MAVEN: Modular Aspect Verification

Max Goldman and Shmuel Katz

Technion — Israel Institute of Technology
{mgoldman,katz}@cs.technion.ac.il

Abstract. Aspects are program modules that include descriptions of key events (called joinpoints) and code segments (called advice) to be executed at those key events when the aspect is bound (woven) to an underlying system. The MAVEN tool verifies the correctness of an aspect relative to its specification, independently of any specific underlying system to which it may be woven. The specification includes assumptions about properties of the underlying system, and guaranteed properties of any system after the aspect is woven into it. The approach is based on model checking of a single state machine constructed using the linear temporal logic (LTL) description of the assumptions, a description of the joinpoints, and the state machine of the aspect advice. The tableau of the LTL assumption is used in a unique way, as a representative of any underlying system satisfying the assumptions. This is the first technique for once-and-for-all verification of an aspect relative to its specification, thereby increasing the modularity of proofs for systems with aspects.

1 Introduction

1.1 Aspect-Oriented Programming

The aspect-oriented approach to software development is one in which concerns that cut across many parts of the system are encapsulated in separate modules called *aspects*. The approach was first presented in the AspectJ [1] extension of Java, and has been generalized to a variety of languages and aspect-oriented software development techniques (see, for example, [2]). When a concern such as security or logging is encapsulated in an aspect, this aspect contains both the code associated with the concern, called *advice*, and a description of when this advice should run, called a *pointcut descriptor*. The pointcut descriptor identifies those points in the execution of a program at which the advice should be invoked, called *joinpoints*. The combination of some *base program* with an aspect (or in general, a collection of aspects), is termed an *augmented program*.

Aspects are of particular interest as a software construct because the pointcuts that govern the execution of their advice are evaluated dynamically. When a pointcut identifies joinpoints, these joinpoints are not static locations in the code; rather, in the most popular and expressive joinpoint models used by aspect-oriented programming languages, joinpoints are well-defined points during the *execution* of a program. Depending on the runtime context of a particular point, such as the methods on the program's stack, or the values currently in certain data fields, the same static code location might match a pointcut at one time, but fail to match it at another. To give the programmer access to these dynamic data, a pointcut may also expose values of program variables to the advice.

1.2 Modular Aspectual Verification

In this work we are concerned with generic formal verification of aspects relative to a specification. The specification of an aspect consists of *assumptions* about any base program to which the aspect can reasonably be woven, and *desired properties* intended to hold for the augmented program (this terminology is applied to aspects in [3]). We view both base programs and aspect code as nondeterministic finite state machines, in which computations are infinite sequences of states within the machine. For both assumptions and desired properties to be verified we consider formulas in linear temporal logic (LTL).

Clearly, given a base program, a collection of aspects with their pointcut descriptors and advice, and a system for *weaving* together these components to produce a stand-alone augmented program, we can verify properties of this augmented system using the usual model checking techniques. Such weaving involves adding edges from joinpoint states of the base program to the initial states of the advice, and from the states at the end of an advice segment to states back in the base program. It would be preferable, however, if we could employ a modular technique in which the aspect can be considered separately from the base program. Instead of examining a particular augmented program, using a generic model of augmented program behavior will allow us to:

- obtain verification results that hold for a particular aspect with any base program from some class of programs, rather than for only one base program in particular;
- use the results to reason about the application of aspects to base programs with multiple evolving state machines describing changing configurations during execution, or to other systems not amenable to model checking; and
- avoid model checking augmented systems, which may be significantly larger than either their base systems or aspects, and whose unknown behavior may resist abstraction.

The second point above relates to object-oriented programs that create new instances of classes (objects) with associated state machine components. Often, the assumption of an aspect about the key properties of those base state machines to which it may be woven can indeed be shown to hold for every possible machine that corresponds to an object configuration of a program. For example, it may involve a so-called *class invariant*, provable by reasoning directly on class declarations, as in [4]. More details on the connections between code-based aspects (as in AspectJ) and the state machine versions are discussed in Sect. 5.

This problem of creating a single generic model that can represent any possible augmented program for an aspect woven over some class of base programs is especially difficult because of the aspect-oriented notion of *obliviousness*: base programs are generally unaware of aspects advising them, and have no control over when or how they are advised. There are no explicit markers for the transfer of control from base to advice code, nor are there guarantees about if or where advice will return control to the base program.

1.3 Results

In this paper we show how to verify once-and-for-all that for any base state machine satisfying the assumptions of an aspect, and for a weaving that adds the aspect advice as indicated in the joinpoint description, the resulting augmented state machine is guaranteed to satisfy the desired properties given in the specification. The verification algorithm is implemented in a prototype called MAVEN. A single generic state machine is constructed from the tableau of the assumption, the pointcut descriptor, and the advice state machine, and verified for the desired properties. Then, when a particular base program is to be woven with the aspect, it is sufficient to establish that the base state machine satisfies the assumptions. Thus the entire augmented program never has to be model checked, achieving true modularity and genericity in the proof. This approach is especially appropriate for aspects intended to be reused over many base programs, such as those in libraries or middleware components.

