do not at all constrain P' to be related to P, in any sense: for example, take for P' every traces in Asp, and extend them by adding the inputs I and outputs O with arbitrary values (e.g. `false` everywhere); it yields P' such that $P'_{|I' \cup I'' \cup O' \cup O''} \subseteq Asp$. This is caricatural, but it illustrates a major semantic problem of aspects from our point of view: we implicitly want P' to be related to P. This is the motivation for our third point.

Third, we require that P and P' be comparable.
looks_like (P, P') is a predicate that compares the traces of P and P' for the inputs and outputs they have in common: it compares $P_{|\mathcal{I} \cup \mathcal{O}}$ with $P'_{|\mathcal{I} \cup \mathcal{O}}$. Different classes of programs and aspects may require different definitions for *looks_like* . Here are some examples:

- We may require that P and P' have the same traces on the inputs/outputs that are not involved in the definition of the aspect: *looks_like* $\stackrel{\text{def}}{=} P_{|I \cup O} = P'_{|I \cup O}$;
- one might enforce this condition by imposing that P and P' have the same behavior on $I' \cup O'$ at the instants when Asp does not modify them. Suppose, for example that Asp modifies the value of an output $o \in O'$ if a new input $i \in I''$ is `false`, as in the validity bit example. The definition of *looks_like* may then impose that o keeps its value whenever i is `true`; this condition is expressed by comparing the clocking of the traces of P' on the set of instants when i is `true` with the same clocking applied to P;
- another admissible comparison criterion is that the traces (masked or clocked) of P' are shifted by $n \in \mathbb{N}$ instants compared with those of P. This might be useful in case the aspect itself introduces this kind of shifting.

This list of comparison criteria is not exhaustive and a mix of the three solutions mentioned above as well as different ways of comparing traces of P and P' may be admissible.

4.2 Operational definition of aspects

This part describes an elementary transformation on automata proposed as a basic construct for a weaving process. It was motivated by some of the aspects one may imagine for reactive systems.

4.2.1 A Stateless Transformation

Consider an automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$. The idea is to modify the transitions sourced in a given set of states, by reinforcing their condition and adding some emitted events. Whenever a transition condition is reinforced, it means that the disjunction of conditions, for all transitions sourced in the same state, may no longer be "true" (the automaton is no longer reactive). To ensure that the result of the transformation is indeed reactive, we also add a transition with the missing condition. It could be a loop on the state, but it seems more general to allow any existing state to be its target state. The result is not always deterministic.

The parameters of the transformation are: a specification ψ of a set of traces on $\mathcal{I} \cup \mathcal{O}$; a condition C on $\mathcal{I}$; a condition m' on additional inputs; a set o' of additional outputs; a set o'' of additional outputs; a specification ψ' of a set of traces on $\mathcal{I} \cup \mathcal{O}$.

For any state q reachable from the initial state by a trace in ψ, for any of its outgoing transitions $q \stackrel{m/o}{\longrightarrow} q'$ whose condition m satisfies C, we transform it into $q \stackrel{m \wedge m'/o \cup o'}{\longrightarrow} q'$, and we add $q \stackrel{m \wedge \neg m'/o''}{\longrightarrow} q''$ for all q'' reachable from the initial state by a trace in ψ'.

4.2.2 Implementing the Examples

Reinitialization of a single automaton by r is obtained by applying this mechanism with the following parameters: $\psi =$ `true` (any sequence of inputs and outputs is accepted by ψ, meaning: all the reachable states); $C =$ `true` (for all transitions); $m' = \neg r$ and $o' = \emptyset$ (reinforcing the existing transitions); $m'' = r$ and $o'' = \emptyset$ (adding the "reset" transitions); $\psi' = \epsilon$ (specifies the initial state).

It is easy to show that any program made of automata, parallel compositions and encapsulations, can be made reinitializable by r. It is sufficient to apply the above transformation on each of the automata. r should be a fresh variable name, not used in the original program.

The conditional inhibition can be implemented with a similar transformation. Adding a validity bit is more complex, because it involves the notion of "*the transition really depends on the input at the current instant*" and this cannot be expressed by reinforcing conditions on existing transitions.

4.3 The Global Picture

Putting together the informal examples of section 3, the declarative point of view of section 4.1, and the operational view of section 4.2, we obtain the following:

(A) A set of aspect candidates: i.e. program transformations taken from classical examples of reactive programming. On this set of aspects, we may wonder whether we really need something new to implement them: do they deserve the name of aspect, or were there already implementable with traditional program constructs?

(B) A declarative characterization of the relationships between a program P, an aspect A, a new program P' resulting from the weaving of P and A, and a *comparison* criterion used to "control" how much P' can differ from P. This semantic scheme may be used to define a set of P' from P, A and the comparison criterion, but not in an operational way; moreover the set of P' that fit in the picture may be empty.

