MAVEN: modular aspect verification and interference analysis

Max Goldman · Emilia Katz · Shmuel Katz

Received: 22 November 2009 / Accepted: 10 October 2010 / Published online: 3 November 2010

Abstract Aspects are program modules that include descriptions of key events (called join-points) 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, and also allows establishing noninterference among aspects, or detecting potential interference. 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 state machine constructed using the linear temporal logic (LTL) description of the assumptions, a description of the join-points, 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. The individual correctness proofs along with proofs of interference freedom are appropriate for a library of reusable aspects, when multiple aspects are to be woven to a system.

Keywords Verification · Aspects · Modularity · Model-checking · Interference · Detection

M. Goldman
Computer Science Department, MIT, Cambridge, MA, USA
e-mail: maxg@MIT.EDU

E. Katz (✉) · S. Katz
Computer Science Department, Technion, Haifa, Israel
e-mail: emika@cs.technion.ac.il

S. Katz
e-mail: katz@cs.technion.ac.il

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 [21] extension of Java, and has been generalized to a variety of languages and aspect-oriented software development techniques (see, for example, [11]). 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 *join-points*. 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 they are not activated by explicit code in the base program and the pointcuts that govern the execution of their advice are evaluated dynamically. When a pointcut identifies join-points, these join-points are not usually only static locations in the code; rather, in the most popular and expressive join-point models used by aspect-oriented programming languages, join-points 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 (both read and write) access to the dynamically changing data, a pointcut may also expose program variables to the advice, extending their scope to the aspect.

1.2 Modular aspect verification

In this work we are concerned with generic formal verification of aspects relative to a specification, and with potential interference among aspects. 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 [28]). 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) because, as will be shown, the automata commonly used in model checking such properties play an essential role in our approach.

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 join-point 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 for each individual aspect 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 particular base program;

- 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 dynamically create new instances of classes (objects) with associated state machine components. Thus, the system representation as a state machine changes, with new components and method calls to objects added or removed. 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 [1]. More details on the connections between code-based aspects (as in AspectJ) and the state machine versions are discussed in Sect. 7.

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 provide formal model checking techniques to establish both that each aspect individually is correct when woven alone, and also to consider possible interference among multiple aspects woven to the same underlying system.

For individual aspects, we show, under some relatively weak restrictions given later in the paper, 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 (only) the aspect advice as indicated in the join-point description, the resulting augmented state machine is guaranteed to satisfy the desired properties given in the specification. Once-and-for-all verification here means that there is no need to repeat the verification process each time we will actually want to weave our aspect into a base system. 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.

Once the individual aspects are known to be correct relative to their specifications, a second verification stage determines whether a collection of aspects can semantically interfere with each other. This means that one causes another to violate its specification when both are woven, even when each is individually correct relative to its specification. This approach is especially appropriate for aspects intended to be reused over many base programs, such as those in libraries or middleware components. Again, proof tasks are defined and automatically checked using MAVEN.

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 paper treats specifications given in LTL precisely because the automata/tableau associated with verifying such properties can be used to represent the desired family of base programs.

The aspects treated are assumed to be *weakly invasive*, as defined in [18] and discussed in Sect. 2.4. 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. There exists an extension of our method to the general case (treating strongly invasive aspects as well), but adding this possibility complicates the treatment, and as there are many aspects of the weakly invasive category, and an easy syntactic check can be performed outside our system to check whether an aspect is weakly invasive, we chose to relate to weakly invasive aspects only in this paper.

Additionally, we assume that any executions of an augmented program that infinitely often include states resulting from aspect advice will be fair (i.e., comprise a fairness set: see clarification in Sect. 2.1), and thus must be considered for correctness purposes. The version here also only treats some cases of join-points influenced by advice.

This paper extends [12] and [17]. Proofs omitted in both papers appear here, and the general theoretic discussion there is extended, formalized and exemplified in the current paper. We also present a new version of the MAVEN tool treating aspects with local memory and extend the library of aspects verified and checked for interference freedom.

