Verifying Aspect Advice Modularly*

Shriram Krishnamurthi
Computer Science Dept.
Brown University
sk@cs.brown.edu

Kathi Fisler
Dept. of Computer Science
WPI
kfisler@cs.wpi.edu

Michael Greenberg
Computer Science Dept.
Brown University
mgreenbe@cs.brown.edu

ABSTRACT

Aspect-oriented programming has become an increasingly important means of expressing cross-cutting program abstractions. Despite this, aspects lack support for computer-aided verification. We present a technique for verifying aspect-oriented programs (expressed as state machines). Our technique assumes that the set of pointcut designators is known statically, but that the actual advice can vary. This calls for a modular technique that does not require repeated analysis of the entire system every time a developer changes advice. We present such an analysis, addressing several subtleties that arise. We also present an important optimization for handling multiple pointcut designators. We have implemented a prototype verifier and applied it to some simple but interesting cases.

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: modular verification, model checking, aspect-oriented software

1. INTRODUCTION

There is growing consensus that traditional software structures have notable abstraction weaknesses, and new techniques are evolving to address them. In general, these techniques perform invasive modifications of programs [4], because their raison d'être is to capture abstractions that cannot easily be expressed through the gentler forms of composition that traditional modules provide. Some techniques are purely static, manipulating the program's source, while others have dynamic elements, offering the ability to reflect on the state of the program's execution and to conditionally modify it.

*This work is partially supported by the U.S. National Science Foundation grants CCR-0305834 and CCR-0305950.

SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.

Aspect-oriented programming (AOP) [17], especially as realized in the AspectJ language [16], is one of the most popular forms of invasive program composition. Aspect specifications have both static and dynamic elements, but the most distinctive ones are arguably in the latter category. In particular, AspectJ provides a pattern language that predicates the execution of an aspect on the shape of the runtime stack. It can therefore express a rich family of coherent, conceptual ideas that would be difficult to encapsulate in most traditional notions of a module.

With AOP's popularity burgeoning, software engineers will 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 heightens the potential for insidious errors.

In this paper, we present a technique for verifying programs with aspects. In particular, we support an interesting use-case of AOP, whereby programmers are able to write high-level specifications of where aspects should apply (the pointcut designators) but may need several development iterations to write the advice correctly. A naïve approach to verification would simply compose all the advice against the program and check the composition on every iteration. This is, however, time consuming, unwieldy, and potentially difficult (in a scenario where the developers don't wish to share their source with authors of advice).

Our technique adapts model-checking [5] to enable developers to verify properties against advice without having access to the program source. Given

1. a set of properties that the program must satisfy, even in the presence of aspects, and
2. a set of pointcut designators (PCDs)

our technique automatically generates sufficient conditions on the program's pointcuts to enable verification of advice in isolation. We do *not* require simultaneous specification of the advice that will correspond to each pointcut designator, or even of its type (before, after, around); this is a departure from aspect languages like AspectJ. Thus, given the PCDs, this enables a *modular* verification strategy, whereby neither the base program nor advice need access to each other for verification. Our approach even handles, in a modular fashion, subtleties such as the extension of pointcuts by the application of advice.

Our approach currently assumes that the programs and advice are given as state machines. There are tools such as FLAVERS [11] and Bandera [6] that currently consume Java source and generate state machines similar to those we need;

we expect similar tools to eventually exist for aspect source, and thus regard this problem as orthogonal to the work presented here. For the experiments we have performed with our prototype checker, we have constructed the automata manually from AspectJ source programs.

The heart of our result is in the following sections: section 3 presents the verification scenario we tackle. Section 5 presents our modular verification technique through a running example. Section 6 discusses several subtleties that our work engenders and addresses. Section 7.2 then explains how we can eliminate a significant performance bottleneck in the algorithm of section 5. Section 8 outlines the formal results that support our work.

2. BACKGROUND

2.1 Aspect-Oriented Programming

Each AOP system defines a family of program locations, called *joinpoints*, at which a programmer can *advise* program execution. *Advice* is a fragment of code that is typically executed either before, after or "around" the evaluation of the joinpoint. An around advice is executed in place of the original joinpoint, though the author of the advice has the ability to *proceed*, i.e., to execute the advised code. These AOP mechanisms thus simulate some of the power found in advanced object-oriented systems [15].

The AspectJ implementation of AOP provides a powerful language for describing when 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 "static" assumption in this paper is that the PCDs are known statically; our technique handles both static and dynamic PCDs.) The subset of the PCD language of AspectJ that we consider in this paper essentially expresses 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., when a stack frame for q is lower on the stack, and 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 that the system must obey. The state machine can be non-deterministic. The property is typically written in temporal logic. In this paper, we use model-checkers that employ the temporal logic CTL.