LTL model checking is based on creating a tableau state machine automaton that accepts exactly those computations that satisfy the property to be verified. Usually, the negation of this machine is then composed as a cross-product with the model to be checked. Here we use the tableau of the assumption in a unique way, as the basis of the generic model to be checked for the desired property. It represents any base machine satisfying the assumption, because the execution sequences of these base programs can be abstracted by sequences in the tableau.

The aspects treated are assumed to be *weakly invasive*, as defined in [5]. This means that when advice has completed executing, the system continues from a state that was already reachable in the original base program (perhaps for different inputs or actions of the environment). Many aspects fall into this category, including *spectative* aspects that never modify the state of the base system (logging is a good example), and *regulative* aspects that only restrict the reachable state space (for example, aspects implementing security checks). Also weakly invasive would be an aspect to enforce transactional requirements, which might roll back a series of changes so that the system returns to the state it was in before they were made. Even a 'discount policy' aspect that reduces the price on certain items in a retail system is weakly invasive, since the original price given as input could have been the discounted one.

Additionally, we assume that any executions of an augmented program that infinitely often include states resulting from aspect advice will be fair (and thus must be considered for correctness purposes). The version here does not treat multiple aspects or joinpoints influenced by the introduction of advice, although the approach can be expanded to treat such cases as well.

In the following section, needed terms and constructs are defined. Section 3 presents the algorithm, and outlines a proof of soundness in the weakly invasive aspect case. This section also uses an abstract example to illustrate the approach.

The MAVEN implementation is described in Section 4, along with descriptions of some typical aspect verifications. Section 5 details works related to the result here, and is followed by the conclusion.

2 Definitions

2.1 LTL Tableaux

Intuitively, the tableau of an LTL formula f is a state machine whose fair infinite paths are exactly all those paths which satisfy the formula f. This intuition will be realized formally in Theorem 1 below.

We define T_f, the tableau for LTL path formula f (equivalently, state formula $\mathsf{A}\, f$), as given in the chapter of [6] on "Symbolic LTL Model Checking," with clarifications described in [7]. We denote $T_f = (S_T, S_0^T, R_T, L_T, F_T)$, where S_T is the set of states; S_0^T is the set of initial states, R_T is the transition relation, L_T is the labeling function, and F_T is the set of fair state sets.

If AP_f is the set of atomic propositions in f, then $L_T : S \rightarrow \mathcal{P}(AP_f)$ — that is, the labels of the states in the tableau will include sets of the atomic propositions appearing in f. A state in any machine is given a particular label if and only if that atomic proposition is true in that state. We also need:

Definition 1. *For path π, let label(π) be the sequence of labels (subsets of AP) of the states of π. For such a sequence $l = l_0, l_1, \ldots$ and set Q, let $l|_Q = m_0, m_1, \ldots$ where for each $i \geq 0$, $m_i = l_i \cap Q$.*

Theorem 1. (from [6], 6.7, Theorems 4 & 5) *Given T_f, for any Kripke structure M, for all fair paths π' in M, if $M, \pi' \models f$ then there exists fair path π in T_f such that π starts in S_0^T and label$(\pi')|_{AP_f}$ = label(π).*

That is, for any possible computation of M satisfying formula f, there is a path in the tableau of f which matches the labels within AP_f along the states of that computation.

In the algorithm of Sect. 3, we restrict the tableau to its reachable component. Such restriction does not affect the result of this theorem, since all reachable paths are preserved, but is necessary in order to achieve useful results. This follows from the observation that the tableau for the negation of a formula has precisely the same states and transition relation, but the complementary set of initial states. Thus, any unreachable portion of the tableau is liable to contain exactly those behaviors which violate the formula of interest.

2.2 Aspects

Advice. An aspect machine $A = (S_A, S_0^A, S_{ret}^A, R_A, L_A)$ over atomic propositions AP is defined as usual for a state machine with no fairness constraint, with the following addition:

Definition 2. *S_{ret}^A is the set of* return states *of A, where $S_{ret}^A \subseteq S_A$ and for any state $s \in S_{ret}^A$, s has no outgoing edges.*

Pointcuts. Recall that a pointcut identifies the states at which an aspect's advice should be activated, and can include conditions on the present state and execution history. We do not give a prescriptive definition for pointcut descriptors; in practice they might take a number of forms, e.g., as in [8] or using variants of regular expressions. Another choice for describing pointcuts might be LTL path formulas containing only past temporal operators. For example, the descriptor $\rho_1 = a \wedge \mathsf{Y}\, b \wedge \mathsf{Y}\,\mathsf{Y}\, b$ would match sequences ending with a state where a is true, preceded by b, preceded by another b (operator Y is the past analogue of X). However expressed, we require that descriptors operate as follows:

Definition 3. *Given a pointcut descriptor ρ over atomic propositions AP and a finite sequence l of labels (subsets of AP), we can ask whether or not the end of l is* matched *by ρ.*

We define $l \models\!\!\mid \rho$ to mean that finite label sequence l is matched by pointcut descriptor ρ in this way.