Finally the comparison criterion should not be completely ad-hoc for an aspect A. There could be several generic comparison criteria, depending on the type of applications. For instance, some applications in reactive programming may allow a modified program to be slightly shifted from the original one (giving the same outputs, but later), while others may not.

(C) A set of elementary program transformations (given on explicit automata) that could constitute the basis of a weaving mechanism.

This setting has some internal consistency requirements. For instance, each elementary transformation proposed in (C) should be such that the transformed program and the original one are comparable in the sense defined by the "looks-like" box in (B).

Once these internal consistency requirements have been expressed and verified on the general setting, the following questions make sense:

– From an informal notion of aspect taken from (A), how to derive the necessary transformations in (C)? We did this for the examples, in a very informal way.

– How to use the framework of (B) to specify one of the candidate aspects of (A)?

– From a formal specification of an aspect in the style of (B), how to produce a sequence of (C) transformations to be applied to P, in order to produce a program P' that fits in the (B) picture?

Finally, some of the general questions people ask about aspects can be expressed in the same semantical framework:

– If we take two trace-equivalent programs P_1 and P_2 and an aspect A, is it true that $P_1 \lhd A$ and $P_2 \lhd A$ are still trace-equivalent?

– How to compare $P \lhd A_1 \lhd A_2$ and $P \lhd A_2 \lhd A_1$? This is related to the more general notion of aspect interference.

5. RELATED WORK

Related work can be found in several directions. First, papers from the AOP community, with emphasis on semantics; the setting of [9] is very close to ours, since it considers sets of parallel input/output automata; however, synchronization is made by shared variables, and the author considers transitions made of sequences of elementary actions, on which the aspectJ-like "insert-before" or "insert-after" transformations make sense. In the above paper, the notions of "property inheritance" and "property superimposition" are defined. They seem to be particular cases of the declarative scheme of section 4.1, relating properties of $P \lhd A$ with properties of P (inheritance"), or A (superimposition"). Superimposition itself was introduced earlier (see [4]), ranging from *spectative* techniques (mere *observers*) to *invasive* techniques. Our work could also be called: static invasive superimposition.

Second, papers on controller synthesis techniques [10] and their application to interfaces [3] in component-based designs, because the definition of P' from P, the aspect specification and the "looks-like" predicate (shown in section 4.1) can be viewed as a controller-synthesis problem.

Third, there have been a number of papers on "property-enforcing techniques" that can be expressed in a aspect-oriented style. The idea is to express safety properties (e.g. with explicit automata like in [8]) and to "run" them in parallel with a program. When the safety property becomes false, the program is stopped. Running them in parallel means performing a product between the program and the property, and can be viewed as dynamic weaving. In our setting, performing the product between the program and a safety property does not need the notion of aspect: the safety property can be expressed as a subprogram in parallel with the original program P, and the whole can be compiled. This yields a new program Q with an output *ok* meaning: the property is true from the beginning of time. However, for embedded systems, waiting execution-time to know about a safety property is not a solution. Static verification techniques are used instead.

6. CONCLUSION

The semantic setting we presented is adequate for the questions α and β of the introduction, and their answers. We are currently investigating the various notions presented here on some simple examples similar to the reinitialization. The idea is not to obtain an automatic method from the semantic picture of figure 2: it is an instance of controller-synthesis, known to be quite costly. Moreover, we presented Boolean programs only, but the general interesting case uses integer variables in reactive programs. The problem then becomes undecidable. The semantic scheme is there only to give a clear meaning to the operational mechanisms we may invent in order to obtain automatic transformation techniques. The examples will probably lead to other basic transformations on automata. A natural extension of the one we presented would allow the creation of new states.

Question γ of the introduction has already been discussed about the validity bit example. We need to define precisely the notion of *dependency* between inputs and outputs in a reactive program, and to prove that it cannot be programmed with the usual constructs, thus deserving the name of aspect. Similar *dependency* notions (like the one of [5], developed for studying security policies) seem worth investigating.

7. REFERENCES

[1] J.-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, San Diego, Sept. 1985.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[3] L. de Alfaro and T. A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of*

Software Engeneering (ESEC/FSE-01), volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, Sept. 10–14 2001. ACM Press.

[4] N. Francez and I. R. Forman. Superimposition for interacting processes. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 230–245, Amsterdam, The Netherlands, 27–30Aug. 1990. Springer-Verlag.

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, CA, Apr. 1982. IEEE Computer Society Press.

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[7] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

[8] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[9] H. B. Sipma. A formal model for cross-cutting modular transition systems. In *Proceedings of the workshop on Foundations of Aspect-Oriented Languages*, Northeastern University, Boston, Mar. 2003.

[10] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, May 1987.