The atoms of CTL are propositions that label states. CTL permits combination of these atoms using standard propositional operators and connectives (negation, conjunction, implication, etc). Finally, CTL can capture *temporal* properties. A formula 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, we must quantify this formula to express 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, while $\mathsf{E}[\phi \ \mathsf{U} \ \psi]$ requires that there Exists a path where this holds. In this paper we also use AG, whose sub-formula must hold in all states; AF, which holds of a state exactly when its sub-formula eventually holds along every path from that state; and EX, whose sub-formula must hold in at least one next state.

The semantics of CTL model-checking is given in terms of state labelings. The model-checker labels the state machine with the sub-formulas of the property, working bottom-up. 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.

3. VERIFICATION PROBLEM SETUP

Our verification system consumes a program along with a set of pointcut designators, sans advice, and a family of properties that express desired temporal behaviors in CTL. The developer expects the system to exhibit these behaviors and the application of advice to not violate them.

Verifying whether the main program exhibits the properties is relatively straightforward using model-checking. In contrast, establishing that the advice does not violate the properties is challenging. The model-checker needs to traverse the paths of both the program and the advice to be able to establish the property. Demanding, however, that the developer combine the advice and program prior to each verification is not always feasible:

1. The advice may be authored at a different time or in a different place from 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 can extend the module simile to demand "separate verification" analogous to separate compilation. In the latter setting, the use of descriptive interfaces enables programmers to work independently. Writing such interfaces for verification is, however, onerous. Instead, given a program, a set of PCDs, and a property, our verifier *automatically* generates interfaces relative to these inputs. These interfaces can be published for use by advice authors, who can employ them to validate advice over a much smaller input (the advice alone, rather than the composed program). When the PCDs change, of course, the corresponding interfaces have to be generated afresh, forcing advice re-verification. On the other hand, changes to the base program that do not affect the interfaces are guaranteed to not affect the results of advice verification. The rest of this paper presents the details of this process.

4. FORMAL MODELS

We establish some terminology for the rest of the paper.

4.1 Programs

A *state machine* M is a tuple $\langle S, T, L, S_{\text{src}}, S_{\text{sink}}, S_{\text{call}}, S_{\text{rtn}} \rangle$. The machine intuitively represents the control-flow graph of a program fragment.

- S is a set of states. Intuitively, these are statements and expressions in the program.

- $T \subseteq S \times S$. This reflects the flow of control between the states of M.
- $L : S \to 2^{\mathrm{AP}}$ for some set of atomic propositions AP. This labels what is known to hold at each state.
- $S_{\mathrm{src}} \in S$ 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. The states in S_{call} carry the label $\mathsf{call}(p)$ (where p is the function being called), and those in S_{rtn} are correspondingly labeled $\mathsf{return}(p)$. Intuitively, every state in S_{call} denotes an invocation of a function, and the corresponding S_{rtn} state is where control returns when the functions completes execution. We represent this in the graph with an edge between the two related states, with no other outgoing edges from that call state, and no other incoming edges to that return state.

A *function* is a state machine with a name taken from some set of symbols. A *program source* is a set of functions with distinct names, including a function named main. This is a sufficiently general model to handle most programming languages; most modern programming abstractions can be mapped to it using standard program analyses.

A *program* is constructed from a program source relative to some inline depth parameter. To generate a program from the program source, traverse 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, remove the edge between c and r; add an edge from c to the source of F; and add an edge from the sink of F to r. Inlining then proceeds recursively in F until it exceeds the depth parameter. This definition of program construction corresponds to the intuition that program execution begins at main and proceeds through each function call. While a better treatment of inlining is a candidate for future work, note that our definitions do *not* require main (and hence the program) to terminate.

4.2 Aspects

Advice is a state machine. Advice machines can, in addition, have distinguished proceed and resume states, analogous to function call and return states.

The *joinpoints* used in this paper are function calls. A *pointcut atom* is one of the following:

- $\mathsf{call}(f)$ for some function name f
- $!\mathsf{call}(f)$ for some function name f
- true

A *pointcut element* is one of the following

- a pointcut atom
- a^* where a is 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:

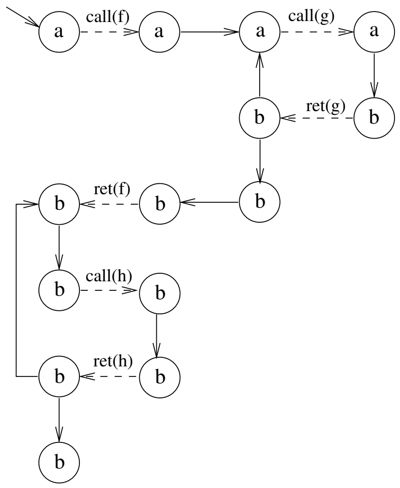


Figure 1: Sample Program

- a pointcut element
- (d) where d is a pointcut designator
- $d_1; d_2$ where d_1 and d_2 are pointcut designators
- $d_1|d_2$ where d_1 and d_2 are pointcut designators

In other words, PCDs are a restricted form of regular expressions. A PCD subscribes a set of states of the program at which it applies. This set of states is called a *pointcut*. We describe the process of identifying the states that match a PCD in section 7.

