Foundations of Incremental Aspect Model-Checking[*][†]

Shriram Krishnamurthi[‡]   Kathi Fisler[§]

Brown University   WPI

May 27, 2006

Abstract

Programs are increasingly organized around features, which are encapsulated using aspects and other linguistic mechanisms. Despite their growing popularity amongst developers, there is a dearth of techniques for computer-aided verification of programs that employ these mechanisms. We present the theoretical underpinnings for applying model-checking to programs (expressed as state machines) written using these mechanisms. The analysis is incremental, examining only components that change rather than verifying the entire system every time one part of it changes. Our technique assumes that the set of pointcut designators is known statically, but that the actual advice can vary. It handles both static and dynamic pointcut designators. We present the algorithm, prove it sound, and address several subtleties that arise, including cascading advice application and problems of circular reasoning.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Programming Languages**]: Language Classifications

General Terms: Algorithms, Languages, Verification

Keywords: incremental verification, modular verification, model-checking, aspect-oriented programming, feature-oriented software

1   Introduction

There is growing consensus [5] that traditional software structures have notable abstraction weaknesses, and new software composition techniques are evolving to address these criticisms. These new techniques help a developer incrementally link segments of user-visible functionality (sometimes called "features" [16]) to programs. Writing these identifiable increments in conventional programming languages is challenging because an increment may affect parts of a program across traditional module boundaries: such increments are called *crosscutting*. This has led to a growing body of work on developing new forms of program modularity that go by different names including AHEAD, mixin layers, etc. [6, 8, 9, 25, 29, 47, 51, 59]. Some techniques are purely static, effectively manipulating the program's

---

1

source, while others have dynamic elements, offering the ability to reflect on the state of the program's execution and then to modify it.

Aspect-oriented programming (AOP) [39], especially as realized in the AspectJ language [38], is one of the most popular forms of incremental program composition. Aspects support both static and dynamic linking specifications, but their most distinctive techniques are arguably in the latter category. In particular, AspectJ provides a pattern language that can predicate the execution of an aspect on the current shape of a program's stack. It can therefore express a rich family of coherent, conceptual ideas that can be difficult to encapsulate in more traditional notions of a module.

With the popularity of AOP burgeoning, software engineers should expect tool support for all stages of the software cycle—including validation of behavioral properties. This is especially important because the expressive power that aspects unleash also heightens the potential for errors. Specifically, a program may satisfy a behavioral property but the application of an aspect to the program may now invalidate that property. The problem is especially insidious because the application of one aspect may cause the application of another, and so on; the resulting property violations can be quite subtle, making their identification daunting.

In this paper, we adapt CTL model checking [12] to verify aspect-oriented programs expressible as state machines. Our technique is designed to identify situations where the application of an aspect to a program (which may already be the result of aspect composition) may violate some desirable properties of the program and, also, of the resulting composed system. Verifying whether a given program exhibits certain properties is a straightforward application of model checking. In contrast, establishing that the application of an advice does not violate the properties is challenging. A model checker traverses program paths, so to establish a property of the composition, it must traverse paths that lie partly within the program and partly within the advice. However, demanding that the developer combine the advice and program prior to each verification is onerous:

1. The advice may be authored at a different time or in a different place than the program, just as modules are developed in spatial and temporal independence.

2. The advice may be edited repeatedly; verification time is proportional to the size of the system, so constantly verifying the changing advice against a fixed program is inefficient.

We instead develop a *modular* technique that analyzes a program and a family of applicable aspects independently, while caching just enough information to identify interactions between them.

Modularity has multiple interpretations. We assume a strict, black-box interpretation, whereby an aspect can assume and return control only at well-defined interface states. A looser, white-box interpretation is that modularity merely means avoiding repeated analysis over the program, but otherwise making information from all states of the program available when analyzing aspects. By treating every pair of states as potential points for attaching advice, our technique could simulate this looser notion, and thus support a much richer family of aspects [35].

To enable modular analysis, we require some information from the programmer specifying where an aspect will apply, which enables our technique to cache some information about the program. Specifically, we depend on specifications called pointcut designators, which programmers already write. Changes to these specifications may invalidate the cached information, forcing a fresh round of modular analyses. Therefore, we conceive this as an *incremental* technique: analogous to incremental compilation, it triggers fresh verification when certain portions of the system

change, while trying to avoid having to verify the system as a whole. Because our technique supports static linking specifications in addition to dynamic ones, it can just as easily be applied to many of the other attempts to improve program modularity [8, 25, 47, 59].

Our result suggests a change to the design of aspect languages. Currently, an aspect in AspectJ requires the co-specification of where an aspect will apply (the pointcut designator) and what it will perform at those points. The formal model in this paper mirrors this tradition. In contrast, however, our verification technique exploits the separation of these two concerns: knowing where the aspect will apply enables our algorithms to record information that can be used to verify the actual injected behavior, which may only be known later (and may change repeatedly).[1]

To use the terminology of Filman and Friedman [24], our work values "quantification" over "obliviousness". In this respect, our work is not alone: there is growing interest in models that have partial specifications of aspectual behavior. Kiczales and Mezini's paper on modular reasoning proposes interfaces with lightweight aspect specifications [40]. Furthermore, the paper by Sullivan, et al. [61] strongly challenges the traditional belief that obliviousness is central to aspect-orientation. Since the separation we introduce can aid other reasoning techniques, as well as tools such as compilers, we believe it would be beneficial for future aspect language designs to cleave the definition of an aspect asunder.

To apply finite-state model checking to infinite-state source programs we must impose some restrictions on the kinds of systems we can analyze, and these are augmented by the nature of aspects. Our work therefore makes the following assumptions:

- The model restricts the sharing of data between the program and advice. It permits advice to read the data of the program, which is already sufficient to implement standard aspect examples such as tracing and logging. The model does not, however, permit an aspect to modify a program's data.

  This begs the question of whether it is possible to have non-trivial aspects that do not modify the main program's data. To answer this question, we examined all the examples in the book by Laddad [43], which appears to be the most comprehensive catalog of non-trivial aspect uses and encompasses many of the examples presented in the research literature (such as adding transactional support, and a variation on implementing design patterns). Our examination of the book revealed that virtually all of its examples fit the assumption we make: most of the interesting advice reads the program's data, but virtually no advice modifies it. The primary counter-example we found was on pages 404–405: an overdraft rule that transfers money from one account into another. (Even the variants of that rule presented in the same chapter (Chapter 12) do not have this property.) Therefore, there are numerous interesting and non-trivial aspects that fit our model.

  It is not impossible to verify programs when advice are permitted to modify the program's data, but it does require a different approach to verification (such as three-valued model checking). We discuss these approaches briefly in section 8. That work is largely orthogonal to the approach we present here, but would need to be augmented with some of the techniques discussed in this paper.

- The form of program model we describe here is not well-suited to programs with heavily recursive control structure. Heavily recursive programs tend to take this form because they are processing rich, recursive data.

---

[1]Our technique does not need the pointcut designators to be specified with the program: it only demands that the designators be supplied at the time of *verifying* the program. This is a subtle but important difference.

These are, however, the kinds of programs for which model-checking is itself often ill-suited, so our choice of model does not impose a fresh restriction beyond that already placed by the use of model-checking.

- Although model checking is, in principle, both sound and complete, we can only ensure soundness. We discuss this in the context of algorithm 5 in section 6.3.

We propose the following route to navigating this paper. Readers unfamiliar with aspects or model checking should consult the background material in section 2. Those who want an overview of the work can skim section 3 for the formal models and then read section 4, which presents the main contribution through a running example, covering the subtleties that arise. The details of how to approximate program stacks, which section 4 depends on, are in section 5. Understanding the rest of the paper requires a more thorough understanding of section 3. A more precise exposition of the algorithms is in section 6. Finally, the stout of heart will want to study the proofs, which are in section 7. They will be rewarded with a detailed description of how our algorithm properly addresses the circular reasoning issues raised in section 4.4.

2 Background

2.1 Aspect-Oriented Programming

This paper uses a simplified version of AOP. Informally, our model has a notion of *advising*, or altering, a program's behavior. The locations where this alteration occurs are called the *joinpoints*, and we say that the altered code has been *advised*. Advice is a fragment of code that is executed either before, after or "around" the evaluation of the joinpoint. (The example in section 4.1 makes these notions concrete.) An around advice is executed in place of the original joinpoint, though the advice can execute a *proceed* command, which executes the code that would otherwise have been elided. In this way, the programmer who uses AOP can directly simulate some of the power of metaprogramming [37].

The AspectJ implementation of AOP provides a powerful language for describing when an advice should apply. This language, of *pointcut designators* (PCDs), can name either static or dynamic conditions under which to advise the program. Static PCDs name static program attributes, while dynamic PCDs specify a run-time condition. The dynamic PCD language of AspectJ permits patterns over the shape of the stack, so that programmers can, for instance, write a PCD of the form "when procedure p is being invoked in the dynamic extent of procedure q" (i.e., a stack frame for q is lower on the stack when p becomes the procedure at the top of the stack).

2.2 Model Checking

Model checking is a popular automated verification technique used to establish properties of finite-state systems. A model checker consumes a description of a system, usually given as a state machine (technically a Kripke structure), and a specification of a property (in a temporal logic) that the system must obey. The state machine can be non-deterministic. The model checker exhaustively explores the state machine to search for executions that could violate the property. The result is either a counterexample showing how the system could violate the property, or a statement that the system respects the property.

Model checking algorithms exist for a variety of temporal logics for property specification. This work uses CTL model checking [12]. The atoms of CTL are propositions that label states. CTL permits combination of these atoms

using the standard propositional operators and connectives (negation, conjunction, implication, etc). Finally, CTL can capture *temporal* properties. A statement of the form $[\phi \; \mathsf{U} \; \psi]$ (where ϕ and ψ are both CTL formulas) is true at a state if ϕ is true now and in the future until a state where ψ is true (read the U as "until"). Because many paths leave a state, CTL requires us to quantify this statement by whether we expect the property to hold in all possible future worlds or only in some. The CTL formula $\mathsf{A}[\phi \; \mathsf{U} \; \psi]$ expects that on All paths, ϕ will hold in every state until a state where ψ is true (which must eventually occur), while $\mathsf{E}[\phi \; \mathsf{U} \; \psi]$ requires that there Exists a path where this holds. Other temporal operators express "in next states" (AX and EX), "in all future states" (AG and EG), and "in some future state" (AF and EF).

The formal syntax and semantics of CTL are as follows:

Definition 1 (CTL **Syntax**) The set of CTL formulas contains propositions and the logical constants true and false, and is closed under the following operators, where ϕ and ψ are CTL formulas: $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\mathsf{AX}(\phi)$, $\mathsf{EX}(\phi)$, $\mathsf{AG}(\phi)$, $\mathsf{EG}(\phi)$, $\mathsf{A}[\phi \; \mathsf{U} \; \psi]$, and $\mathsf{E}[\phi \; \mathsf{U} \; \psi]$. $\mathsf{AF}(\phi)$ and $\mathsf{EF}(\phi)$ abbreviate $\mathsf{A}[\mathsf{true} \; \mathsf{U} \; \phi]$ and $\mathsf{E}[\mathsf{true} \; \mathsf{U} \; \phi]$, respectively.

Definition 2 (CTL **Semantics**) A *Kripke structure* is a tuple $\langle S, R, L \rangle$ where S is a set of states, $R \subseteq S \times S$ is a transition relation, and L is a function from S to sets of atomic propositions that label states. A *path* is a potentially infinite sequence of states $s_0, s_1, \ldots$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. Given a Kripke structure $M = \langle S, R, L \rangle$, state $s \in S$, and a CTL formula φ, the formula φ is true at s (denoted $M, s \models \varphi$) under the following conditions:

- $M, s \models p$, where p is an atomic proposition, iff $p \in L(s)$
- $M, s \models \neg\phi$ iff $s \not\models \phi$
- $M, s \models \phi \vee \psi$ iff $M, s \models \phi$ or $M, s \models \psi$
- $M, s \models \phi \wedge \psi$ iff $M, s \models \phi$ and $M, s \models \psi$
- $M, s \models \mathsf{EX}(\phi)$ iff for some state s' such that $(s, s') \in R$, $s' \models \phi$.
- $M, s \models \mathsf{AX}(\phi)$ iff for all states s' such that $(s, s') \in R$, $s' \models \phi$.
- $M, s \models \mathsf{E}[\phi \; \mathsf{U} \; \psi]$ iff $M, s \models \psi$ or if there exists a path $s_0, s_1, \ldots, s_k$ starting from s such that $M, s_k \models \psi$ and for all $0 \leq i < k$, $s_i \models \phi$.
- $M, s \models \mathsf{A}[\phi \; \mathsf{U} \; \psi]$ iff $M, s \models \psi$ or if every path $s_0, s_1, \ldots$ starting from s contains a state s_k such that $M, s_k \models \psi$ and for all $0 \leq i < k$, $M, s_i \models \phi$.
- $M, s \models \mathsf{EG}(\phi)$ iff there exists an infinite path $s = s_0, s_1, \ldots$ such that for all i, $M, s_i \models \phi$.
- $M, s \models \mathsf{AG}(\phi)$ iff for every infinite path $s = s_0, s_1, \ldots$, $M, s_i \models \phi$ for all i.