Specifications. In addition to its advice, in state machine A, and pointcut, described by ρ, an aspect has two pieces of formal specification:

– Formula ψ expresses the assumptions made by the aspect about any base machine to which it will be woven. This ψ is thus a requirement to be met by any such machine.
– Formula ϕ expresses the desired result to be satisfied by any augmented machine built by weaving this aspect with a conforming base machine. In other words, ϕ is the guarantee of the aspect.

2.3 Weaving

Weaving is the process of combining a base machine with some aspect according to a particular pointcut descriptor; the result is an augmented machine that includes the advice of the aspect.

The weaving algorithm has the following inputs:

– aspect machine $A = (S_A, S_0^A, S_{ret}^A, R_A, L_A)$ over AP,
– pointcut ρ over AP, and
– base machine $B = (S_B, S_0^B, R_B, L_B, F_B)$ over $AP_B \supseteq AP$.

And it produces as output:

– augmented machine $\widetilde{B} = (S_{\widetilde{B}}, S_0^{\widetilde{B}}, R_{\widetilde{B}}, L_{\widetilde{B}}, F_{\widetilde{B}})$.

Set AP can be thought of as the 'visible' labels of B with which the aspect is concerned; labels local to the aspect are not included.

The weaving is performed in two steps. First we construct from the base machine B a new state machine B^ρ which is *pointcut-ready* for ρ, wherein each state either definitely is or is not matched by ρ. Then we use B^ρ and A to build the final augmented machine $\widetilde{B}$.

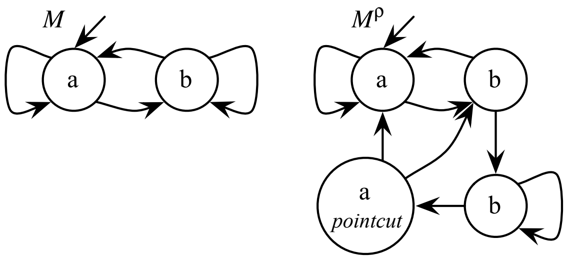


Fig. 1. Constructing a pointcut-ready machine M^ρ for the given M and LTL past formula pointcut descriptor $\rho = a \wedge \mathsf{Y}\, b \wedge \mathsf{Y}\,\mathsf{Y}\, b$

Constructing a Pointcut-Ready Machine. Pointcut-ready machine $B^\rho = (S_{B^\rho}, S_0^{B^\rho}, R_{B^\rho}, L_{B^\rho}, F_{B^\rho})$ is a machine in which unwinding of certain paths has been performed, so that we can separate paths which match pointcut descriptor ρ from those that do not. The pointcut-ready machine contains states with a new label, *pointcut*, that indicates exactly those states where the descriptor has been matched.

This machine must meet the following requirements:

– $S_{B^\rho} \supseteq S_B$
– L_{B^ρ} is a function from S_{B^ρ} to $\mathcal{P}\left(AP_B \cup \{pointcut\}\right)$
– For all finite-length paths $\pi = s_0, \ldots, s_k$ in B^ρ such that $s_0 \in S_0^{B^\rho}$, we have $label(\pi) \models\!\!\!\equiv \rho \Leftrightarrow s_k \models pointcut$.
– For all infinite sequences of labels $l = (\mathcal{P}(AP_B))^\omega$, there is a fair path π_{B^ρ} in B^ρ where $label(\pi_{B^\rho})|_{AP_B} = l$ if and only if there is a fair path π_B in B where $label(\pi_B) = l$.

Note that since B and B^ρ have the same paths (over AP, ignoring the added *pointcut* label), they must satisfy exactly the same LTL formulas over AP.

Figure 1 shows a simple example of this construction. Note that in state diagrams, the absence of an atomic proposition indicates that the proposition does not hold, not that the value is unknown or irrelevant. This is in contrast to a formula, where unmentioned propositions are not restricted.

Finally, note that for a pointcut descriptor that examines only the current state, the splitting and unwinding is unnecessary, and *pointcut* can be added directly to the states in which the pointcut descriptor is matched.