An *aspect* consists of a PCD, an advice type, and the advice. The advice types are before, after, and around.

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 (recall that we advise only function invocations):

before For each state in the pointcut, replace the edge from the call state 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.

after After is treated analogously.

around Replace the edge from the call state to the source state of the function with an edge to the source state of the advice. Replace the edge from the sink state of the function to the return state 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.

5. MODULAR VERIFICATION

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

The sample program is given in figure 1. The solid lines show transitions within a function, and the dashed lines show function invocation and return. The main program

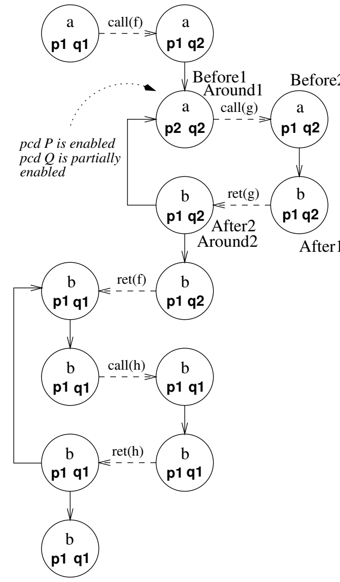


Figure 2: Cross Product

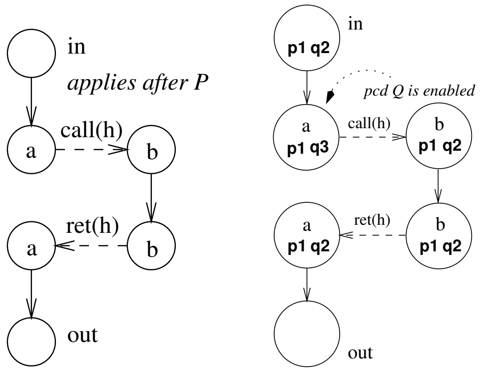


Figure 3: Advice A and Labeled

invokes function f and then h, while f invokes g. We will use the CTL property

$$\mathsf{AG}(\mathsf{A}[a \;\mathsf{U}\; b])$$

which says that in all states of the system, a is true on all paths until reaching a state where b is true. A quick inspection reveals that the program satisfies this property.

Suppose we are verifying this program in the presence of two PCDs, which we shall refer to as P and Q respectively:[1]

P: true^*; $\mathsf{call}(g)$

Q: true^*; $\mathsf{call}(f)$; true^*; $\mathsf{call}(h)$

Observe that the program enables PCD P. Furthermore, the program *partially* enables PCD Q by calling f, leaving open the possibility that advice might invoke h and thereby trigger the PCD.

Because the PCDs are regular, we can imagine using standard algorithms for converting them into automata (as we discuss in section 7) and labeling the program with the states of the PCD automata. Figure 2 shows what such a labeling might look like. The labels p1 and p2 represent states in the automaton for P. The label p2 represents having witnessed an invocation of g while p1 indicates being prepared to do so. Therefore, p2 labels only the state where g is invoked. Similarly, q1 and q2 capture the state of matching Q. The state q1 is the initial state, while q2 represents a call to f currently being on the stack.

This labeled automaton identifies states that satisfy a PCD. These are states where a programmer might eventually apply advice. As we have observed, only PCD P is enabled, at the state indicated by the label and arrow.

This labeled state-machine and a property are fed to a model-checker. The model-checker labels each state with all the sub-formulas of the property that are true of that state.

[1]In AspectJ, these would be written as `call(g)` and `call(h) && cflow(call(f))`, respectively.

Because the indicated state matches a PCD, we 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 contains the labels that the model checker ascribes to the states that lead to and return from the advice, but does not include information about the rest of the program's states.[2]

The heart of the modular verification process is as follows. Suppose the advice A (shown on the left in figure 3), which invokes h, is applied as after-advice at the pointcut of PCD P. Note that A in isolation does not satisfy the property (because the state before the sink satisfies a but not b), but that the program composed with A continues to do so. Our model-checker seeds A's sink state *out* with the labels for $After_2$ from the interface (b, $\mathsf{A}[a \;\mathsf{U}\; b]$, and $\mathsf{AG}(\mathsf{A}[a \;\mathsf{U}\; b])$) and seeds A's source state *in* with the propositions from $After_1$ from the interface (b). It then model-checks each label stored for $After_1$ on A (checking source labels against copied sink labels matches the backward propagation inherent in the CTL model-checking algorithm [5]). If all of these checks pass, the composed program will satisfy the property. If a check fails, the advice may violate that property (if the property depended on the violated label); the checker 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 $Before_2$ in figure 2.