Model checkers usually implement the CTL semantics by traversing a formula bottom-up, labeling each state with those subformulas that are true at that state. As a result, when the checker is done, *each state is labeled with all the sub-formulas of the property that are true of that state*. We will exploit this important invariant in this paper. (For readers who want to study the model checking algorithm, we recommend the expository presentation in the book by Huth and Ryan [33, pages 222–230].)

The dependence of our technique on this invariant justifies why our work uses CTL rather than LTL [64]. LTL is a temporal logic in which formulas describe *paths* rather than *states*. A state-labeling approach is known to handle a subset of LTL (the deterministic subset defined by Maidl [48]), but in general the state-centric view is inconsistent with the LTL semantics. It would be possible to create a modular aspect analysis using LTL instead, but we do not explore this problem in this paper.

3 Formal Models

3.1 Programs

We represent programs as Kripke structures (state machines). States correspond to statements and expressions in the program, while transitions reflect the control flow between expressions. A set of propositional labels indicates the information known at each state.

To more closely resemble the structure of source code, we start with state machines that capture individual functions (with special states to designate function call and return locations), including a main function. Given a set of state machines for functions, we then use a straightforward algorithm to produce a state machine for the entire program by inlining copies of the function state machines between call and return states (up to a given inlining depth parameter). It is important to note that this only requires the inlining process, not the program itself, to terminate. While this technique appears restrictive, it is the same one used by state-of-the-art tools such as FLAVERS [23] and Bandera [17], which have been successfully employed in several software verification tasks.

In principle, our work does not rely on inlining as a construction technique. All we require are state machines that follow a *stack discipline* of calls and returns (meaning that calls and returns are properly matched), and access to the bodies of individual advisable code (to analyze when an advice proceeds). Tools such as FLAVERS and Bandera consume program source and employ program analyses to generate state machines similar to those we need. We therefore regard the use of inlining as orthogonal to our work.

The rest of this section presents these details formally. The definitions assume the existence of a set *FN* of function names, including the name main.

Definition 3 A *state machine* M is a tuple $\langle S, T, L, S_{\mathrm{src}}, S_{\mathrm{sink}}, S_{\mathrm{call}}, S_{\mathrm{rtn}}, T_{\mathrm{cr}} \rangle$ where

- S is a set of states.

- $T \subseteq S \times S$.

- $L : S \to 2^{\mathrm{AP}}$ for some set of atomic propositions AP, which are boolean encodings of program data.

- $S_{\mathrm{src}} \in S$ and $S_{\mathrm{sink}} \in S$ such that S_{src} is a source and S_{sink} is a sink when viewing $\langle S, T \rangle$ as a directed graph. We call these the *source* and *sink* states of M. (Intuitively, these are the entry and exit points of the program fragment.)

- $S_{\mathrm{call}} \subset S$ and $S_{\mathrm{rtn}} \subset S$. We call the states in these sets *call* and *return* states, respectively. S_{call} and S_{rtn} are disjoint and are in a bijective relationship that is captured in $T_{\mathrm{cr}} \subset S_{\mathrm{call}} \times S_{\mathrm{rtn}}$ (T_{cr} is called the *call-return relation*). The states in S_{call} carry the label call(f) (where f is the function being called), and those in S_{rtn} are correspondingly labeled ret(f). Intuitively, every state in S_{call} denotes an invocation of a function, and the corresponding S_{rtn} state is where control returns when the function completes execution.

Preserving the projection of a state machine's transitions to those between call and return states (as recorded by T_{cr}) is important because the inlining construction will remove the edges between these states, but our verification technique needs to determine which return states correspond to which call states.

Definition 4 A *function* is a tuple $\langle name, M \rangle$, where $name \in FN$ and M is a state machine.

Algorithm 1 (Constructing Programs from Source) Given a set of functions with distinct names, including one named main, and a number indicating the inline-depth, generate a *program* by traversing main. At each call-return state pair, inline a *fresh copy* of the state machine for the function labeling the call state. To *inline* a function F between states c and r in M, add an edge from c to the source state S_{src} of F; and add an edge from the sink state S_{sink} of F to r. Continue inlining recursively in F until the depth parameter is exceeded, at which point add an edge between c and r.

3.2 Aspects

In a typical aspect language, a proceed statement is analogous to a procedure call to the elided code. To mimic the structure of our function call-and-return states, we model proceed statements with proceed-resume state pairs.

Definition 5 *Advice* is a state machine with two additional components, $S_{\mathrm{proceed}} \subset S$ and $S_{\mathrm{resume}} \subset S$. These are the proceed and resume states, respectively. S_{proceed} and S_{resume} are in a bijective relationship, and are disjoint from one another and from S_{call} and S_{rtn}. S_{proceed} and S_{resume} may be empty.

Pointcuts are sets of states at which advice can apply. Our model treats only function applications as joinpoints. Our techniques extend naturally to richer sets of joinpoints as long as those joinpoints are represented in the finite-state program model. The choice of joinpoint language could therefore affect the choice of model extraction algorithm. Pointcuts are specified through a subset of regular expressions over function calls; these regular expressions describe the shape of the stack at program states (as opposed to sequences of calls leading to states). This allows our language to capture dynamic pointcuts analogous to those captured by cflow and within in AspectJ. The following definition presents our regular expression language for PCDs. Section 6.1 describes the process of identifying the states that match a PCD.

Definition 6 The following grammar specifies expressions that define pointcuts:

A *pointcut atom* is one of the following:

- call(f) for some function name f (other than main)
- !call(f) for some function name f (other than main)
- true

A *pointcut element* is one of the following:

- a pointcut atom
- (e) where e is a pointcut element
- $e_1 \wedge e_2$ where e_1 and e_2 are pointcut elements
- $e_1 \vee e_2$ where e_1 and e_2 are pointcut elements

A *pointcut designator* (PCD) is one of the following:

- a pointcut element

- e^* where e is a pointcut element
- (d) where d is a pointcut designator
- $d_1; d_2$ where d_1 and d_2 are pointcut designators
- $d_1 \mid d_2$ where d_1 and d_2 are pointcut designators
- $d_1 \ \& \ d_2$ where d_1 and d_2 are pointcut designators

Informally, a state satisfies a PCD if some stack trace leading up to the state is in the language of the PCD. The operators have the usual semantics from propositional logic and regular expressions. The alphabet of these expressions are atoms of the form call(f) and !call(f) for some function name f, where !call(f) denotes a call to a function other than f. Allowing both call and !call atoms enables the language to distinguish between top-level and nested invocations (Kiczales [36] provides a good use-case for this). The $\wedge$ operator is only meaningful if at most one of its operands is a call label, since one state can have a call to at most one function. Operators $\mid$ and & represent disjunction and conjunction, respectively. Concatenation (;) distributes across $\mid$ and &, while $\mid$ and & distribute across one another in the usual way.

Our language is less expressive than full regular expressions by virtue of the restriction that Kleene-star operates only on pointcut elements rather than full PCDs. This restriction supports sound and complete identification of stack-based joinpoints (barring functions that cannot terminate) using a technique described in Section 5.2. Section 5.1 presents a sound but incomplete approach to joinpoint identification that could handle full regular expressions. Building this restriction into the grammar results in separate definitions for pointcut elements and PCDs. To help distinguish between these levels, we use different symbols for disjunction ($\vee$ and $\mid$) and conjunction ($\wedge$ and &) across the levels.

Definition 7 An *aspect* is a tuple $\langle d, t, A \rangle$, where d is a PCD, t is an advice type (before, after, or around), and A is an advice.

Given a program and an aspect, applying advice at the aspect's pointcut yields a new, *composed*, program. This program is constructed according to the type of advice, which indicates where to insert the advice relative to the pointcut (recall that we advise only function invocations). Figure 1 (left) illustrates the source and target states for transitions to inserted advice based on the advice type, assuming a PCD matched at the state labeled call(f). Our model inserts before and after advice within the scope of the advised function; this decision could be changed by moving the *Before*$_1$ and other labels to different positions relative to the call. We construct a composed program from a program and aspect according to the advice type as described in the following algorithm. (The definition of a composed program is necessary to provide a point of reference for demonstrating the soundness of our method in section 7.)

Algorithm 2 (Advising Programs) To advise a program with an aspect based on the type of advice:

If the advice type is before: for each state in the pointcut, replace the edge from the call state (*Before*$_1$) to the source state of the function with an edge to the source state of the advice; add an edge from the sink state of the advice to the source state of the function (*Before*$_2$), as shown in figure 1 (right).

If the advice type is after: for each state in the pointcut, replace the edge from the terminal state of the function (*After*$_1$) to the return state for the call with an edge to the source state of the advice; add an edge from the sink state of the advice to the return state for the call (*After*$_2$).
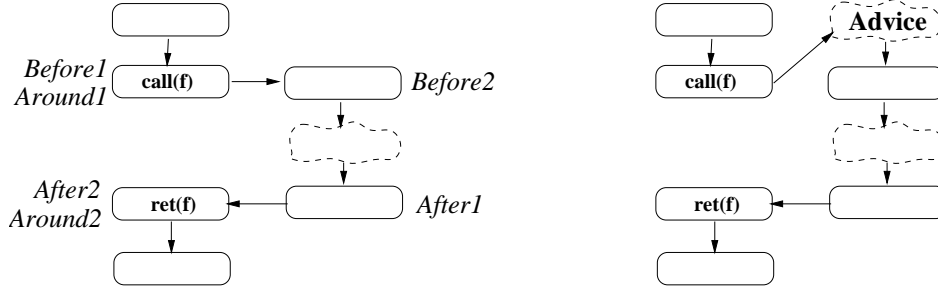
Figure 1: Where Advice Connects (left); Illustrating Before Advice (right)

If the advice type is around: replace the outgoing edge from the call state ($Around_1$) with an edge to the source state of the advice. Replace the incoming edge to the return state ($Around_2$) with an edge from the sink state of the advice. Between each pair of proceed-resume states in the advice, insert a copy of the body of the advised function. It is not necessary to remove any elided states because the algorithms in this paper depend on reachability, which automatically ignores any such states.

4   The Verification Process, Informally

Given our model of programs and aspects, we can now describe the actual verification technique. To make the presentation more accessible, we first present our work in terms of a simple running example, before formalizing it in section 6.

4.1   Scenario

Our running example illustrates aspects in the context of computer-based slideshow presentations. Consider preparing a slideshow presentation on a research project. Ideally, parts of the talk should be reusable in different venues such as a conference, a general computer science colloquium, or a seminar talk to colleagues in the same research area. The project motivation, core techniques, and experimental results should be common to all of the talks, but the colloquium version should review background material while the seminar version should cover deeper technical details. Different seminar audiences may even need different combinations of background material. Users of modern slideshow tools either copy and paste slides across the talks (which has obvious shortcomings when edits are required), or switch between slideshows during the talk. An aspect-oriented organization provides a clean alternative.

Consider a talk on the work in this paper as an example; figure 2 shows the talk outline in the form of a state machine (representing the control flow of the talk).[2] To maintain the analogy to programs, we view talks as programs that have cohesive sections that are invoked (akin to functions), as well as standalone slides (akin to program statements). The talk's background section contains slides on aspects, but no background slides on model checking. Figure 3 shows two pieces of advice: one containing slides on the model checking algorithm, and one containing slides on the formal

---

[2]Although this example program contains no cycles, our formal model and algorithms fully support cycles.
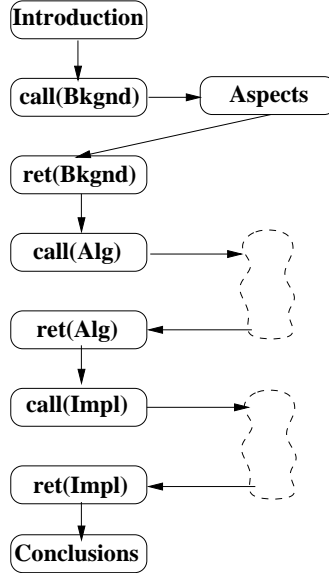
Figure 2: Sample Program

---

syntax and semantics of CTL. Turning these into aspects requires PCDs and advice types that identify where the additional slides would be appropriate. The model checking slides could be inserted at the end of the background section (as after advice); the CTL slides must precede the presentation of the model checking algorithm within the background section (as before advice). The following PCDs capture these two loci:[3]

P: true*; call(Bkgnd)

Q: true*; call(Bkgnd); true*; call(MC)

Observe that the original talk enables PCD P at the call(Bkgnd) state. Furthermore, the program *partially* enables PCD Q by calling Bkgnd, leaving open the possibility that an advice might insert the model checking section and thereby trigger the PCD.