Constructing an Augmented Machine. We construct the components of augmented machine $\widetilde{B} = (S_{\widetilde{B}}, S_0^{\widetilde{B}}, R_{\widetilde{B}}, L_{\widetilde{B}}, F_{\widetilde{B}})$ as follows:

– $S_{\widetilde{B}} = S_{B^\rho} \cup S_A$
– $S_0^{\widetilde{B}} = S_0^{B^\rho}$
– $(s,t) \in R_{\widetilde{B}} \Leftrightarrow \begin{cases} (s,t) \in R_{B^\rho} \;\wedge\; s \not\models pointcut & \text{if } s,t \in S_{B^\rho} \\ (s,t) \in R_A & \text{if } s,t \in S_A \\ s \models pointcut \;\wedge\; t \in S_0^A & \\ \qquad\qquad \wedge\; L_{B^\rho}(s)|_{AP} = L_A(t) & \text{if } s \in S_{B^\rho},\ t \in S_A \\ s \in S_{ret}^A \wedge L_A(s) = L_{B^\rho}(t)|_{AP} & \text{if } s \in S_A,\ t \in S_{B^\rho} \end{cases}$

Note that this relationship is 'if and only if.' In words, the path relation contains precisely all the edges from the pointcut-ready base machine B^ρ and from aspect machine A, except that *pointcut* states in B^ρ have edges only to matching start states in A, and aspect return states have edges to all matching base states.

- $L_{\widetilde{B}}(s) = \begin{cases} L_{B^\rho}(s) \text{ if } s \in S_{B^\rho} \\ L_A(s) \;\text{ if } s \in S_A \end{cases}$
- $F_{\widetilde{B}} = \{F_i \cup S_A \mid F_i \in F_{B^\rho}\}$

From the definition of $F_{\widetilde{B}}$, a path is fair in $\widetilde{B}$ if it either satisfies the original fairness constraint of the pointcut-ready machine, or if it visits some aspect state infinitely many times. A weaving is considered *successful* if every reachable node in $S_{\widetilde{B}}$ has a successor according to $R_{\widetilde{B}}$.

2.4 Weakly Invasive Aspects

As mentioned above, we show our result for the broad class of aspects which, when they return from advice, do so to a reachable state in the base machine. Without this restriction, the aspect may return to unreachable parts of the base machine whose behavior is not bound by assumption formula ψ. In this case, the augmented system contains portions with unknown behavior, and is difficult to reason about in a modular way.

Definition 4. *An aspect A and pointcut ρ are said to be* weakly invasive *for a base machine B if, for all states in S_{B^ρ} that are reachable by following a fair path in $\widetilde{B}$, those states were reachable by following a fair path in B^ρ.*

In particular, this means that all states to which the aspect returns are reachable in the pointcut-ready base machine. This could of course be checked directly, but would require construction of the augmented machine — precisely the operation we would like to avoid. In many cases (see [5]), the aspect can be shown weakly invasive for any base machine satisfying its assumption ψ, by using local model checking, additional information (our reasoning in the discount price example from Sect. 1.3 uses such information), or static analysis (both spectative and regulative aspects can be identified in this way).

3 Algorithm

The modular verification algorithm builds a tableau from base requirement ψ and weaves A with this tableau according to pointcut descriptor ρ, then performs model checking on the augmented tableau to verify desired result ϕ.

Algorithm. Given:

- set of atomic propositions AP;
- assumption ψ for base systems, an LTL formula over AP;
- desired result ϕ for augmented systems, an LTL formula over AP; and
- aspect machine A and pointcut descriptor ρ over AP.

Perform the following steps:

0. For all $a \in AP$, if ψ does not include a, augment ψ with a clause of the form $\cdots \wedge (a \vee \neg a)$, so that ψ contains every $a \in AP$, without altering its meaning.
1. Construct T_ψ, the tableau for ψ. Since ψ contains every AP, the result of Theorem 1 will hold when all labels in AP are considered.
2. Restrict T_ψ to only those states reachable via a fair path.
3. Weave A into T_ψ according to ρ, obtaining $\widetilde{T_\psi}$.
4. Perform model checking in the usual way to determine if $\widetilde{T_\psi} \models \phi$.

This algorithm gives us a sound proof method provided that whenever the model check of the constructed augmented tableau (in step 4 above) succeeds, then for any base system satisfying ψ, applying aspect A according to pointcut descriptor ρ will yield an augmented system satisfying ϕ. This is expressed below:

Theorem 2. *Given AP, ψ, ϕ, A, and ρ as defined, if $\widetilde{T_\psi} \models \phi$, then for any base program M over a superset of AP such that A and ρ are weakly invasive for M, if $M \models \psi$ then $\widetilde{M} \models \phi$.*