Observe that the algorithm verifies the advice state machine without traversing the body of the program. Ideally, we would like to show that this process is nonetheless sufficient: if this check succeeds, so would verifying the program with the advice explicitly spliced in. Unfortunately, this is not (yet) true!

To see the problem, recall the PCD Q. The main program invoked f; the applied advice A invokes h. Therefore, the program and advice *combine* to trigger the PCD. Indeed, an actual AOP implementation would detect this condition.

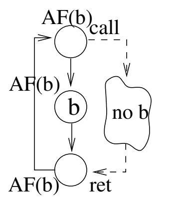
While it is clear we must label the advice A with the states of the PCD automata (for the same reason we did with the main program), it is easy to do this incorrectly. If we effec-

[2]While there is one interface for each state in the pointcut, we expect the number of states in a pointcut *relative to a* PCD to often be small. Furthermore, in most cases these interfaces will have logically related formulas (because, in general, the advice will tend to apply in similar circumstances).

tively compute the cross-product with the PCD automata in their initial states, we would still fail to notice the enabling of PCD Q. Instead, we need to initialize the PCD automata *in the states they were in at the point of applying the advice*, which is information we must record in the interface; indeed, the automaton for PCD Q is not in its initial state, since it has witnessed an invocation of f.

The resulting labeling is shown on the right in figure 3. This labeling correctly identifies that PCD Q is satisfied by a combination of the main program and advice A in the indicated state (q3 labels the state in the automaton for Q where it is satisfied), resulting in a second generated interface. If advice associated with Q violates a property, our verifier is able to report the violation in terms of both aspects, resulting in a helpful diagnostic.

Verifying around-advice modularly is more subtle. This verification uses the $Around_i$ labels. Around (without proceed) can bypass existing states, rendering them unreachable after the advice is applied. In contrast, before- and after-advice merely expand an existing transition with additional states. In the figure at right, solid edges are transitions of the base program while the dashed edges reflect an around-advice. Only the indicated state has the label b. This is sufficient for the base program to satisfy the property $\mathsf{AF}(b)$. The advice neither invokes proceed nor has a state labeled b, with the result that this property no longer holds of the advised system. Unfortunately, the verification process described above will fail to notice this property violation. The heart of the problem is that around advice can invalidate a label copied to the advice's sink state.



Fortunately, this problem arises only with formulas that capture eventual behavior where there is a path from the state at the exit of the advice to the state that enters the advice. The problem illustrated in the figure arises because the eventuality is satisfied *on a path that the around advice eliminates.* Whenever model-checking labels a call state with an eventuality property that labels the corresponding return state, we must determine whether the property is discharged before reaching the return state. This test is easily generated from the property and expressed in CTL; in this example, it is $\mathsf{A}[\mathsf{!return}\ \mathsf{U}\ b]$. (Had the property been an existential, the test would correspondingly have used an existential path quantifier.) If this formula succeeds at the call state of the base program, the formula is included in the interface for the call state, and is also checked against the advice during modular verification. This check would have failed in the example shown above. A formal description of a similar problem, the solution and its proof of correctness appear in our technical report [13].

Verification Process Summary

Given a fixed set of PCDs, we can verify advice in isolation from the base program. To do so, we must generate an interface that "caches" the state of the verification at pointcut states. The challenge is to cache correctly. The interface must store labels on the states $Before_1$, $Before_2$, $After_1$ and $After_2$ ($Around_1$ is the same as $Before_1$ and $Around_2$ is $After_2$), and the state of the PCD automata. (We can shrink the interface slightly if we know the advice type in advance.) At each pointcut state, the verifier seeds the advice program fragment with interface labels and verifies their preservation. It also employs the state of the other PCDs at this joinpoint to determine whether additional advice must apply.

6. SUBTLETIES

The above approach has the potential to fall prey to numerous subtleties. We discuss each of these in turn.

▷ *Can applying advice cause joinpoints in the advice to become members of a pointcut?*

Yes, and our technique is sensitive to this possibility. Indeed, the running example of section 5 addresses this scenario. The interfaces described in this paper store information to accurately identify such additions to pointcuts.

▷ *What happens when an around advice uses* proceed*?*

Suppose we are advising 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 and 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 $\mathsf{AF}(\phi)$ may no longer hold.) For this reason, we currently inline a copy of f and verify f's body in the context of the proceed-resume states.

In practice, however, we believe this will often be unnecessary. When an around advice invokes proceed, the aspect itself often performs operations orthogonal to those being advised. For instance, the aspect might increment and decrement counters, a generic operation that has no effect on the program's properties. In such cases, the set of labels will not be affected by the advice, which means the labels on f can be reused safely. We believe a value-flow analysis can help identify cases when we can reuse the existing labels.