We now turn our attention to properties. In the context of incrementally building up talks, we care that certain material be presented in a particular order: for example, the section on implementing an algorithm should never come before the algorithm description. The following CTL formula captures this property:

$$A[!call(Impl) \; U \; call(Alg)]$$

4.2 Basic Algorithm

We begin by model checking the property on the program. If the property fails to hold, the designer should correct the program before applying advice (recall that the goal in this work is to preserve program properties over applications of advice; section 9 discusses properties that arise from aspects). Section 2.2 mentioned that the model checking

---

[3]In AspectJ, these would be written as `call(Bkgnd)` and `call(MC) && cflow(call(Bkgnd))`, respectively.
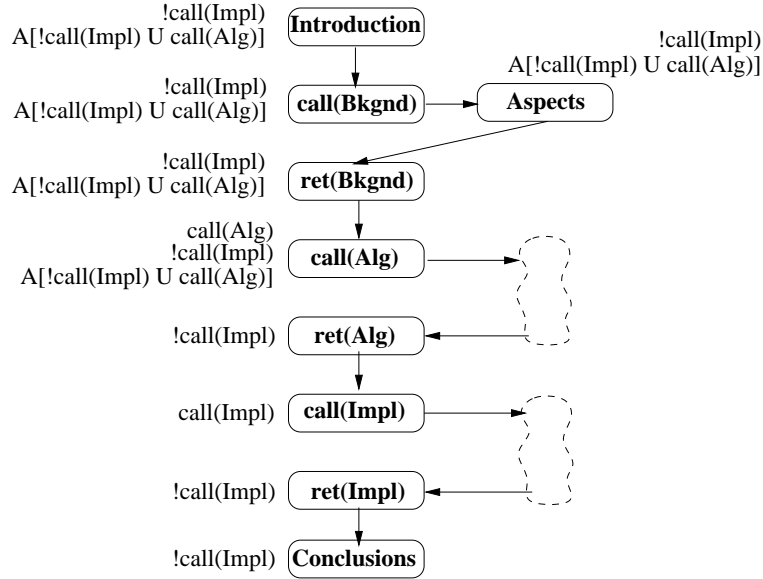
Figure 3: Two Pieces of Advice

Figure 4: Program Annotated with CTL Labels

algorithm labels each state of the program with those subformulas of the property that are true at that state. Figure 4 shows the program annotated with labels after checking the property. For each state at which advice might apply, these labels form the *interface* for verification at that state. The interface is effectively a cache of the state of the verification process at that program state.[4]

Interfaces need to be stored only for states to which advice might connect. Advice can connect to the $Before_1$, $Before_2$ and similar states from figure 1. PCDs identify the $Before_1$ states; the rest are computed from $Before_1$ states. Recall that PCDs describe stacks: we therefore need to locate those states at which the program's stack could match a PCD. Several techniques could perform this analysis; we defer a discussion of the tradeoffs and our specific technique to section 5. Figure 5 shows our example program annotated with both model checking labels and stack contents (the latter in bold). The call(Bkgnd) state matches PCD P; we must generate an interface at this state to use for verifying the advice when it becomes available. The interface reflects the state of the model checking process. It includes the labels on the states that lead to and return from the advice, but does not include information about the rest of the

---

[4]The interface is thus analogous to a closure that represents the delayed substitutions in a programming language with first-class procedures.

!call(Impl)
A[!call(Impl) U call(Alg)]   **Introduction**   !call(Impl)
                                                                      A[!call(Impl) U call(Alg)]

true* ; call(Bkgnd)   !call(Impl)
                           A[!call(Impl) U call(Alg)]   **call(Bkgnd)** → **Aspects**

                           !call(Impl)
                           A[!call(Impl) U call(Alg)]   **ret(Bkgnd)**

                           call(Alg)
true*   !call(Impl)
                           A[!call(Impl) U call(Alg)]   **call(Alg)**

                           !call(Impl)   **ret(Alg)**

true*   call(Impl)   **call(Impl)**

             !call(Impl)   **ret(Impl)**

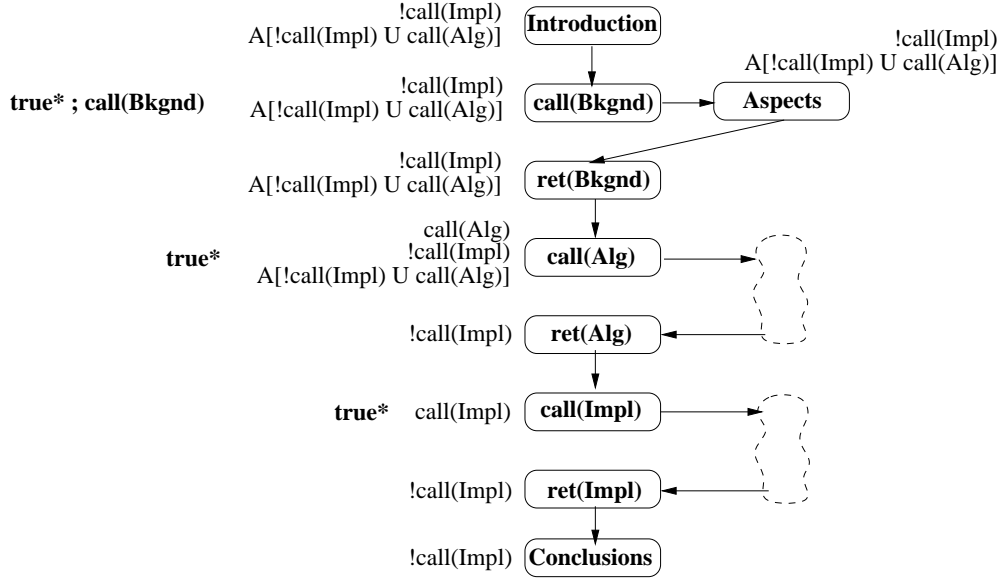   !call(Impl)   **Conclusions**

Figure 5: Program Annotated with CTL Labels and Stack Contents

program's states.[5]

The heart of advice verification is as follows. Suppose the advice for the model checking slides (shown in figure 3) is applied as after advice at the pointcut of PCD P. Note that the advice in isolation does not satisfy the property because the advice does not show the Alg slides, but that the advised program continues to do so. Our algorithm extends the advice with states *in* and *out*, seeds *out* with the labels of *After*$_2$ (in this case, the state labeled ret(Bkgnd)) from the interface and seeds *in* with the propositions of *After*$_1$ from the interface; figure 6 shows the resulting state machine. It then verifies that each label on the *After*$_1$ state holds on the *in* state of the advice, assuming that the labels for *After*$_2$ hold on the *out* state (checking source labels against copied sink labels matches the backward propagation inherent in the CTL model-checking algorithm [13]). If all of these checks pass, the program with advice will satisfy the property. If a check fails, the advice may violate that property (if the property depended on the violated label); the algorithm uses the location of the pointcut to report the potential violation of the program's behavior at that locus and by the corresponding aspect. Before advice is treated analogously using the states *Before*$_1$ and *Before*$_2$ (as illustrated in figure 1 (left)). Around advice is addressed in section 4.4.

Observe that the algorithm verifies the advice state machine without traversing the program's state machine (though it may need to traverse fragments of the program source referred to by the advice). Ideally, we would like to show that this process is nevertheless sufficient: if this check succeeds, so would verifying the program with the advice explicitly spliced in. Unfortunately, this is not (yet) true!

---

[5]While there is one interface for each state in the pointcut, in most cases these interfaces will have logically related formulas (because in general, the advice will tend to apply in similar circumstances). In principle, we could employ deductive reasoning over temporal logic to shrink the number of distinct interfaces.
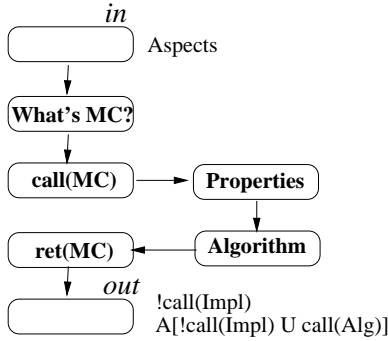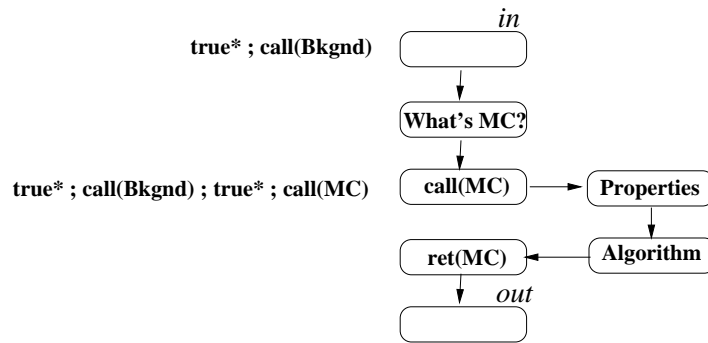
Figure 6: Advice Prepared for Verification

Figure 7: Detecting Joinpoints in Advice

4.3 Cascading Advice

To see the problem, recall PCD Q. The main program invokes call(Bkgnd); the applied advice invokes call(MC). Therefore, the program and advice *combine* to trigger the PCD. Indeed, an actual AOP implementation would detect this condition. Our technique relies on having accurate interfaces for all states at which advice might apply; this means that we must generate interfaces for states in the advice that trigger existing PCDs.

While it is clear we must compute the stacks that could exist at each state in the advice (for the same reason we did with the main program), it is easy to do this incorrectly. If we compute the stacks in the advice starting with the empty stack, we would still fail to notice the enabling of PCD Q. Instead, we need to initialize the stacks *with their contents at the point of applying the advice*, which is information we must record in the interface. In this instance, the stack at the entry to the advice contains a call to Bkgnd. The annotated advice is shown in figure 7. It shows that PCD Q is satisfied by a combination of the main program and the first advice in the indicated state, resulting in a second generated interface. If an advice associated with Q violates a property, our algorithm can report the violation in terms of both aspects, resulting in a helpful diagnostic.
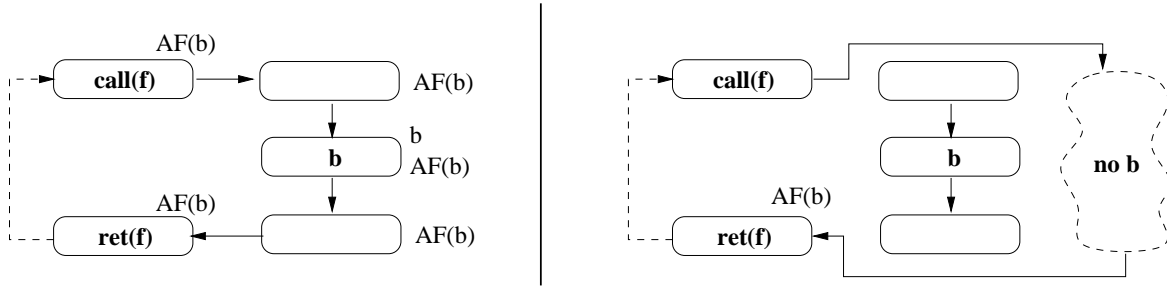
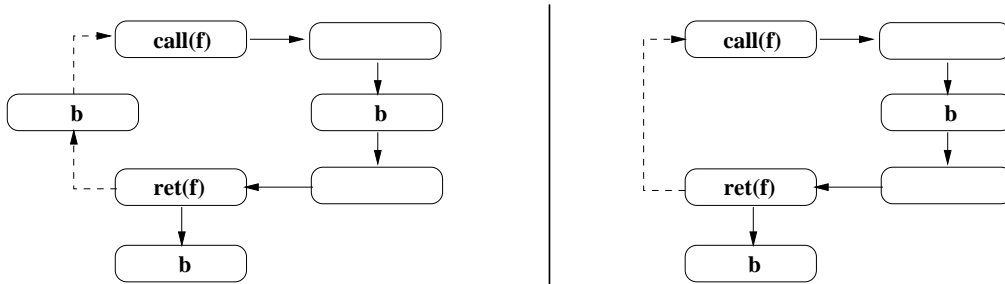Figure 8: How Around Advice can Violate Properties

Figure 9: When Around Advice Violates Properties

4.4 Around Advice

Verifying around advice modularly is more subtle. Consider the program in figure 8 (left), which shows labels ascribed to states by model checking the formula $AF(b)$. (The dashed lines show control paths along which some states have potentially been suppressed.) The program in figure 8 (right) results from applying an around advice without proceed to the original program: as shown, around advice without proceed can bypass states from the original program. Our verification process as described so far will copy the $AF(b)$ label from the return state of the left figure to the return state of the right figure and attempt to confirm the $AF(b)$ label on the call state. This check succeeds, even though the call state on the right clearly violates $AF(b)$. To avoid this form of *circular reasoning*, we need to refine the verification process when states can be bypassed (which by construction can only occur with around advice without proceed).