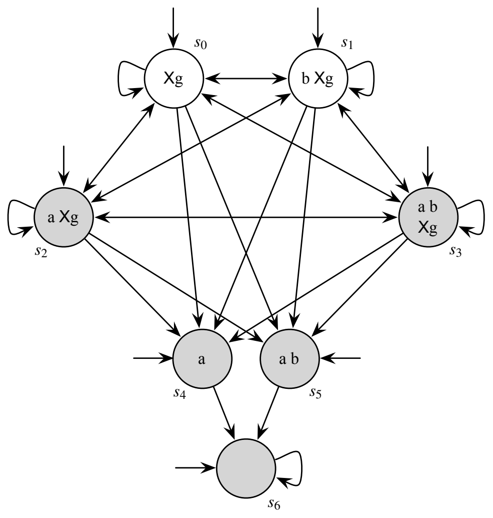
The proof is omitted for reasons of space; it can be found in [7]. It involves an inductive analysis of the paths in the augmented system $\widetilde{M}$ over an arbitrary base system M that satisfies the assumptions ψ. Each such path is shown to correspond to a path in the augmented tableau $\widetilde{T_\psi}$. If the model check in the algorithm succeeded, then all these paths satisfy ϕ, as required.

Although we make use of the entire reachable part of tableau T_ψ, it does not serve as the mechanism for performing LTL model checking, but rather forms (part of) the system to be checked. The tableau for even a complex assumption formula is likely to be much smaller than models of concrete bases systems that satisfy such assumptions. Of course, during the model checking step of the algorithm, which dominates the time and space complexity, any sound optimizations may be employed to reduce the complexity.

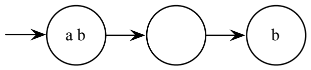
As a first abstract example, suppose we have an aspect with base system assumption $\psi = \mathsf{A}\,\mathsf{G}\,((\neg a \wedge b) \rightarrow \mathsf{F}\,a)$ — that is, any state satisfying $\neg a \wedge b$ is eventually followed by a state satisfying a. We would like to prove that the application of our aspect to any base system satisfying ψ will give an augmented system satisfying result $\phi = \mathsf{A}\,\mathsf{G}\,((a \wedge b) \rightarrow \mathsf{X}\,\mathsf{F}\,a)$ — that is, any state satisfying $a \wedge b$ will eventually be followed by a later state satisfying a.

Figure 2(a) shows the reachable portion of the tableau for the assumption ψ. In the diagram, shaded states are those contained in the only fairness set. The notation $\mathsf{X}g$, not formally part of the state label, designates states in the tableau which satisfy $\mathsf{X}g$ for subformula $g = \mathsf{F}\,a$ (this labeling serves only to differentiate states; other labels of this form have been omitted for clarity, and all such labels become invalid after weaving). For the example pointcut descriptor $\rho = (a \wedge b)$, this tableau machine is also pointcut-ready for ρ (since ρ references only the current state), simply by adding *pointcut* to the labels of s_3 and s_5.

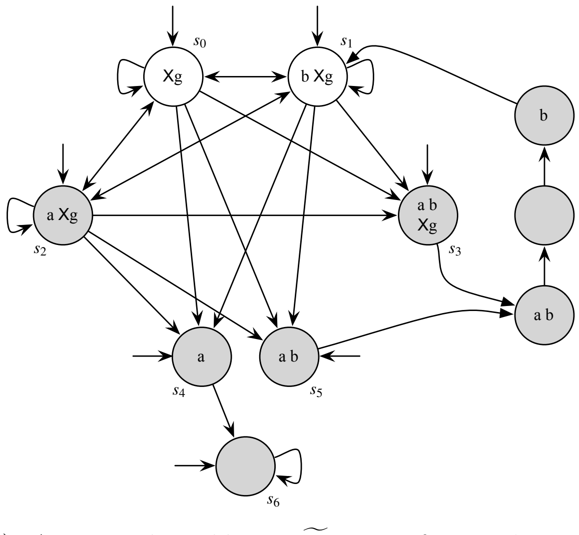
Figure 2(b) shows the state machine A for the advice of our aspect. This advice will be applied at the states matched by ρ, and Fig. 2(c) gives the weaving of



(a) The reachable portion of tableau T_ψ for $\psi = \mathsf{A}\ \mathsf{G}\ ((\neg a \wedge b) \rightarrow \mathsf{F}\, a)$

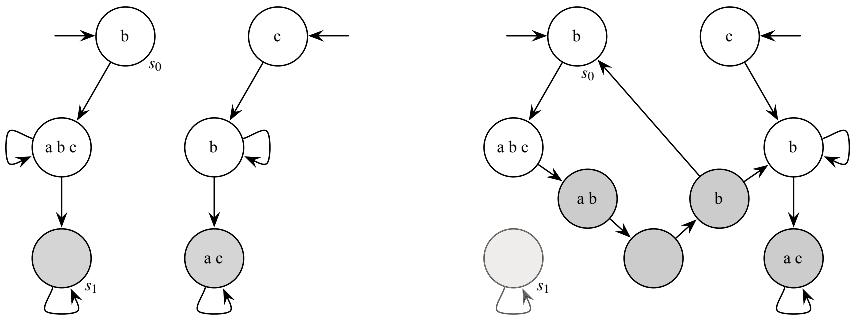


(b) A simple aspect machine A.