▷ *Can applying advice cause new joinpoints in the program to enter a pointcut once the advice has completed?*

This cannot happen in our model. Once 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. As just one example, the use of `if` in a PCD makes it possible to write complex predicates that can, for instance, detect mutations performed by advice. In such cases, our tool would need to perform a value-flow analysis to determine when an advice can cause a joinpoint to enter a pointcut, to conservatively over-estimate to preserve soundness, and to 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.

▷ *What happens when multiple* PCD*s apply at a state?*

Implementations like AspectJ employ a simple strategy to order the applicable advice. In our model, if two PCDs match at a state, the application of the advice corresponding to one PCD cannot affect whether or not the second PCD still applies. This follows by the same reasoning about the stack discipline used to establish that advice cannot cause new joinpoints to enter a pointcut once the advice has completed. We therefore simulate AspectJ's strategy of ordering applicable advice.

▷ *Can applying advice remove states from pointcuts?*

Yes! For instance, suppose an around advice does not invoke proceed; pointcut elements in the fragment being advised will no longer execute. Or, if 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.

▷ *What about properties introduced by aspects?*

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. While our approach is not designed to tackle this, we can partially simulate it: see the example in section 9.

7. IDENTIFYING POINTCUT STATES FROM PCDS

The technique presented in section 5 depends on being able to identify the pointcut states corresponding to each PCD, independently of how the program was constructed. We will begin by describing a straightforward technique for doing so (which we believe is the one most readers would expect), explain its shortcomings, and provide a superior solution that does not suffer from these difficulties.

7.1 Using Cross-Products

For simplicity, assume there is only one PCD. Suppose we can convert the PCD into an automaton (henceforth referred to as a PCD *automaton*). Taking the cross-product of the program with this automaton identifies the states of the program that belong in the pointcut. As the example in section 5 showed, the cross-product is a useful conceptual model: it both identifies states where advice applies and helps generate accurate interfaces that correctly predict the addition of advice joinpoints to pointcuts.

In principle, we can use standard algorithms to compile the PCD, which is regular, into an automaton. There are, however, several reasons why this is not straightforward. Some of these are relatively easy to resolve. In a naïve compilation of PCDs, the resulting automata will expect nested function calls to happen immediately on entry into the enclosing function; the generated automata must instead stutter. Also, standard algorithms for generating automata from regular expressions produce Mealy machines, whereas verification relies on Moore machines; however, converting between the two is routine.

There are, unfortunately, two significant shortcomings with actually computing the cross-product. First, PCDs mention only calls, not returns. This is problematic because PCDs, which are regular, monitor stack contents, which are context-free, and could hence incorrectly recognize a sequence of non-nested calls as a sequence of nested calls.

Second, consider a small number of PCDs, each with simple PCD automata: say 5 PCDs with automata of 5 states each. Each automaton contributes a multiplicative factor to the number of states in the cross-product, with the result that the resulting state machine will have 5^5 or about 3000 times as many states as the original program. Model-checking is sensitive only to the number of *reachable* states, but the more distinct the PCD automata are (representing different conditions that trigger the application of distinct features), the more the states that will actually be reachable.

We can address the first problem by adding additional transitions to a PCD automaton to track function returns. Indeed, because we have bounded the depth of inlining (and thus of recursion), we can modify the automaton generator to precisely count the number of function applications and returns. Doing this, however, magnifies the size of the PCD automata even more, thereby further exacerbating the effect of the cross-product on automaton size. Therefore, we would benefit from a strategy that avoids an explicit cross-product construction while preserving the two benefits that the cross-product confers.

7.2 Avoiding Cross-Products

In the absence of an explicit cross-product, pointcuts can be identified by traversing the program and advice graphs. We exploit the graph traversal power of model-checkers to perform this identification.

Consider the PCD labeled Q in section 5, and suppose the program contains calls of h nested within those of f. We can easily express this PCD as a temporal logic formula, but a direct application of model-checking would not properly identify pointcut states. This is because the model-checker would label the formula 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 examine 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 require either a separate algorithm or would incur an exponential blow-up on translation into regular CTL.)

We can examine the past efficiently 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

$$\mathsf{call}(h) \wedge \mathsf{E}[\mathsf{true}\ \mathsf{U}\ \mathsf{call}(f)] \qquad (1)$$

This formula should label exactly those states in the reversed machine with a $\mathsf{call}(h)$ 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 only signifies that the state does not belong in a pointcut. (Observe that we are exploiting the model-checking's traversal power to do something quite distinct from verification.)

This proposal is, however, not sound. Consider a program that invokes f and h in sequence, with f in turn invoking h also. Figure 4 (left) shows such an example. This program should match the PCD only once (the call to h within the dynamic extent of f).