Understanding our refined algorithm requires some intuition about why the problem arises in the first place. Consider the program in figure 9 (left). The original algorithm suffices for this program because the $AF(b)$ label on the return state does not depend on the occurrence of b in the bypassed states. This suggests that the problem lies in the location of the b: in figure 8 (left) the labels on the call and return states rely on the b label on the same state, while in figure 9 (left) different instances of b justify the call and return labels. The issue, however, is deeper than whether call and return depend on the same b states. In the program in figure 9 (right), the instance of b between the call and return helps justify $AF(b)$ on both the call and return states, but the original algorithm suffices. The real issue is whether the $AF(b)$ label on the return state depends on the $AF(b)$ label on the call state: if it does, then because advice can elide paths, the original technique may be unsound.

14

We could attempt to augment the model checker to track formula dependencies, but this would be overkill because we generally don't need dependency information on all subformulas. The original algorithm is only unsound when (1) the same AU or EU formula labels both a call state and its corresponding return (recall that AF is a special case of AU), (2) the return state label depends on the call state label, and (3) the applied advice is around without proceed.[6] Items 1 and 3 are easy to check. For item 2, we can write a CTL formula to check whether the return state label depends on a call state label. For the label $AF(b)$ in our example, checking $A[!call(f) \; U \; b]$ at the return state returns true when the return state label does not depend on a call state label. If the return state label does depend on a call label, then the label is not copied to the return state during modular verification. This forces the advice to justify the label on the call state, which the bypassed states must have done in the original program.

Now suppose we are verifying around advice that does invoke proceed. Say the advice advises an application of function f. The body of f in the source program has already been traversed by the model checker at the point of application of the advice. Since this is the same code that will execute at the proceed-resume states, it is tempting to reuse this verification effort by adopting the labels already in the program, thereby avoiding re-verification of the body of f.

Reusing the labels on this copy of f is, unfortunately, not necessarily sound. The fragment of the advice that appears after resumption may invalidate some of the labels that are on the states of f. (For instance, since we have added a new path, a label of the form $AF(\phi)$ may no longer hold.) For this reason, we currently replace the proceed-resume states with a copy of f and repeat verification on f's body.

5 Approximating Program Stacks, Informally

Our technique relies on the ability to predict the contents of the stack at each state in the program. Our technique is sound so long as the stack analysis locates all states at which the stack could match a PCD. If the stack analysis over-approximates this set of states, our technique could report property violations that could not occur in practice. We discuss two different techniques for computing the stack's content.

5.1 Using Automata

One approach observes that PCDs are regular expressions, and hence can be compiled to regular automata. (An extended version of this is employed by Sereni and de Moor [56].) Taking the cross-product of a program and a PCD automaton would identify states at which the program could satisfy the PCD. Two subtleties arise in building a regular automaton for a PCD for this purpose. Consider the PCD $true^*; call(f); call(g)$: as a standard regular expression, this expects $call(g)$ to occur in the state immediately following the one satisfying $call(f)$. Recall, however, that PCDs describe call stacks, not traces of all program states. As a result, the automaton needs to stutter until the next call state. Furthermore, the PCD does not (by design) account for return states; a $call(g)$ state that occurred after a $ret(f)$ state should not satisfy the given PCD. The compilation algorithm would need to address both concerns.

Even after addressing these concerns, the automaton-based approach suffers from two potential drawbacks. First, this technique will overapproximate the set of identified states on most programs as it uses a regular automaton (for the

---

[6]The Until operators are the only ones that need this check because they are the only ones that depend on *eventually* reaching a state that satisfies a formula in a variable number of steps.

PCD) to approximate a context-free language (the actual set of stacks). Second, it has the potential to be too expensive on PCDs that use Kleene-star and disjunction operators. In the worst case, cross-product constructions grow the size of one machine by a multiplicative factor in the other. If a PCD is just a concatenation of call statements, the PCD and program transitions will align deterministically, resulting in no growth in the program state space. Growth occurs when the PCD automaton is non-deterministic, as can happen with the Kleene-star and disjunction operators. However, we might be able to generate fewer interfaces by being able to track when multiple PCDs are triggered simultaneously; this requires constructing a cross-product of several PCDs with the program. Experimental analysis would help determine the extent to which these issues would be problematic in practice.

5.2 Using CTL

As an alternative (the one we formalize in this paper), we can exploit the CTL model-checker to track stack contents. The PCD labeled Q in section 4 resembles the CTL formula EF(call(Bkgnd) $\wedge$ EF(call(MC))). Can we use the model checker to find those states that satisfy this formula, and hence the PCD? This approach would not properly identify pointcut states because the formula would be true at states at the *start of a path that could reach* a pointcut state, rather than at the pointcut state itself—a reflection of the future-time nature of CTL. Capturing the pointcut states requires a way to look at the past from a given state and ask whether it reflects the correct sequence of calls in progress. (Past-time CTL could handle this, but would either require a separate algorithm or incur an exponential blow-up on translation into regular CTL [44].)

A different approach results in a cleaner algorithm and more accurate joinpoint identification. We can look at the past as follows. First, we *reverse* each of the edges in the program's state machine. Second, we employ a CTL formula that matches the stack's contents in reverse. For the PCD Q, this formula would be

$$\text{call(MC)} \wedge \text{EF(call(Bkgnd))}$$

This formula should label exactly those states in the reversed machine with a call(MC) label and for which the stack has a form that matches the PCD. It is crucial to note that this model-checking run cannot "fail": failure to assign a label to a state signifies only that the state is not a member of a pointcut. (That is, we are exploiting the model checker's traversal power to do something quite distinct from verification.)

In its current form, this proposal overapproximates the set of joinpoint states. Consider a program that contains functions f and h; the program invokes f and h in sequence, with f in turn also invoking h (as shown on the left in figure 10). Assume we were matching this program against a PCD similar in shape to Q, namely call(f) $\wedge$ EF(call(h)). This program should match the PCD only once (the call to h within the dynamic extent of f). The formula is, however, true at two states of the reversed machine. The error here is a failure to handle return states. (Put otherwise, this formula cannot distinguish between sequential and nested calls.) In this case, the formula needs to check that f does not return on the path from the invocation of h to that of f. While we could patch the formula to check for return states, the resulting formula would be cumbersome and could still overapproximate the set of joinpoints, as CTL captures only regular sets.

Our solution to this problem lies in constructing the reversed state machine differently. On a reverse path from a given state s, subpaths that traverse the program between a return state and its corresponding call explore states that have been popped from the stack before control arrives at s. The traversal should therefore "bypass" matching call and
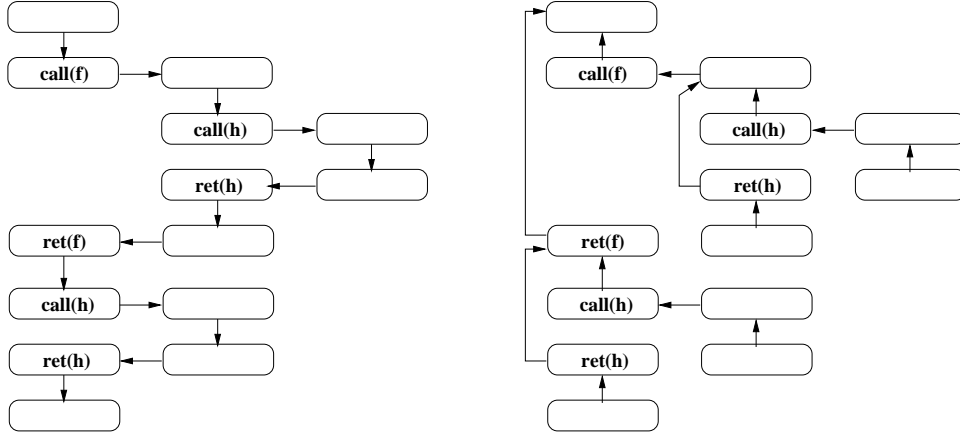
Figure 10: A Program and its Reverse-Bypass Version

return states and the paths betwixt, visiting only call states whose returns have not yet occurred. Bypassing states is straightforward: any edge in the reverse graph that goes out from a return state is redirected to point to the successors (in the reversed graph) of the corresponding call state. For instance, the graph on the right of figure 10 shows a version of the graph on the left with the edges reversed and calls bypassed. This construction effectively removes the called function from the stack at the return state. Formulas checked against this state machine need not match calls with returns because completed function invocations have been elided from the paths the model checker will traverse. This addresses the mismatch between the context-free stack and the regular PCD, and provides exact joinpoint identification (relative to the inlining depth and assuming that every function can terminate on some path).

Reversing the machine takes time linear in its size (the stack tracks the bypass states). The formula is linear in the size of the PCD. The model checker takes time linear in the product of the sizes of the state machine and the formula. Since the size of the formula can usually be bounded by a small constant (because PCDs tend to be small, and the corresponding formula as constructed in section 6.1 is linear in the size of the PCD), we can usually determine the pointcut states in linear time. We note that the reverse-bypass construction could, if desired, be implemented easily with symbolic representations [13]: reversing the edges corresponds to swapping the current- and next-state variables (or their interpretation) in the BDD for the transition relation, projection identifies the edges to add and delete (for bypassing), and actual addition and deletion are just BDD-or and -and, respectively.[7]

This CTL-based approach cannot be used if the PCD language is extended to all regular expressions because not all regular expressions can be captured in CTL. Consider the PCD $(\text{true}; \text{call}(f))^*$ which checks whether every even call is a call to f. Wolper proved that the regular expression "p is true in every even state" is not expressible in LTL [67]. The proof that Wolper's result extends to the subset of CTL used here is beyond the scope of this work. Nonetheless, Wolper's result motivated the restriction of Kleene-stars in our PCD language to a useful subset that is expressible within CTL.

---

[7]All of these operations are standard in the APIs for BDD packages.

6 The Verification Process, Formally

This section formalizes the intuitive description of our algorithm and its data structures. Specifically, we define how to locate states that satisfy PCDs, define interfaces, and give the algorithms for generating interfaces, verifying advice, and generating new interfaces from advice. Section 7 proves the soundness of these details.

6.1 Locating Joinpoints

To locate joinpoints, our algorithm constructs the reverse-bypass version of a state machine, compiles PCDs to CTL formulas, and uses the model checker to analyze the reversed machine against the formulas.

Constructing a reverse-bypass version of a state machine requires two steps: reversing transitions, and bypassing paths from return states to their corresponding call states (thus leaving only those paths that correspond to actual stack traces). The following formal definition constructs the transition relation for the reverse-bypass machine from the transition relation of the original machine and the relation T_{cr} that stores the correspondence between call and return states for the original machine; the latter is used to locate the target states for the bypassing transitions.

Algorithm 3 (The Reverse-Bypass Construction) Let M be a state machine with transition relation T and call-return relation T_{cr}. The *reverse bypass of M*, denoted M_{RB}, is the state machine with the same components as M other than T, which is replaced with transition relation T_{RB}, defined as follows (recall that S_{rtn} and S_{call} are the sets of call and return states in M):

$$
\begin{aligned}
T_{\mathrm{RB}} = &\{(s_2, s_1) \mid s_2 \notin S_{\mathrm{rtn}} \land (s_1, s_2) \in T\} \cup \\
&\{(s_r, s) \mid s_r \in S_{\mathrm{rtn}} \land \exists s_c \in S_{\mathrm{call}} \mid (s_c, s_r) \in T_{\mathrm{cr}} \land (s, s_c) \in T\}
\end{aligned}
$$

We now turn to compiling PCDs into CTL formulas. This translation is cleaner if the PCDs are in a form where the ; operators are distributed over the | and & operators.

Definition 8 A PCD is in *concatenation style* iff it does not contain the | and & operators. A PCD in which the ; operators are distributed over the | and & operators maximally (resulting in no | or & operator within a ; operator) is in *concatenation normal form* (called simply *concatenation form* in the rest of this paper).

By these definitions, a PCD in concatenation normal form consists of a boolean expression (using | and &) over concatenation-style PCDs.