(c) Augmented tableau $\widetilde{T_\psi}$, satisfying $\phi = \mathsf{A}\ \mathsf{G}\ ((a \wedge b) \rightarrow \mathsf{X}\ \mathsf{F}\, a)$.

Fig. 2. Example augmented tableau



(a) One particular base machine M. (b) $\widetilde{M}$: M woven with A according to ρ.

Fig. 3. Example weaving where $M \models \psi$ and $\widetilde{M} \models \phi$

A with T_ψ according to ρ. Model checking this augmented tableau will indeed establish that it satisfies the desired property ϕ. This result follows neither from the aspect nor base machine behavior directly, but from their combined behavior mediated by ρ. And since $\widetilde{T_\psi} \models \phi$, any $M \models \psi$ will yield $\widetilde{M} \models \phi$.

Figure 3(a) depicts a particular base machine M satisfying ψ, as could be verified by model checking. Again, the shaded states are those in the only fairness set. Although this M is small, it does contain atomic proposition c not 'visible' to the aspect, and it has a disconnected structure very unlike the tableau.

From Fig. 3(b), one sees it is indeed the case that the augmented machine $\widetilde{M}$ satisfies ϕ — but there is no need to prove this directly by model checking. This holds true even though the addition of the aspect has made a number of invasive changes to M: state s_1 is no longer reachable, because its only incoming edge has been replaced by an advice edge; a new loop through s_0 has been added, while in M there was no path visiting s_0 more than once; there is a new path connecting the previously separated left-hand component to the right-hand; and so forth. In more realistic examples, the difference in size between the augmented tableau (involving only ψ, ρ, and A) and a concrete augmented system with advice over a full base machine would be substantial.

4 MAVEN

The verification algorithm defined in the previous section has been implemented in a prototype system called MAVEN, for "Modular Aspect VerificatioN." In MAVEN, aspects are specified directly as state machines, albeit using a more convenient and expressive language than direct definition of the machine states and transitions. MAVEN operates on the level of textual input to and output from components of the NuSMV model checker [9]. NuSMV is a CTL (branching-time logic) and LTL model checker that accepts its input as textual definitions of state machine systems and their specifications. We have extended the NuSMV finite state machine language to create FSMA, for "finite state machine aspects," which describes aspects and their specifications. The language is based closely on the usual input language of NuSMV, with some added restrictions, and with a collection of new keywords used for aspect-specific declarations:

VAR --BASE. Following this directive, one or more definitions of base machine variables can appear. NuSMV allows the user to specify variables which take their value from a symbolic set or numerical range, in addition to booleans.

VAR --ASPECT. Following this directive, one or more definitions of aspect machine variables can appear.

POINTCUT. Describes the aspect's pointcut. Only current-state expressions are valid; (past) LTL syntax is not permitted. The complete pointcut is taken to be the disjunction of all POINTCUT directives; this allows the user to specify multiple logical pointcuts for the aspect.

INIT. Describes the initial states of the aspect machine.

TRANS. Gives a restriction on the set of valid transitions within the aspect machine. As in NuSMV, the conjunction of all TRANS directives forms the complete restriction. Unlike in NuSMV, TRANS is the only directive available for specifying state machine transitions in FSMA.

RETURN. Describes the return states of the aspect machine. Return states have no outgoing transitions, even if TRANS would indicate otherwise.

LTLSPEC --BASE. Defines an expression which must hold as part of the base system requirement (and is used to build the tableau).

LTLSPEC --AUGMENTED. Defines an expression which must hold as part of the augmented system result (and will be model checked).

From the definition of the POINTCUT directive, one limitation of MAVEN is immediately clear: only pointcuts which are restricted to examining the current state are permitted. That is, this prototype does not include the step of creating pointcut-ready machines during its weaving. However, many pointcut languages and specific applications indeed examine only the current state.

Tableau construction in MAVEN is performed by ltl2smv, an independent component of NuSMV. The ltl2smv program takes as input an LTL formula in the syntax used by NuSMV, and outputs the corresponding tableau state machine. We weave the tableau with the aspect according to the pointcut by modifying this textual representation; the result is a valid NuSMV input file representing the woven tableau and the augmented system results that must hold in it, which can be given directly to the model checker for verification.

The aspects verified while developing and testing MAVEN have not been challenging for the model checker because aspect advice typically contains only relatively short code segments. For all the correctly-specified models verified heretofore, runtime has been measured in seconds, and the number of states generated has been no more than the low thousands.