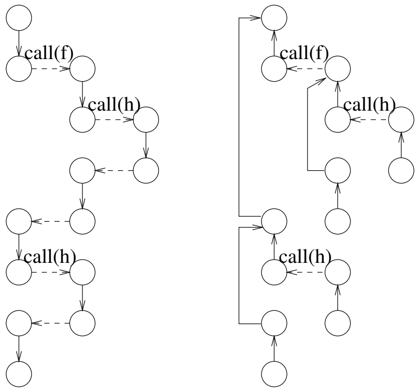


Figure 4: A Reverse-Bypass Construction

Formula 1 matches this machine with its edges reversed at two places. The error here is similar to that with naïve PCD automata, namely 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 can patch the formula to include this as an additional temporal condition, it is cumbersome to properly accommodate nested calls to the same function; in addition, adding these terms magnifies the size of the formula, which is a multiplicative factor in the running time of the model-checker. The situation is therefore analogous to the one we faced with PCD automata.

The 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 statement 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 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 would point to a return state should instead point to the successors (in the reversed graph) of the corresponding call. For instance, the graph on the right of figure 4 shows a version of the graph on the left with the edges reversed and calls bypassed. Formulas checked against this state machine need not match calls with returns, because completed function invocations have been elided from the traversed paths.

We translate PCDs into CTL formulas over these machines as follows. Since the PCD grammar disallows | operators within *, we can distribute ; over |, resulting in a PCD with all | operators at the outermost level. Since | corresponds to logical or, the heart of the algorithm is therefore the conversion of PCDs with the ; operator into CTL formulas. Given a PCD *pcd*, we define its CTL *identifier* as follows. Reverse *pcd* and append the symbol ∘; call this *pcd*′. The CTL identifier is PCD2CTL(*pcd*′) where PCD2CTL is defined as:

PCD2CTL(*pcd*) =
case *pcd* **of**

$$
\begin{array}{ll}
\text{a} & = \text{a} \\
(e) & = (\textsc{pcd2ctl}(\text{e})) \\
e_1 \wedge e_2 & = \textsc{pcd2ctl}(e_1) \wedge \textsc{pcd2ctl}(e_2) \\
e_1 \vee e_2 & = \textsc{pcd2ctl}(e_1) \vee \textsc{pcd2ctl}(e_2) \\
\circ & = \mathsf{start} \\
a_1; a_2^*; p & = (\mathsf{call} \wedge a_1) \wedge \mathsf{EX}(\mathsf{E}[(\mathsf{call} \rightarrow a_2)\ \mathsf{U}\ \textsc{pcd2ctl}(p)]) \\
a^*; p & = (\mathsf{call} \vee \mathsf{start}) \wedge \mathsf{E}[(\mathsf{call} \rightarrow a)\ \mathsf{U}\ \textsc{pcd2ctl}(p)] \\
a; p & = (\mathsf{call} \wedge a) \wedge \mathsf{EX}(\mathsf{E}[!\mathsf{call}\ \mathsf{U}\ \textsc{pcd2ctl}(p)])
\end{array}
$$

where a, a_1, and a_2 are pointcut atoms and p is a PCD. This algorithm assumes every call state has the label call and that the source state of the main function has the label start.

Identifying pointcut states in this manner requires model-checking the CTL identifier against the reverse-bypass state machine. The labels generated on states by this process now become part of the interface associated with each pointcut state. They take the place of the PCD automata states we alluded to in section 5, fulfilling essentially the same role.

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 size of the state machine and the formula. As a result, in linear time we can determine the pointcut states and avoid the explosion in the size of the input to the verification. This is crucial because space is often the dominant factor in verification. We note in passing that our optimization can be implemented easily with symbolic representations [5] also, because reversing the edges corresponds to swapping the current- and next-state variables in the BDD for the transition relation, identifying the edges to add and delete (for bypassing) is accomplished by projection, and actual addition and deletion are just BDD-or and -and, respectively.

Finding Pointcut States in Advice

As we saw in section 5, advice can induce new states in pointcuts. There, we suggested that cross-producting the advice with automata for the PCD, initialized to their state taken from the interface, can identify these new pointcut states. Now that we have shown the use of a model-checker to identify pointcut states in the program, we must demonstrate that this idea applies to advice as well.

Identifying pointcut states in advice composes two techniques we have already seen: model-checking a program fragment, and constructing the reverse-bypass version of a fragment. There are two salient details:

1. Because we will model-check the reversed advice machine, we must copy labels to the source end of the advice (rather than to its sink). Note that the only labels that matter for this process are those generated by PCD2CTL; these are unrelated to the labels from verifying the system's desired behavior.

2. Rather than confirm labels, we model-check the CTL identifiers on the reverse advice. Whereas with verification we are trying to establish the preservation of properties, here we only want to label states to determine whether or not a state belongs in a pointcut (just as with the base program).