Given a PCD, our translation strategy reverses the order of concatenated terms, appends a "bottom of stack" marker, converts to concatenation-normal form, then translates the resulting PCD into CTL. The bottom-of-stack marker provides a base case for the translation. Let $\circ$ be the bottom-of-stack marker. Given a PCD d, d^{-1} denotes its reversal (reversing the order of all concatenated terms, but making no changes to the terms themselves). The term $d^{-1}; \circ$ denotes the reversal with the appended marker. Let PCD2CNF be a function that converts a PCD into concatenation normal form. Given a PCD d, its corresponding CTL *identifier* is PCD2CTL(PCD2CNF($d^{-1}; \circ$)) where PCD2CTL is shown in figure 11. This algorithm assumes every call state has the label call, that the source state of the main function has the labels call and call(main) and that the sink state of the main function has the label return.

PCD2CTL$(expr) =$

case $expr$ **of**

$\quad \circ \qquad\ = \mathsf{call}(\mathsf{main})$

$\quad a \qquad\ = a$

$\quad e_1 \wedge e_2\ = \text{PCD2CTL}(e_1) \wedge \text{PCD2CTL}(e_2)$

$\quad e_1 \vee e_2\ = \text{PCD2CTL}(e_1) \vee \text{PCD2CTL}(e_2)$

$\quad e^*; p \qquad = (\mathsf{call} \wedge \mathsf{E}[(\mathsf{call} \rightarrow \text{PCD2CTL}(e))\ \mathsf{U}\ \text{PCD2CTL}(p)])$

$\quad e; p \qquad = \mathsf{call} \wedge \text{PCD2CTL}(e) \wedge \mathsf{EX}(\mathsf{E}[!\mathsf{call}\ \mathsf{U}\ \text{PCD2CTL}(p)])$

$\quad (de) \qquad = (\text{PCD2CTL}(de))$

$\quad d_1 \mid d_2\ = \text{PCD2CTL}(d_1) \vee \text{PCD2CTL}(d_2)$

$\quad d_1\ \&\ d_2 = \text{PCD2CTL}(d_1) \wedge \text{PCD2CTL}(d_2)$

where a is a pointcut atom, e, e_1 and e_2 are pointcut elements, d_1 and d_2 are concatenation-form PCDs, de is a PCD or pointcut element, p is a concatenation-style PCD and $expr$ is a concatenation-form PCD with appended end-of-stack marker. Appending the end-of-stack marker implies that no case is needed for a standalone pointcut element.

Figure 11: The PCD2CTL Procedure

The translation from PCDs into CTL formulas is a bit more subtle than our intuitive example suggests because the PCD is an expression over call states while the formula must be an expression over states in general. The use of EU rather than EF in the compilation highlights this difference.

Definition 9 Let M be a state machine and d be a PCD. State s in M is a *joinpoint state* for d if $M_{\mathrm{RB}}, s \models$ PCD2CTL$(d^{-1}; \circ)$.

6.2 Generating Interfaces

The informal description of our algorithm motivated the information that an interface must cache to reconstruct the state of the verification process and the shape of the stack for a particular PCD. At each joinpoint state $Before_1$ that satisfies the PCD, the labels on the states $Before_1$, $Before_2$, $After_1$ and $After_2$ as shown in figure 1 (left) capture the state of the verification process ($Around_1$ is the same as $Before_1$ and $Around_2$ is $After_2$). An additional set of formulas (called U_{check}) contains labels on $After_2$ that may be violated by around advice. The labels ascribed to $Before_1$ while analyzing the CTL form of the PCD capture the shape of the stack. We do not need to store the stack shape at state $After_1$ because our stack discipline assumptions guarantee that the stack shape is the same at $Before_1$ and $After_1$. We could eliminate some of this information if we knew the advice type in advance; here, we present only the general case.

Definition 10 A *joinpoint interface* is a tuple of the form $\langle d, \varphi, L_1, L_2, L_3, L_4, U_{\mathrm{check}}, stacks \rangle$ where d is a PCD, φ is a CTL formula, and the remaining components are sets of CTL formulas.

This definition of an interface does not include the actual state at which the interface is generated. Storing this state would be useful for tasks such as counterexample generation. We refrain from doing so in this paper to reduce notation and to focus on the heart of the algorithms and their correctness.

Joinpoint interfaces can be generated automatically from a program, PCD, and property using a CTL model checker as follows:

Algorithm 4 (Generating Joinpoint Interfaces) Given a program P, a concatenation-form PCD d and CTL formula φ, the set of all joinpoint interfaces is generated by the following steps:

1. Use a CTL model checker to verify φ against P. Let *lab* be the function from states in P to the labels (CTL formulas) that the model checker assigned to states.

2. Use a CTL model checker to analyze $\varphi_d = \text{PCD2CTL}(d^{-1}; \circ)$ against P_{RB}. Let lab_{RB} be the function from states in P (same as the states in P_{RB}) to the labels (CTL formulas) that the model checker assigned to states. Let JP be the set of states s such that $\varphi_d \in lab_{\mathrm{RB}}(s)$.

3. For each joinpoint state $Before_1$ in JP, create a joinpoint interface

$$\langle d, \varphi, lab(Before_1), lab(After_1), lab(Before_2), lab(After_2), U_{\mathrm{check}}, stacks \rangle$$

where

- $Before_2$ is the state such that $(Before_1, Before_2)$ is in T of P,
- $After_2$ is the state such that $(Before_1, After_2)$ is in T_{cr} of P,
- $After_1$ is the state such that $(After_1, After_2)$ is in T of P,
- U_{check} is defined below, and
- $stacks$ is $lab_{\mathrm{RB}}(Before_1)$.

U_{check} is defined as follows. Let Q be a meta-variable representing either the A or E path quantifier, and f be the function being called at $Before_1$. The set U_{check} consists of all formulas of the form $Q[\phi \mathrel{\mathsf{U}} \psi]$ that are in both $lab(Before_1)$ and $lab(After_2)$ and for which the property $Q[(!\mathsf{call}(f) \wedge \phi) \mathrel{\mathsf{U}} \psi]$ fails to verify at $After_2$ in P.

6.3   Analyzing Advice

Joinpoint interfaces identify those properties that advice must satisfy in order to preserve desired program properties. As aspects advise a verified program, two steps must occur. First, the aspects must be checked against the interfaces (for whether they preserve properties). Second, interfaces must be generated for all new joinpoints that are triggered by the combination of the program and the advice: this allows future applications of advice to be checked against all relevant joinpoints in the advised program. We present these two steps as separate algorithms, but note that in practice they would operate in tandem: an aspects compiler generally checks whether new advice triggers new joinpoints for aspects that have already advised the program; if so, the aspects are advised at the new joinpoints [31]. To ensure soundness, each application of advice requires the use of both of the following algorithms until no additional advising is required.

We employ the following algorithm at every joinpoint interface that was generated for the PCD in the advice to be applied:

Algorithm 5 (Advice Verification) Given an aspect $\langle d, t, A \rangle$ to verify against a joinpoint interface

$$\langle d, \varphi, lab(Before_1), lab(After_1), lab(Before_2), lab(After_2), U_{\text{check}}, stacks \rangle$$

(for the same PCD, d, as in the aspect):

1. (Prepare the aspect) Add new states called *in* and *out* to A such that *in* has an outgoing transition to the start state S_{src} of A and *out* has an incoming transition from the terminal state S_{sink} of A. No other edges should enter or leave *in* and *out*.

2. (Check the advice)

   - If $t = $ before, copy the labels in $lab(Before_2)$ to *out*, copy the propositions from $lab(Before_1)$ to *in*, and verify each label in $lab(Before_1)$ at *in*.

   - If $t = $ after, copy the labels in $lab(After_2)$ to *out*, copy the propositions from $lab(After_1)$ to *in*, and verify each label in $lab(After_1)$ at *in*.

   - If $t = $ around and A has no proceed states, copy the labels in $lab(After_2) - U_{\text{check}}$ to *out*. Copy the propositions from $lab(Before_1)$ to *in* and verify all formulas in $lab(Before_1)$ at *in*.

3. (Report result) If all checks succeed, report the advice as preserving the program's properties; otherwise report that φ may fail.

Note that a check failing does not guarantee that the advice violates a program property because the algorithm is not complete. Intuitively, the incompleteness arises because the labels that we check against the advice are sufficient, but not necessary, to preserve properties; disjuncts are a simple example of potentially unnecessary labels. A more sophisticated analysis would be required to determine both which labels are necessary and which labels could be satisfied through alternate formulas.

Algorithm 6 (Generate Interfaces from Advice) Given an aspect $\langle d, t, A \rangle$ and a joinpoint interface

$$\langle d, \varphi, lab(Before_1), lab(After_1), lab(Before_2), lab(After_2), U_{\text{check}}, stacks \rangle$$

for which algorithm 5 has been run (so the model checker has ascribed labels to states in the advice), seed *in* in A_{RB} with the formulas from *stacks*, then reuse steps 2 and 3 of algorithm 4, replacing P with A.

Using algorithm 6, the set of joinpoint interfaces for a program grows as advice is applied to the program. Each aspect applied to the system must be analyzed against each of the interfaces corresponding to that aspect's PCD.

7 Soundness

Our approach is sound if all properties declared true using the algorithms of section 6 would be true if verified against the entire composed program. To prove this, we must show that our technique locates all joinpoints, and that labels ascribed to states in the modular algorithm would also be ascribed by model checking the entire composed program. This section formally states and proves these correctness criteria. All these statements and proofs are stated relative to the state machine model of the program, but the model may not precisely capture the original program due to inlining and other approximations made during model generation.

7.1 Joinpoint Identification

Definition 11 A *stack* is a sequence of labels of the form call(f) (where f is a function name other than main). Given a finite path Π through a state machine, the *stack trace* corresponding to Π, denoted $stack(\Pi)$, is the stack obtained by starting with an empty stack and traversing Π from initial to final state, pushing each call(f) label (other than call(main)) and popping at each return label along Π (other than the sink state of the main function).

This definition excludes main from the call stack because our proofs depend on identifying states that terminate paths whose stack traces are in the languages defined by PCDs. The PCD language treats the bottom of stack (represented by call(main) as implicit (call(main) is not in the PCD language). The restriction against popping the final return label prevents a pop with no corresponding push.

Joinpoint identification is sound if it locates every state at which the program stack can satisfy a PCD. Soundness allows the technique to over-approximate the set of joinpoints; ideally, the technique should only identify states that satisfy PCDs. Under the assumptions of stack discipline and every function having *some* terminating path, our CTL-based technique satisfies both requirements. We prove each direction in a separate theorem.

The proof of soundness must show that every state that triggers a PCD (on a path that satisfies stack discipline) satisfies the CTL identifier for the PCD in the reverse-bypass machine. The CTL formulas for concatenation-style PCDs are essentially chains of nested EU operators, where each EU expression detects a prefix of the desired stack contents. Intuitively, each call state in the path that satisfies a prefix of a concatenation-style PCD should satisfy the corresponding EU formula that detects that stack in the reversed program. This claim is the heart of the proof. Given a concatenation-style PCD and a path, we construct a function that stores which states satisfy prefixes of the PCD along the path; the proof will show that the same states satisfy the formulas for those prefixes. This function is formally defined as follows:

Definition 12 Let Π be a path through a program and d be a concatenation-style PCD such that $stack(\Pi)$ is in the language of d. Let d_k denote the prefix of d with k concatenated terms. Define the *stack witness function SW* from prefixes of d to sets of states in Π such that $SW(d_k)$ is the set of all call states s in Π for which the stack trace of the prefix of Π up to and including s is in the language of d_k. For the empty prefix d_0, define $SW(d_0)$ to be the set containing the initial state of Π (which is also labeled with call by construction–our proofs depend on the invariant that call labels every state in a set in the codomain of SW). For a concatenation-form PCD D, we define SW as described for all maximal-length concatenation-style PCDs within D.

Example: Given PCD call(f); true*; call(g) and a path s_0, s_1, s_2, s_3 where s_1 has label call(f), s_2 has label call(g) and s_3 has label call(h), $SW(\text{call}(f)) = \{s_1\}$, $SW(\text{call}(f); \text{true}^*) = \{s_1, s_2, s_3\}$ and $SW(\text{call}(f); \text{true}^*; \text{call}(g)) = \{s2\}$.

Observation 1 Note that if $stack(\Pi)$ is in the language of concatenation-style PCD d, then SW must be non-empty for all prefixes of d. This is obvious for prefixes ending in non-starred atoms, as each non-starred atom must appear somewhere in the stack. For starred atoms, any state which satisfies the previous prefix will also satisfy the prefix ending in a starred atom (because the starred atom appears zero times). Our soundness proof relies on this observation.

Given that our joinpoint identification method operates on the reverse-bypass version of the original program, our soundness proof must show that the paths that witness stack traces (via the formulas) in the reverse-bypass version

reflect paths in the original program. The next lemma proves a condition for when a path between two call states in the reverse-bypass program also exists (in reverse direction) in the original program.