The MAVEN tool, usage instructions, and a few implemented examples, are available for download at the website noted in [7]. The examples were selected for their ability, in a simple and highly abstract way, to demonstrate real-world situations in which the verification technique is applicable and effective.

In one, the abstract example given earlier is rephrased to describe a more realistic situation: prospective base systems are known to have the property that, whenever a *request* for a status display is made when the *display* is not active at the same time, eventually the *display* will be shown. We wish to use an aspect to guarantee the new behavior that, when the *request* comes while the *display* is already active, a later status *display* will still occur, presumably with updated information. This becomes the formal specification below, which has the same structure as the example:

$$\psi = \mathsf{A}\ \mathsf{G}\ ((\neg display \wedge request) \rightarrow\ \mathsf{F}\, display)$$
$$\phi = \mathsf{A}\ \mathsf{G}\ ((display \wedge request) \rightarrow\ \mathsf{X}\ \mathsf{F}\, display)$$

The construction and model check clearly succeed.

Another example involves the notion of a retail store discount policy aspect, discussed at length in [10]. In the introduction, we noted that such an aspect, when correctly implemented, is in fact weakly invasive, even though it is altering the prices of items. We verified a concrete discount aspect whose specified goal is to implement a "50% Off the Entire Store" policy at the point of purchase. In particular, we showed both a "healthiness constraint" that assuming all prices in the underlying system are nonzero, the same is true of the augmented one, and that the new augmented system has a ceiling on its prices that is half of the previous ceiling. If the aspect code is incorrect, and zero prices can result, MAVEN reports that the aspect is not weakly invasive for the given base, because an aspect return state (one with zero price) differs from all base states, and it displays one such state, provided by NuSMV as its verification failure counterexample.

The use of such counterexamples is further investigated in an example aspect designed to alert users about the occurrence of errors, using an assumption about an existing message delivery system in the base system. By reasoning about the circumstances presented in a counterexample produced from model checking the augmented tableau, we can improve one or more of the specification, pointcut, or advice of our aspect. In general, the need to refine the specification indicates either that our original base system assumption was not strong enough, or possibly that our augmented result was too strong to prove (note that no assumptions are made about the relationship between the formulas, and we can vary them independently). Refining the pointcut can be necessary when the counterexample reveals a situation where the aspect fails to execute advice when it is needed, or activates advice in an inappropriate situation. The advice may need to be altered if the counterexample reveals circumstances under which our original implementation is inadequate; in the case where the advice model has been derived via abstraction from source code, the counterexample could indicate a place where our abstraction needs refinement.

The example fails to verify at first because the assumption is too weak, allowing multiple announcements for the same message; a revision to correct this fails due to situations with overlapping announcements. Correction now requires changing the advice, and ultimately leads to a version that passes verification. The important point is that by using the modular verification method, we were able to reason about the aspect's correctness independent of any particular base machine. Furthermore, the method has forced us to think carefully and precisely about what the aspect will assume, do, and guarantee; precision and certainty being the goal of formal analysis.

5 Related Work

The first work to separately model check the aspect state machine segments that correspond to advice is [11], where the verification is modular in the sense that base and aspect machines are considered separately. The verification method also allows for joinpoints within advice to be matched by a pointcut and themselves advised. However, the treatment there is for a particular aspect woven directly to a particular base program. Additionally, it shows only how to extend properties which hold for that base program to the augmented program (using branching-time logic CTL). A key assumption of their method is that after the aspect machine completes, the continuation is always to the state following the joinpoint in the original base program. This requirement is much stronger than the assumption used here of a weakly invasive aspect.

In [12], model checking tasks are automatically generated for the augmented system that results from each weaving of an aspect. That approach has the disadvantage of having to treat the augmented system, but offers the benefit that needed annotations and set-up need only be prepared once. That work takes advantage of the Bandera [13] system that generates input to model checking tools directly from Java code, and can be extended to, for example, the aspect-oriented AspectJ language. Bandera and other systems like Java Pathfinder [14] that generate state machine representations from code can be used to connect common high-level aspect languages to the state machines used here.

In [5] a semantic model based on state machines is given, and the treatment of code-level aspects and joinpoints defined in terms of transitions, as in AspectJ, is described. The variations needed to express in a state machine weaving the meaning of *before*, *after*, and *around* with *proceed* advice are briefly outlined.

The notion of reasoning about systems composed from two or more state machines is not new, and the most prevalent method for doing so is the assume-guarantee paradigm, which forms the basis of this work. In [15] and [16], among others, an assume-guarantee structure for aspect specification is suggested, similar to the specifications here, but model checking is not used. In [15], proof rules are developed to reason in a modular way about aspect-oriented programs modeled as alternating transition systems; the treatment is for a particular base program in combination with an aspect. And in [16], aspects are examined as transition system transformers, but a verification technique is not introduced.