8. THEOREMS

The correctness of this work relies on the modular algorithms producing the same results as analyzing the composed program. Our modular technique identifies a state as belonging to a pointcut iff that state would lie in the pointcut in the composed program. The modular algorithm labels a state with a CTL formula if model checking the composed program would ascribe that label to that state. The

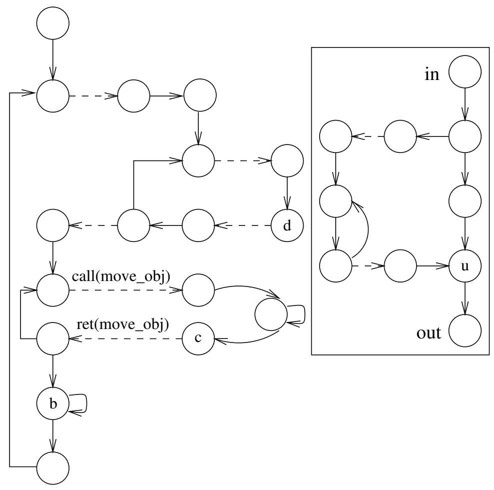


Figure 5: Display Update Example

following formally state these correctness criteria. We omit the details owing to lack of space.

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. Let G_{RB} denote the reverse-bypass of graph G.

Theorem 1 (optimization). *Let s_p be a state in P, pcd be a* PCD*, and φ_{pcd} be the* CTL *identifier of pcd. $P_{\mathrm{RB}}, s_p \models \varphi_{\mathrm{pcd}}$ iff s_p is a call state for which the sequence of calls on the stack at s_p (viewed as strings) is in the language defined by (the regular expression) pcd.*

Theorem 2 (modular identification). *Let s_a be a state in A. Let φ be the* CTL *identifier of a* PCD*. $A_{\mathrm{RB}}, s_a \models \varphi$ by the modular pointcut identification algorithm (section 7.2) iff $(P{\cdot}A)_{\mathrm{RB}}, s_a \models \varphi$.*

Theorem 3 (modular verification). *Let φ be a* CTL *formula. For all states s in P, $(P{\cdot}A), s \models \varphi$ if $P, s \models \varphi$ and the advice verification algorithm reports all labels are preserved. For all states s in A, $(P{\cdot}A), s \models \varphi$ if $A, s \models \varphi$ by the advice verification algorithm.*

9. ADDITIONAL EXAMPLE

We have successfully applied our model-checker to some examples that are inspired by samples in the Eclipse AspectJ tutorial. Our goal in this study has only been to establish the potential of our approach. Because we are working with a prototype model-checker (we are not aware of checkers that let users seed states with labels, which we need), the actual performance numbers are not meaningful. Due to space limitations, we present only one example here.

Figure 5 shows the controller fragment of a GUI implemented using the model-view-controller paradigm. The figure shows the controller's program (the primary state machine) and an advice (shown in the box). The controller creates, moves and deletes objects based on user interactions. In our example, we show states where the controller deletes objects (labeled d) and moves them (by invoking move_obj). When an object has been moved, its status is marked as changed in the state labeled c until the controller has issued a broadcast (at b).

With this program, we would like to establish the following property: when an object changes, it should not get deleted until the change has been appropriately broadcast. We can encode this as

$$\mathsf{AG}(c \rightarrow \mathsf{A}[!d\ \mathsf{U}\ b]) \tag{2}$$

The program clearly exhibits this property, as the model-checker establishes. Given the PCD true*; call(move_object) (in AspectJ notation, `call(move_object)`), the checker also generates an interface at the call to move_object. This interface now enables us to verify multiple pieces of concrete advice. For example, the advice in figure 5 performs a visual update of the moved object (at state u). Since the update must happen after a move, this is inserted as an after advice. From the perspective of the program, this advice creates new control-flow paths between the change and its broadcast. Fortunately, our modular technique establishes that the property continues to hold without traversing the program again. (It also introduces properties about u at the source of the advice. Modularity keeps these from being propagated to the program states, highlighting the incompleteness of state label ascription.) When, instead, we apply advice that (prematurely) deletes the object, our verifier reports the offending aspect and program location.

Formula 2 is about the program's behavior, and we are interested in its *preservation* by advice. We might also want to verify properties that emerge as a result of applying an aspect. For instance, the advice in figure 5 introduces the property

$$\mathsf{AG}(c \rightarrow \mathsf{AF}(u))$$

This property fails of the original program (which has no updating), but is manifest in the system once the aspect is applied. To demonstrate that applying the aspect impacts the establishment of this property, we can instead verify the *negation* of the property—which the program does in fact satisfy. Verifying the advice then *violates* the (negated) property, which 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.)

10. RELATED WORK

There are many efforts to define formal semantics for aspects, some of which have been accompanied by proposals on employing the semantics for verification. For instance, Andrews [3] uses process algebras to offer a foundation for AOP. That 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. [22], later refined by Sereni and de Moor [26]. 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 the shapes of the stacks possible at each 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 [21] 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. [8] use aspects to specify concurrency properties, then synthesize code with appropriate safety protocols and verify the result. Nelson, et al. [24] use both model-checkers and model-builders to verify woven programs. Both Ubayashi and Tamai [31] and Denaro and Monga [7] employ model-checkers to verify Java programs. These papers do not, however, describe a modular verification methodology or address the accompanying subtleties.