Lemma 1 *Let s and s' be* call *states in a program P such that there is a path from s to s' that obeys stack discipline. There is a path from s' to s in P_{RB} unless the* return *state corresponding to s occurs between s and s' on all paths in P.*

Proof: The transition relation in the reverse-bypass construction (algorithm 3) changes only transitions that would otherwise start from a return state. As a result, if there is a path with no return state between s and s', then there must be a path in P_{RB} from s' to s. Assume that every path from s to s' in P contains a return state; let Π be such a path. Following the assumptions of the lemma statement, we assume that none of the return states on Π is for the same function as the call at s. The reverse-bypass construction removes paths from each return to its corresponding call. For s to become unreachable from s' in P_{RB} via bypassing, s would have to have occurred between corresponding call and return states. Since s is itself a call state, stack discipline would require that the return state for s also occur between the states that resulted in the bypass. This violates the assumptions of the lemma, so the lemma holds.
□

We now present the theorems that joinpoint identification is sound and complete using our CTL-based approach. The main theorems prove soundness and completeness for concatenation-style PCDs. The following lemma argues that these theorems extend to concatenation-form PCDs (which in turn cover all PCDs).

Lemma 2 (Concatenation-Style Suffices) *Let D be a concatenation-form PCD and let $d_1, \ldots, d_k$ be the maximal-length concatenation-style PCDs within D. An algorithm for identifying joinpoints that is sound (resp. complete) for all d_i's is also sound (resp. complete) for D.*

Proof: By definition, a concatenation-form PCD is a boolean logic expression over concatenation-style PCDs. The boolean operators in PCDs have their standard semantics, so the lemma follows from the soundness and completeness of propositional boolean logic.

Theorem 1 (Joinpoint Identification Sound) *Let s be a state in program P, d be a concatenation-style PCD, and φ_d be PCD2CTL($d^{-1}; \circ$). If there exists a path Π in P from the initial state to s such that* stack(Π) *is in the language of d and Π follows stack discipline then $P_{\mathrm{RB}}, s \models \varphi_d$.*

Proof: Let Π be a path in P such that $stack(\Pi)$ is in the language of d. Let SW be the stack witness function for Π and d (definition 12). We claim that for all prefixes d' of d and all states $s' \in SW(d')$, $P_{\mathrm{RB}}, s' \models$ PCD2CTL($d'^{-1}; \circ$). The desired result is a corollary of this claim. We prove the claim by induction on the length of the prefix. Let d_i denote the prefix of length i and φ_{d_i} denote PCD2CTL($d_i^{-1}; \circ$). In the base case, the prefix (d_0) contains no atoms from d; it therefore contains only the end of stack marker $\circ$. By definition, $SW(d_0)$ contains only the start state of P. PCD2CTL($\circ$) is the formula call(main), which labels only the initial state, so the base case holds.

For the inductive case, assume that the claim holds for all prefixes up through d_{k-1}. We must prove that it holds for d_k. Let s' be a state in $SW(d_k)$. By the PCD grammar, d_k (which is concatenation-style) has one of two forms:

- If $d_k = d_{k-1}; e$, then φ_{d_k} is call $\wedge$ PCD2CTL(e) $\wedge$ EX(E[!call U PCD2CTL($d_{k-1}^{-1}; \circ$)]). By the definition of SW, s' satisfies call; it must also satisfy PCD2CTL(e) because the prefix of Π ending at s' includes s' and the stack

23

trace of Π is in the language of d_k. The $\mathsf{EX}(\ldots)$ portion of the formula is satisfied by finding a state s'' in $SW(d_{k-1})$ such that there is a path from s' to s'' in P_{RB} with no call states between s'' and s'. Let s'' be the state in $SW(d_{k-1})$ closest to s' on Π; by the definition of SW, state s' must be the first state after s'' on Π to satisfy d_k. Therefore, any call states between s'' and s' in Π must have been popped from the stack by a matching return state. The reverse-bypass construction elides these call states, so we only need to show that P_{RB} contains a path from s' to s''. Since the call at s'' is on the stack at s', the return state corresponding to s'' must occur after s'; lemma 1 therefore guarantees that there is a path from s'' to s'.

- If $d_k = d_{k-1}; e^*$, then φ_{d_k} is $(\text{call} \wedge \mathsf{E}[(\text{call} \rightarrow \text{PCD2CTL}(e)) \ \mathsf{U} \ \text{PCD2CTL}(d_{k-1}^{-1}; \circ)])$. If s' is in $SW(d_{k-1})$, then s' satisfies $\varphi_{d_{k-1}}(= \text{PCD2CTL}(d_{k-1}^{-1}; \circ))$ by the inductive hypothesis. By the semantics of EU and the fact that every state in a set in the codomain of SW has a call label, s' satisfies $\varphi_{d_{k-1}}$ so the theorem holds. Assume s' is not in $SW(d_{k-1})$. The difference between this case and the one for the other form of d_k is that, in this case, $SW(d_k)$ could have multiple elements (due to the * on e). Using a similar argument as in the previous case, we can build a path from the element of $SW(d_k)$ that is closest to the initial state in Π that satisfies φ_{d_k}. We can complete the proof for the remaining states in $SW(d_k)$ inductively, again using a similar argument as in the previous case to construct the path between them. The theorem therefore holds.

$\square$

Theorem 2 (Joinpoint Identification Accurate) *Let s be a state in program P, d be a concatenation-style* PCD, *and φ_d be* PCD2CTL$(d^{-1}; \circ)$. *If $P_{\text{RB}}, s \models \varphi_d$ and all functions terminate along some path, then there exists a path Π in P from the initial state to s such that* stack(Π) *is in the language of d and Π follows stack discipline.*

Proof: Let s be a state such that $P_{\text{RB}}, s \models \varphi_d$. By the form of φ_d (nested EUs), satisfying φ_d requires the existence of a path from s to s_0 along which the subformulas of φ_d are satisfied; call this path Π_{RB}. Reversing the direction of the transitions in Π_{RB} yields a path Π_{fwd} from s_0 to s; note that while the transitions in Π_{fwd} are in the same direction as transitions in P, Π_{fwd} is not itself a path in P (due to the states that were bypassed during the construction of P_{RB}). In order to prove the existence of a path Π as required in the theorem, we will first prove that the stack trace of Π_{fwd} is in the language of d, then we will construct Π from Π_{fwd} in a manner that preserves the stack trace from Π_{fwd}.

Computing the stack trace of Π_{fwd} accurately requires us to first remove all return labels from the states in Π_{fwd}. The reverse-bypass construction guarantees that no call state that matches a return state in Π_{fwd} can itself lie in Π_{fwd}. Removing the return labels prevents the stack trace computation from popping non-existent calls, while retaining all un-returned calls. Removing these labels would not have affected the truth of φ_d on Π_{RB}, as that formula never refers to a return label.

Let d_i be the prefix of d containing i concatenated terms. We claim that for all prefixes d_i of d, if φ_{d_i} labels s' in Π_{RB}, then the stack trace of the prefix of Π_{fwd} up to and including s' is in the language of d_i. Our desired result that the stack trace of Π_{fwd} is in the language of d is a corollary of this claim. We prove the claim inductively on i. In the base case, i is zero: d_0 is empty and φ_{d_0} is call(main), which is true only at the start state of P. The start state has the empty stack trace (since stack traces ignore the call(main) label by definition), which is in the language of the empty prefix, so the base case holds.

For the inductive case, assume the claim holds for prefixes up to and including d_{k-1}. We must prove it holds for d_k. Let s' be a state in Π_{RB} such that φ_{d_k} is true at s'. Unless d_k is the empty prefix, PCD2CTL$(d_k^{-1}; \circ)$ requires some

state s'' in Π_{RB} to satisfy PCD2CTL(d_{k-1}^{-1}; $\circ$). By the inductive hypothesis, the stack trace up to s'' is in the language of d_{k-1}. The PCD d_k has one of two forms:

- If $d_k = d_{k-1}; e$, then PCD2CTL(d_k^{-1}; $\circ$) requires that there be no call state between s' and s'' (by virtue of the expression E[!call U ...]); this portion of the path therefore cannot push any calls onto the stack. In addition, it cannot pop any calls because Π_{fwd} has no return labels by construction. The stack trace at state s' therefore appends the call at s' onto the stack trace at s'', resulting in a stack which is in the language of d_k.

- If $d_k = d_{k-1}; e^*$, then the argument depends on whether s' satisfies PCD2CTL(d_{k-1}^{-1}; $\circ$). If it does, then $s' = s''$ and by the inductive hypothesis the stack trace on Π_{fwd} up to s' satisfies d_{k-1}. The same stack trace must satisfy d_k by interpreting the * as zero occurrences, so the claim holds. If it does not, s'' is different from s' and the argument is similar to that in the non-starred case, except all of the call labels pushed on the stack satisfy e; the resulting stack is in the language of d_k since the * can match all of the occurrences.

Having established that the stack trace of Π_{fwd} is in the language of d, we now need to construct a path Π in P with the same stack trace as Π_{fwd}. First, restore the return labels on the states in Π_{fwd}. Next, undo the bypass step of the reverse-bypass algorithm: for every return state s_r in Π_{fwd} replace the transition from its predecessor s_1 (in Π_{fwd}) to s_r with a path from s_1 to s_r in P that has matching call and return states. (Such a path must exist because we assumed that every function has some path along which it terminates and P has matching call and return states by construction.) The matching states imply that the added states cannot change the stack traces at states in Π that were in Π_{fwd} so the theorem holds.

$\qquad\square$

Theorem 3 (Identifying Joinpoints in Aspects) *Let s_a be a state in A and let d be a PCD. $A_{\mathrm{RB}}, s_a \models$ PCD2CTL(d^{-1}; $\circ$) (from algorithm 6) iff $(P{\cdot}A)_{\mathrm{RB}}, s_a \models$ PCD2CTL(d^{-1}; $\circ$).*

Proof: Theorems 1 and 2 establish that the CTL formulas for PCDs accurately reflect the stack contents. Algorithm 6 ascribes these stack labels to the initial state of an aspect. The stack discipline assumption implies that an aspect cannot affect the advised program's stack. The correlation between the formulas and the stacks therefore extends into the aspects by a repetition of the argument in the preceding theorems with a change only to the base case (to initialize the stacks with the contents from P, rather than the empty stack).

7.2 Modular Verification

In the following theorem statements, let P be a program, A be advice to be applied to P, and $P{\cdot}A$ be the composed program (constructed using algorithm 2).

Theorem 4 (Program Labels Accurate) *Let φ be a CTL formula. For all states s in both P and $P{\cdot}A$, $(P{\cdot}A), s \models \varphi$ if $P, s \models \varphi$ and the advice verification algorithm reports that all labels are preserved. For all states s in A, $(P{\cdot}A), s \models \varphi$ if $A, s \models \varphi$ during the advice verification algorithm.*

Proof: We assume without loss of generality that all CTL formulas are given in negation normal form (meaning that all negation operators are pushed inward so that only propositions are negated). The proof is by induction on the

structure of φ. In the base case, φ is a positive or negative atomic proposition. Since advising programs does not alter propositional values in either programs or aspects, the theorem holds in the base case.

For the inductive case, assume that the theorem holds for all formulas of size up to and including k. By the CTL semantics, the labels on states in A depend on the propositions in A and the labels assumed on state *out* in A, which are copied from the corresponding state in P. If the theorem holds for formulas of size $k + 1$ on states in P, then it must hold for formulas of size $k + 1$ in A by the semantics of model checking. We therefore only need to argue the inductive case for states in P (using the inductive assumption on states from both P and A).

Let φ be a formula of size $k + 1$ that labels state s in P. We must prove that φ labels s in $P{\cdot}A$; the proof depends on φ's outermost operator.

- If φ is of the form $\phi \wedge \psi$, then both ϕ and ψ must label s in P by the CTL semantics and must label s in $P{\cdot}A$ by the inductive assumption. φ must therefore label s in $P{\cdot}A$ by the CTL semantics.

- If φ is of the form $\phi \vee \psi$, then one of ϕ and ψ must label s in P by the CTL semantics and must label s in $P{\cdot}A$ by the inductive assumption. φ must therefore label s in $P{\cdot}A$ by the CTL semantics.

- If φ is of the form $\mathsf{AX}(\phi)$ or $\mathsf{EX}(\phi)$ the argument depends on whether s is a joinpoint state. If it is not, then all of its successors must also lie in P. By the inductive hypothesis, all those successors with ϕ as a label in P must have ϕ as a label in $P{\cdot}A$, so φ must also be true at s in $P{\cdot}A$. If s is a joinpoint state, then by construction its only successor in $P{\cdot}A$ is the start state of A; this means that there is no distinction between AX and EX at s. Since the advice verification algorithm reported that A preserves φ by assumption, the start state of A must satisfy ϕ. This label must carry over to $P{\cdot}A$ by the inductive assumption, so φ must label s in $P{\cdot}A$.