In most model checking works based on assume-guarantee, the notion of compositionality is one in which two machines are composed in parallel. Composing machine M with M' yields a machine in which composed states are pairs of original states that agree on atomic propositions shared by the two machines. The work of [17] introduced tableaux to modular verification. Under the parallel composition model, no issue analogous to aspect invasiveness arises, because the machines are combined according to jointly-available states.

An alternative mode of verification for composed systems is seen in [18], treating feature-oriented programs built from collections of state machines that implement different features within a system. Consequently, that framework uses a weaving-like process of adding edges between initial and return states of individual machines, but those feature machines explicitly receive and release control over the global state, unlike the oblivious base machines here. Work on extending properties modularly for features is presented in [19].

6 Conclusion

By reusing the notion of a tableau containing all behaviors that satisfy a particular formula, we can achieve a modular verification for aspects. The approach is based on augmenting this tableau with the advice according to a pointcut descriptor and examining the result. In order to do so we must restrict our view to aspects which are weakly invasive and always return to states which were reachable in the original base system. Any computation that infinitely often visits an aspect state is considered fair, to guarantee that it is checked.

A number of directions for future work present themselves. While the current technique only addresses a single aspect and pointcut descriptor, in principle it can be extended to work for multiple aspects, given proper definitions of the weaving mechanics. Further development of how weaving is formulated will also allow treatment of aspects with advice whose addition changes the set of joinpoints. Furthermore, the entire discussion here is given in terms of states and state machines, while, as noted earlier, the usual basic vocabulary of aspect-oriented programming languages refers to events. Problems of real object systems still must be fully expressed in the state-based model checking used here.

Nevertheless, the generic method in this paper allows us for the first time to model check aspects independently of a concrete base program, and already the MAVEN modular aspect verifier can provide useful results. This technique is a significant step toward the truly modular verification of aspects.

References

1. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings ECOOP 2001. LNCS 2072 (2001) 327–353 http://aspectj.org.
2. Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
3. Sihman, M., Katz, S.: Superimposition and aspect-oriented programming. BCS Computer Journal **46**(5) (2003) 529–541
4. Abraham, E., de Boer, F., de Roever, W.P., Steffen, M.: An assertion-based proof system for multithreaded java. Theoretical Computer Science **331**(2-3) (2005) 251–290

5. Katz, S.: Aspect categories and classes of temporal properties. In: Transactions on Aspect Oriented Software Development, Volume 1, LNCS 3880. (2006) 106–134
6. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA (1999)
7. Goldman, M.: Modular verification of aspects. MSc thesis, Technion — Israel Institute of Technology (2006) Available at http://www.cs.technion.ac.il/Labs/ssdl/thesis/finished/2006/max.
8. Sereni, D., de Moor, O.: Static analysis of aspects. In: AOSD'03: Proc. 2nd Intl. Conf. on Aspect-oriented Software Development, ACM Press (2003) 30–39
9. NuSMV. (http://nusmv.irst.itc.it/)
10. Douence, R., Südholt, M.: A model and a tool for Event-based Aspect-Oriented Programming (EAOP). TR 02/11/INFO, Ecole des Mines de Nantes (2002)
11. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularly. In: Proc. SIGSOFT Conference on Foundations of Software Engineering, FSE'04, ACM (2004) 137–146
12. Katz, S., Sihman, M.: Aspect validation using model checking. In: Proc. of International Symposium on Verification. LNCS 2772 (2003) 389–411
13. Hatcliff, J., Dwyer, M.: Using the Bandera Tool Set to model-check properties of concurrent Java software. In Larsen, K.G., Nielsen, M., eds.: Proc. 12th Int. Conf. on Concurrency Theory, CONCUR'01. Volume 2154 of LNCS., Springer-Verlag (2001) 39–58
14. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT) **2**(4) (2000)
15. Devereux, B.: Compositional reasoning about aspects using alternating-time logic. In: Proc. of Foundations of Aspect Languages Workshop (FOAL03). (2003)
16. Sipma, H.: A formal model for cross-cutting modular transition systems. In: Proc. of Foundations of Aspect Languages Workshop (FOAL03). (2003)
17. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems **16**(3) (1994) 843–871
18. Blundell, C., Fisler, K., Krishnamurthi, S., Hentenryck, P.V.: Parameterized interfaces for open system verification of product lines. In: Proc. 19th IEEE International Conference on Automated Software Engineering, ASE'04, Washington, DC, IEEE Computer Society (2004) 258–267
19. Guelev, D.P., Ryan, M.D., Schobbens, P.Y.: Model-checking the preservation of temporal properties upon feature integration. In: Proc. 4th Intl. Workshop on Automated Verification of Critical Systems (AVoCS). Electronic Notes in Theoretical Computer Science 128(6) (2004) 311–324