There is a growing body of work on techniques to study interference between aspects, such as those of Störzer and Krinke [29] 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). We believe these techniques can strengthen our work.

In a series of papers (e.g., [10]), 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 [14, 18].) 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 [9] also 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 too be able to exploit results on generating environment models under parallel composition. It is, however, unclear what price this model extracts in return for its power, especially given that languages such as AspectJ use sequential composition. His formalization employs alternating-time logic, for which tool support does not appear to be as mature as for CTL.

Mousavi, et al. [23] 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. [25] 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 but do not appear to provide a formal framework for actually performing such reasoning. Tesanovic, et al. [30] perform timing analysis of real-time programs. Their work, however, offers only a very simple model of pointcuts, and does not identify pointcuts in advice.

Xu, et al. [32] reduce 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 study. 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, or how to translate results of verification into a form meaningful to AOP developers.

Sihman and Katz employ "superimpositions", which are aspect-like notations parameterized to be more reusable. Their work helps users of Bandera model-checking [27] 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 [28], they 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. In sum, while their work deals heavily with issues of modularity, they do not appear to have an actual modular analysis.

Modular verification is an old problem, often referred to as assume-guarantee reasoning in the verification community. Most assume-guarantee techniques assume that modules compose in parallel, while AspectJ aspects compose sequentially. Some research [2, 19] has considered modular model-checking with sequential control flow. The original work [19] lacks a design framework that drives the decomposition of the design, and hence is not concerned with the problems of interface generation that we address. The other works use, for instance, hierarchical state machines [2] to provide this decomposition. All of these works, however, assume that the whole program is given at analysis time; in other words, they assume closed rather than the open systems aspects give rise to.

Alur, et al. [1] 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. Their work would nevertheless be useful for extending our results to lift the restriction on nesting depth.

Fisler and Krishnamurthi [12] present a model for verifying product-line systems where each module encapsulates a feature. That work, which inspires our result, addresses the possibility of concurrency within each module, which is not addressed here. However, their composition occurs only at fixed points in the source, corresponding to static joinpoints and ignoring dynamic ones, which we address. Li, et al. [20]

extend these results to address open-system problems that also apply to aspects, but again their work is restricted to a much simpler notion of composition.

11. DISCUSSION AND FUTURE WORK

We have presented a verification technique for modularly analyzing aspects relative to fixed PCDs. This work leaves many questions for future exploration. First, we explore only a very limited source language, excluding features such as exceptions and concurrency. Second, the technique is limited by assumptions such as an inlining depth for the call-stack. Third, we do not address several AspectJ features such as initialization advice. Finally, model extraction tools currently do not address the features of aspect languages.

The execution cost of our algorithm depends on several parameters. While the underlying model checker runs in time linear in the size of the model (which can be the base program or an advice machine), in the worst case each advice must be verified once per state in the pointcut it advises. We believe, however, that in many cases the differences between the labels on states will not matter relative to the advice machine in question, and in such cases we can avoid what is effectively redundant verification. We can potentially identify such related interfaces using deductive techniques.

Finally, this paper presents a technique for preserving those properties of a program that aspects must not invalidate. Aspects, however, engender several other verification scenarios. For one, aspects may themselves introduce properties that the base program must not affect. In addition, an aspect may be applied to "repair" a program's behavior, i.e., to transform a program that "almost" satisfies a property to actually obeying it. These are interesting challenges for future work.

Acknowledgments. We are grateful to Gregor Kiczales for valuable discussions. We thank the anonymous reviewers and Christopher Dutchyn for their detailed comments. We parenthetically toast Dan Friedman, thanks to whom this trio of authors came to know one another over a period of twelve years.

12. REFERENCES

[1] Alur, R., K. Etassami and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

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

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

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

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

[6] 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*, 2000.

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

[8] 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.

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

[10] 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*, October 2002.

[11] 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.

[12] 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.

[13] Fisler, K. and S. Krishnamurthi. Modular verification of feature-oriented software models. Technical Report WPI-CS-TR-02-28, WPI, Department of Computer Science, October 2002.

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

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

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

[17] 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*, June 1997.

[18] 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.

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

[20] Li, H. C., S. Krishnamurthi and K. Fisler. Modular verification of open features through three-valued model checking. *Automated Software Engineering: An International Journal*, 2003.

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

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

[23] 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*, October 2002.

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

[25] 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*, 2002.

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

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

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

[29] Störzer, M. and J. Krinke. Interference analysis for AspectJ. In *Foundations of Aspect-Oriented Languages*, 2003.

[30] 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.

[31] Ubayashi, N. and T. Tamai. Aspect oriented programming with model checking. In *International Conference on Aspect-Oriented Software Development*, pages 148–154, April 2002.

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