- If φ is of the form $\mathsf{A}[\phi \mathbin{\mathsf{U}} \psi]$ then the argument depends on whether s is a joinpoint state. We first prove that the theorem holds for the joinpoint states, then use that to prove that it holds for all other states.

  Assume s is a joinpoint state. If s satisfies ψ, the theorem holds due to the inductive hypothesis and the CTL semantics (which dictates that a state that satisfies ψ satisfies a formula of the form $\mathsf{A}[\cdots \mathbin{\mathsf{U}} \psi]$). By the CTL semantics, if s does not satisfy ψ, then it must satisfy ϕ and every path leaving s must eventually reach a state that satisfies (or "discharges") ψ.

  The argument depends on the location of the states that discharge ψ. Let WP (for "witnesses in P") be the minimal set of states closest to state s in P that satisfy ψ and that the CTL semantics uses to justify φ. Let $s_{a \to p}$ be the state in P to which the instance of A inserted at s returns to P. The proof must consider how applying A alters the paths to WP under each advice type.

  If the advice type is before or after, then all states in WP, and all states leading up to states in WP from s in P, lie in both P and $P{\cdot}A$ by construction. By the inductive hypothesis, P and $P{\cdot}A$ agree on whether ϕ and ψ label each such state, so φ must be a valid label on $s_{a \to p}$. The preservation check on A would only confirm φ on A if every path satisfied φ under the assumption of φ on $s_{a \to p}$ (even if this assumption was not necessary, due to A satisfying φ) so the result holds in this case.

  If the advice type is around and the advice does not invoke proceed, then states in WP may have been removed. If all paths from s in A satisfied φ without relying on a φ label on $s_{a \to p}$, then φ is clearly true at s in $P{\cdot}A$ and this case is complete. Assume that A contains a path from s (*in*) to $s_{a \to p}$ (*out*) that does not contain a

state satisfying ψ. Since the advice verification algorithm reports that φ is true, the model checker needed label φ on $s_{a \to p}$ to prove φ at *in*. The proof therefore reduces to showing that φ was an accurate label on *out* in $P \cdot A$. The aspect verification algorithm only copies AU labels from $s_{a \to p}$ to *out* that do not lie in U_{check}. From the definition of U_{check} (in Algorithm 4), if φ is not in U_{check} then the formula $\mathsf{A}[(!\mathsf{call}(f) \land \phi) \, \mathsf{U} \, \psi]$ must be true from $s_{a \to p}$. This formula requires that ψ was discharged before reaching a state labeled $\mathsf{call}(\mathsf{f})$: in other words, no path that witnesses φ required any states that could be elided by applying around advice at a call to f. This means that all states needed to establish φ from $s_{a \to p}$ are in both P and $P \cdot A$; by the inductive hypothesis, their ϕ and ψ labels from P are valid in $P \cdot A$. The φ label on $s_{a \to p}$ is therefore accurate in $P \cdot A$, so this case holds.

If the advice type is around and the advice does invoke proceed, no states are elided, but the label φ on $s_{a \to p}$ in P could still be invalid in $P \cdot A$ if (1) that label depended on φ labeling s and some state inserted by the advice violates ϕ, or if (2) A contains a cycle that does not satisfy ψ. The advice verification algorithm only copies AU labels from U_{check} to *out* that could be satisfied from s without the same label on $s_{a \to p}$; this restriction breaks any potential circular dependency between $s_{a \to p}$ and s with regard to φ labels. The inductive case therefore holds when s is a joinpoint state.

If s is not a joinpoint state, then either each path from s in P discharges ψ before reaching a joinpoint, or φ must have labeled the joinpoint. On paths that discharge ψ before reaching a joinpoint, all states involved in satisfying φ lie in P and have the same ϕ and ψ labels in $P \cdot A$ by the inductive hypothesis; all of these states must be in $P \cdot A$ because to be elided they would have to be on a path from s that includes a joinpoint. φ must therefore label s in $P \cdot A$. On paths that discharge ψ after reaching a joinpoint, φ must label the joinpoint by the CTL semantics. We have already established that φ must label the joinpoint in $P \cdot A$. The states between s and the joinpoint lie in both P and $P \cdot A$, and preserve their ϕ and ψ labels in $P \cdot A$ by the inductive assumption. The inductive case therefore holds at s.

- The argument for $\mathsf{E}[\phi \, \mathsf{U} \, \psi]$ is analogous to that for AU.

- If φ is of the form $\mathsf{AG}(\psi)$ or $\mathsf{EG}(\psi)$, the result holds because all states in P and $P \cdot A$ agree on the ψ labels by the inductive hypothesis, and all states in A are checked against φ by the aspect verification algorithm. The aspect check relies on the φ labels on the return state from the aspect for this check, but those labels cannot be inaccurate without some state in A failing to satisfy ψ (since advising adds no new states other than those in A). The inductive case therefore holds in this case.

$\square$

8 Related Work

There are many efforts to define formal semantics for aspects, such as the denotational model of Wand, Kiczales and Dutchyn [66]. Some of these have been accompanied by proposals on employing the semantics for verification. For instance, Andrews [4] uses process algebras to offer a foundation for AOP. That work is based on the earlier formulation of aspects [39] in terms of arbitrary "weavers". The work emphasizes proofs of the correctness of program weaving, using program equivalence to establish the correctness of a particular weaver.

The notion of compiling the PCDs to automata and matching these against the stack is due to Masuhara et al. [50]. This idea is refined by Sereni and de Moor [56]. They provide a language of PCD primitives (which we used as the basis for ours), and present a static analysis based on this. The analysis determines, for each call site, the shapes of the stacks possible at that site, and presents a pre-computation on a fixed set of PCDs that can reduce the work of the analysis. Their work does not, however, discuss verification (though it is a natural application) and, in particular, does not provide a methodology for, or discuss the subtleties of, modular verification in this context. Adaptive programming systems like Demeter [47] also rely on compiling regular specifications into automata to guide the traversal process. While we have not formally investigated the application of our techniques to Demeter, we believe such an application should indeed be possible.

Some researchers have considered aspect verification but in the context of analyzing the program after composition. Deng, et al. [19] use aspects to specify concurrency properties, then synthesize code with appropriate safety protocols and verify the result. Nelson, et al. [54] use both model checkers and model-builders to verify woven programs. Both Ubayashi and Tamai [63] and Denaro and Monga [18] employ model checkers to verify Java programs. These papers do not, however, describe a modular verification methodology or address the accompanying subtleties such as advice triggering fresh advice.

There is a growing body of work on techniques to study interference between aspects, such as those of Störzer and Krinke [60] and Kniesel [personal comm.]. These approaches are essentially orthogonal to our work in that they do not consume a user-specified property but rather analyze aspects for a fixed characteristic (like traditional type systems do). These efforts are therefore complementary to the work proposed here, but could potentially strengthen our work.

In a series of papers (e.g., [22]), Douence, et al. also study this problem through a formalism for AOP based on events. This has the benefit of lifting aspects to a more semantic level, which they use to define two notions of independence of an aspect, depending on whether or not it can be impacted by a particular program. (This is related to work on interface generation under parallel composition [27, 42].) The event-based definition shifts the work to a fundamentally parallel setting, however, which is difficult to compare with ours. While they provide proof rules for reasoning about programs, they do not specify the implementation status and whether the tools would run in as automated a fashion as a model checker.

Devereux [21] maps programs and aspects to concurrent systems. This leads to a fundamentally different style of reasoning, since our composition is sequential while his is parallel. His approach supports a rich family of aspect-like mechanisms, and may also be able to exploit results on generating environment models under parallel composition [27, 42]. However, it is unclear what price this model extracts in return for its power, especially given that languages like AspectJ use sequential composition. His formalization needs alternating-time logic, for which tool support does not appear to be as mature as for CTL or LTL.

Aldrich [1] presents a formal model of aspects, describes a type system for them, and proves an abstraction theorem enabling modular reasoning. While useful, this approach does not address verifying behavioral properties of programs. Its more major drawback is that it fails to tackle any of the invasive, dynamic features of aspects that make them both interesting and controversial. As such, therefore, the work is really about a slightly extended, but fairly traditional, module system, and it is unclear to what extent the aspects found in practice fall under his rubric.

Mousavi, et al. [53] discuss a new tool-suite for embedded systems. This suite is designed to exploit aspects in the design phase. While they discuss the desire to support verification at this level, it is not yet clear that they provide concrete support for it. Regimbal, et al. [55] discuss the use of aspects in hardware specification, concretely in a

system-on-a-chip packet filter using the e [sic] language, which includes an aspect-like advice mechanism. They also discuss the advantages of reusing verification in this setting. However, they do not appear to provide a formal framework for actually performing such reasoning. Tesanovic, et al. [62] perform timing analysis of real-time programs using the worst-case execution time framework. Their work, however, offers only a very simple model of pointcuts, and does not identify pointcuts in advice.

Xu, et al. [68] propose reducing aspect verification to prior work on reasoning about implicit invocation systems. In particular, they suggest using work that employs model- rather than proof-theoretic techniques. It is, however, unclear how their work addresses several issues that we do. They do not discuss around advice, which is arguably the most interesting kind, since it elides paths through a previously verified program, potentially rendering the result of prior verification invalid. At a more abstract level, it is unclear what the consequences of their reduction would be: whether verification works in a way that is meaningful to aspects, whether they can identify pointcuts induced by advice, what the formal properties about implicit invocation verification mean in the context of aspects, how to translate results of verification into a form meaningful to AOP developers, and so on.

Sihman and Katz employ "superimpositions", which are aspect-like notations parameterized to be more reusable. Their work helps users of Bandera model-checking [57] avoid the practical problem of annotating the program differently for each aspect's properties by employing superimpositions to weave in the annotations specific to each aspect. Their focus is on properties of aspects that programs might violate, and their interfaces target verifying the preservation of such properties. These interfaces, however, appear to be written entirely manually. Their methodology also covers preserving properties of the base program by aspects, but not through separate analysis of program and aspects as in our work. They do discuss the possibility of verifying the aspect independently in the context of a dummy program, and observe that this is an open-system verification problem, but do not offer a prescription for the generation of these dummy programs.

In another paper [58], Sihman and Katz present a sophisticated discussion of exactly what it means to verify advice and program. They also classify types of advice based on whether or not they alter control- and data-flow in the program. The work presented in this paper directly addresses what they dub "spectative" aspects. Related work [10, 46] can, however, extend our result to much broader sets of aspects. We therefore believe their work can help classify our result and point to useful directions for extending it.

More recently, Katz [35] gives a classification of aspects and shows implications for extending properties true of the base program to systems including aspects from various categories. For spectative aspects, the most restrictive category, the verification presented here is usually redundant once the aspect has been identified as falling in that category. For more general categories, however, that work does not demonstrate the implications for extending properties, especially for liveness; thus, the technique presented in this paper can prove many properties not treated there.

Modular verification is an old problem, often referred to as "assume-guarantee reasoning" [34, 52]. Most assume-guarantee techniques for model checking, following the lead of Clarke, et al [15], assume that modules compose in parallel, whereas aspects compose sequentially. Some research [3, 14, 45, 49] has considered modular verification with sequential control flow. Laster and Grumberg decompose programs into sequential fragments and verify the fragments incrementally from the end of the program to the initial state; each verification increment assumes properties already proven of later segments [45]. For verifying before and after advice, our approach resembles theirs. Our work goes much farther, however, to handle around advice (which can bypass states), cycles between multiple program fragments (which their decomposition rules out by construction), and the identification and maintenance of joinpoints.

Furthermore, a substantial philosophical difference exists between our work and theirs regarding the motivation for modular reasoning. They decompose a complete program into fragments to make model checking tractable, while we use modular reasoning to support a modular design methodology. Laster and Grumberg leave open the question of how to choose a decomposition; the works by Alur and Yannakakis [3] and Clarke and Heinle [14] use hierarchical state machines and StateCharts, respectively, to provide this decomposition. The design-driven motivation for our work forces us to handle partial (or open) programs, and thus to articulate interfaces between modules. These other works treat logical dependencies between fragments as an internal algorithmic detail. Masson, et al. [49] avoid the problem of articulating interfaces by decomposing systems around much stronger requirements, namely that all modules satisfy the property; we believe this requirement is not realistic in practice, as our example shows.

Several projects for software verification are loosely based on the same idea: apply predicate abstraction to the program source, verify, and use the counterexample to refine the abstraction [7, 11, 30, 32]. They differ in their details of how they generate and refine the abstractions. Other tools either translate to a model checker's input [17], conduct path exploration [28, 65], or generate verification conditions for theorem proving [20]. None of these approaches, however, tackle the complex modularities described in this proposal.

Alur, et al. [2] present a temporal logic that includes call and return statements for capturing properties of pushdown systems. While their logic would capture PCDs without the need for the bypass construction we use to identify pointcuts, their work does not address modular verification, and their use of pushdown systems makes it difficult to reuse existing verification tools.

Kicazles and Mezini [40] present a case study in modular aspect reasoning. Their example illustrates that reasoning about a single aspect must take into account the context of the aspect's application in the rest of the system, analogous to the interfaces generated in prior work by Blundell, et al. [10]. As such, we can view their work as parallel to that presented here, except done by informal reasoning through a specific example.

Fisler and Krishnamurthi [26] present a model for verifying product-line systems where each module encapsulates a feature. That work addresses the possibility of concurrency within each module. However, composition there occurs only at fixed points in the source, thus ignoring dynamic joinpoints. Li, et al. [46] and Blundell, et al. [10] extend these results to address open-system problems that also apply to aspects, respectively using three-valued model-checking and a two-phase technique based on constraint generation and solving. These works present approaches that would enable verification in a context where the advice was permitted to modify the data of the base program. They do not, however, address the important and interesting cases of dynamic composition and cascading aspects.

9 Conclusion and Future Work

There is a folklore belief that aspects inhibit any form of "modular" verification. This paper demonstrates that these claims are exaggerated. Aspects do admit modular, incremental verification in the presence of an appropriate form of lightweight specification that delimits their effect (just as type signatures and public/private designators enable separate type-checking). In our work, the PCDs play the role of these specifications. Indeed, our work makes an implicit case for separating the definition of PCDs from that of the actual advice.

Our technique generates interfaces at each state where advice can apply, storing enough information to enable separate verification of the advice. These interfaces cater specifically to the property under analysis, and may not be useful for unrelated properties. While the resulting technique is less general than one that utilizes interfaces constructed

manually by the developer, it has the benefit of being entirely automated. We call this process *property-driven interface generation*.

Relative to fixed PCDs, our technique is sound, and addresses subtleties introduced by around advice and the triggering of advice by other advice. The technique handles dynamic PCDs and, by virtue of supporting static PCDs, can also apply to other methods for modularizing crosscutting code [6, 8, 9, 25, 29, 47, 51, 59].

Our goal is to port these ideas to more scalable verification frameworks, especially exploiting known techniques [17, 23] for generating state machines from source. In addition, tackling Java source requires expanding the PCD language, and also handling more language features. We can already model some, such as static variables, by slightly altering the way we inline procedures; others, such as concurrency, will probably require techniques similar to ones we have developed in prior work [26].

Independent of the language, our technique must extend along several dimensions:

- We would like to exploit knowledge about the PCDs and properties to drive the model generation process.

- Once an advice completes, it restores the stack to the same state it had before invocation. Invoking advice can therefore have no impact on the pointcuts of either static or dynamic PCDs. In a system like AspectJ, which has a richer PCD language, this claim is no longer true. For instance, the use of `if` in a PCD makes it possible to write complex predicates that can, for instance, detect mutations performed by advice. A simpler example would use the `within` pointcut.

  In such cases, a tool would need to perform a value-flow analysis to determine when an advice can cause a joinpoint to enter a pointcut, conservatively over-estimate to preserve soundness, and use the body of the advice to determine whether or not to perform verification at a joinpoint. The model we present here remains applicable—only the set of states for which we generate interfaces changes—though a weak analysis would generate interfaces and suggest verification at unreasonably many states.

- Applying advice can remove states from pointcuts. For instance, section 4.4 considered the case of advice that does not invoke proceed; in that instance, pointcut elements in the fragment being advised will no longer execute. Or suppose the advice terminates program execution; then the rest of the program is no longer reachable. While it is sound to verify advice application at these states anyway, it can certainly lead to predictions of errors that do not occur on execution (since the program does not visit those states).

  This problem is not serious. Any advice can affect pointcuts if it terminates program execution in some or all paths, but this is easy to detect and address (indeed, this often indicates an error in the advice). In the absence of this, before and after advice are not problematic. The only remaining case is when no path through an around advice invokes proceed (which is easy to detect by reachability). In this instance, we need not verify joinpoints in the advised code. The set of such joinpoints can be recorded in the interface.

- When an around aspect does invoke proceed, the aspect itself often performs operations orthogonal to those being advised. For instance, the aspect might increment and decrement counters, or perform other such generic operations that have no effect on the program's properties. In such cases, the set of labels will not be affected by the advice state machine, which means the existing labels on the advised procedure can be reused safely. We believe that flow analyses can help identify cases when we can reuse the existing labels.

- The technique presented in this paper is designed to establish the *preservation* of program properties by aspects. In fact, aspects often introduce new invariants about programs. We can partially simulate this by verifying the *negation* of the property on the program; verifying that the advice then *violates* the (negated) property indicates that the property holds in at least some cases. (This violation of the property does not automatically guarantee that the original property holds, because our technique is not complete.) Because it is impossible to anticipate such properties, however, we need a better approach.

- Our technique assumes that state modified by aspects does not affect control flow in the remainder of the program. In general, this assumption is too restrictive. Our prior work presents two different techniques [10, 46] that use different analyses for addressing such systems, but does not account for all the subtleties of aspects (though we believe these are largely orthogonal). One key observation in that prior work is that it may be more useful to view the problem as one of constraint generation rather than as one of model checking, as the latter ultimately aims to give definitive answers. We intend to adapt those results to the aspect context.

Acknowledgments

References

[1] Aldrich, J. Open modules: Modular reasoning in aspect-oriented programming. In *Foundations of Aspect-Oriented Languages*, pages 7–18, March 2004.

[2] Alur, R., K. Etassami and P. Madhusudan. A temporal logic of nested calls and returns. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–481, 2004.

[3] Alur, R. and M. Yannakakis. Model checking of hierarchical state machines. In *Symposium on the Foundations of Software Engineering*, pages 175–188, 1998.

[4] Andrews, J. H. Process-algebraic foundations of aspect-oriented programming. In *Reflection*, pages 187–209, September 2001.

[5] Aspect oriented programming (article series). *Communications of the ACM*, 44(10), October 2001.

[6] Aßmann, U. *Invasive Software Composition*. Springer-Verlag, 2003.

[7] Ball, T. and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, January 2002.

[8] Batory, D. Feature-oriented programming and the AHEAD tool suite. In *International Conference on Software Engineering*, pages 702–703, 2004.

[9] Batory, D. and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[10] Blundell, C., K. Fisler, S. Krishnamurthi and P. Van Hentenryck. Parameterized interfaces for open system verification of product lines. In *IEEE International Symposium on Automated Software Engineering*, pages 258–267, September 2004.

[11] Chaki, S., E. Clarke, A. Groce, S. Jha and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[12] Clarke, E., E. Emerson and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

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

[14] Clarke, E. M. and W. Heinle. Modular translation of Statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie Mellon University School of Computer Science, August 2000.

[15] Clarke, E. M., D. E. Long and K. L. McMillan. Compositional model checking. In *IEEE Symposium on Logic in Computer Science*, pages 353–362, 1989.

[16] Clements, P. and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[17] Corbett, J. C., M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby and H. Zheng. Bandera : Extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448. IEEE Press, 2000.

[18] Denaro, G. and M. Monga. An experience on verification of aspect properties. In *International Workshop on Principles of Software Evolution*, pages 184–188, September 2001.

[19] Deng, X., M. B. Dwyer, J. Hatcliff and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.

[20] Detlefs, D. L., K. R. M. Leino, G. Nelson and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

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

[22] Douence, R., P. Fradet and M. Südholt. A framework for the detection and resolution of aspect interactions. In *International Conference on Generative Programming and Component Engineering*, pages 173–188, October 2002.

[23] Dwyer, M. B. and L. A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. Technical Report UM-CS-1999-052, University of Massachusetts, Computer Science Department, August 1999.

[24] Filman, R. and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns*, October 2000.

[25] Findler, R. B. and M. Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[26] Fisler, K. and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 152–163, September 2001.

[27] Giannakopoulou, D., C. Pasareanu and H. Barringer. Assumption generation for software component verification. In *IEEE International Conference on Automated Software Engineering*, pages 3–12, 2002.

[28] Godefroid, P. Model checking for programming languages using VeriSoft. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[29] Harrison, W. and H. Ossher. Subject-oriented programming: a critique of pure objects. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 411–428, 1993.

[30] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre. Software verification with Blast. In *SPIN Workshop on Software Model Checking*, number 2648 in Springer Lecture Notes in Computer Science, pages 235–239. Springer-Verlag, 2003.

[31] Hilsdale, E. and J. Hugunin. Advice weaving in AspectJ. In *International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

[32] Holzmann, G. and M. H. Smith. Software model checking - extracting verification models from source code. *Software Testing, Verification, and Reliability*, 11(2):65–79, June 2001.

[33] Huth, M. and M. Ryan. *Logic in Computer Science*. Cambridge University Press, second edition, 2004.

[34] Jones, C. B. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[35] Katz, S. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development*, 1:106–134, 2006. Published as *Lecture Notes in Computer Science* number 3380.

[36] Kiczales, G. The more the merrier. *Software Development*, October 2004. http://www.sdmagazine.com/documents/s=8993/sdm0410g/.

[37] Kiczales, G., J. des Rivières and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[38] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[39] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[40] Kiczales, G. and M. Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering*, pages 49–58, 2005.

[41] Krishnamurthi, S., K. Fisler and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 137–146, November 2004.

[42] Kupferman, O., M. Vardi and P. Wolper. Module checking. In *International Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 75–86. Springer-Verlag, 1998.

[43] Laddad, R. *AspectJ in Action*. Manning Publications Co., 2003.

[44] Laroussinie, F., N. Markey and P. Schnoebelen. Temporal logic with forgettable past. In *IEEE Symposium on Logic in Computer Science*, pages 383–392, 2002.

[45] Laster, K. and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 20–35, 1998.

[46] Li, H. C., S. Krishnamurthi and K. Fisler. Modular verification of open features through three-valued model checking. *Automated Software Engineering Journal*, 12(3):349–382, July 2005.

[47] Lieberherr, K. J. *Adaptive Object-Oriented Programming*. PWS Publishing, Boston, MA, USA, 1996.

[48] Maidl, M. The common fragment of CTL and LTL. In *Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[49] Masson, P.-A., H. Mountassir and J. Julliand. Modular verification for a class of PLTL properties. In *Integrated Formal Methods*, pages 398–419, November 2000.

[50] Masuhara, H., G. Kiczales and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction*, pages 46–60, 2003.

[51] Mezini, M. and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 97–116, October 1998.

[52] Misra, J. and M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[53] Mousavi, M., G. Russello, M. Chaudron, M. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta and D. C. Schmidt. Using Aspect-GAMMA in design and verification of embedded systems. In *International Workshop on High Level Design Validation and Test*, pages 69–75, October 2002.

[54] Nelson, T., D. D. Cowan and P. S. C. Alencar. Supporting formal verification of crosscutting concerns. In *Reflection*, pages 153–169, 2001.

[55] Regimbal, S., J.-F. Lemire, Y. Savaria, G. Bois, E. M. Aboulhamid and A. Baron. Aspect partitioning for hardware verification reuse. In *Workshop on System-on-Chip for Real-Time Applications*, pages 183–192, 2002.

[56] Sereni, D. and O. de Moor. Static analysis of aspects. In *International Conference on Aspect-Oriented Software Development*, pages 30–39, March 2003.

[57] Sihman, M. and S. Katz. Model checking applications of aspects and superimpositions. In *Foundations of Aspect-Oriented Languages*, pages 51–60, March 2003.

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

[59] Smaragdakis, Y. and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.

[60] Störzer, M. and J. Krinke. Interference analysis for AspectJ. In *Foundations of Aspect-Oriented Languages*, pages 35–44, 2003.

[61] Sullivan, K., W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 166–175, September 2005.

[62] Tesanovic, A., J. Hansson, D. Nyström, C. Norström and P. Uhlin. Aspect-level WCET analyzer. In *International Workshop on Worst-Case Execution Time Analysis*, July 2003.

[63] Ubayashi, N. and T. Tamai. Aspect oriented programming with model checking. In *International Conference on Aspect-Oriented Software Development*, April 2002.

[64] Vardi, M. Y. and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1), November 1994.

[65] Visser, W., K. Havelund, G. Brat and S. Park. Model checking programs. In *IEEE International Symposium on Automated Software Engineering*, pages 3–12, September 2000.

[66] Wand, M., G. Kiczales and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, 2004.

[67] Wolper, P. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

[68] Xu, J., H. Rajan and K. Sullivan. Aspect reasoning by reduction to implicit invocation. In *Foundations of Aspect-Oriented Languages*, pages 31–36, March 2004.