In the following section, needed terms and constructs are defined. Section 3 presents the algorithm to model check an individual aspect relative to its specification, and gives a proof of soundness in the weakly invasive aspect case. Section 4 shows how to detect potential semantic interference among aspects, using specially constructed models. This section also uses an abstract example to illustrate the approach. The MAVEN implementation is described in Sect. 5, as is its use both for showing correctness of an individual aspect and detecting any interference among a collection of aspects. Section 6 presents parts of a small aspect library. Section 7 details works related to the results here, and is followed by the conclusion in Sect. 8.

2 Definitions

2.1 LTL Tableaux

The specifications of aspects we consider are written in Linear Temporal Logic [24]. This is a logic over sequences of states that correspond to possible computations. It enables us to express both properties of a single computation and statements about the entire set of computations of a given system. The temporal modalities we use to define properties of a single computation are:

- "G" (meaning, "Globally", from now on). Given a computation π of a system S, a state s in it and a temporal logic formula φ (built from predicates over state variables of S, and temporal logic operators),we say that the temporal logic formula $\mathsf{G}\varphi$ holds at s in π (denoted by $\pi, s \models_S \mathsf{G}\varphi$) if φ holds at every state of π from s and on, including s itself. We also say that π satisfies $\mathsf{G}\varphi(\pi \models_S \mathsf{G}\varphi)$ if $\mathsf{G}\varphi$ holds at the initial state of π.
- "F" (meaning, "Finally", eventually). Given a computation π of a system S, a state s in it and a temporal logic formula φ, we say that the temporal logic formula $\mathsf{F}\varphi$ holds at s in π (denoted by $\pi, s \models_S \mathsf{F}\varphi$) if some state of π in which φ holds can be eventually reached from s (notice that it can be the state s itself). As before, we say that π satisfies $\mathsf{F}\varphi$ ($\pi \models_S \mathsf{F}\varphi$) if $\mathsf{F}\varphi$ holds at the initial state of π.
- "X" (meaning, "neXt", in the next state). Given a computation π of a system S, a state s in it and a temporal logic formula φ, we say that the temporal logic formula $\mathsf{X}\varphi$ holds at s in π (denoted by $\pi, s \models_S \mathsf{X}\varphi$) if φ holds at the state s_1 of π, where s_1 is the next state after s in π. We say that π satisfies $\mathsf{X}\varphi$ ($\pi \models_S \mathsf{X}\varphi$) if $\mathsf{X}\varphi$ holds at the initial state of π. A property described by $\mathsf{X}\varphi$ is called a *next-state property*.
- "U" (meaning, "Until"). Given a computation π of a system S, a state s in it and two temporal logic formulas, φ and ψ, we say that the temporal logic formula $\varphi \mathsf{U} \psi$ holds at s in π if there exists a state s_1 that appears after s in π such that ψ holds in s_1 and φ holds at every state between s and s_1 (including s).

We refer to systems given as a tuple $S = (St_S, S_0^S, R_S, L_S, F_S)$, where St_S is a set of all the states in S, S_0^S is the set of the initial states of S, R_S is the transition relation, L_S is the labeling function, and F_S is the set of fair state sets. A computation π of S is a fair path in the state-transition graph of S, where a fair path is a path that visits each set in F_S infinitely many times.

We say that a computation π of a system S satisfies an LTL formula f (denoted by $\pi \models_S f$) if $\pi, s_0 \models_S f$, where s_0 is the initial state of π. A system S is said to satisfy an LTL formula f ($S \models f$) if every computation of S satisfies f.

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 as given in the chapter of [7] on "Symbolic LTL Model Checking". We denote $T_f = (S_T, S_0^T, R_T, L_T, F_T)$. If AP_f is the set of atomic propositions in f, then $L_T : S \to \mathcal{P}(AP_f)$—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 $label(\pi)$ 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 [7], 6.7, Theorems 4 & 5) *Given T_f, for any Kripke structure M, for all fair paths π' in M, if $\pi' \models_M f$ then there exists a fair path π in T_f such that π starts in S_0^T and $label(\pi')|_{AP_f} = label(\pi)$.*

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 a restriction does not affect the result of this theorem, since all reachable paths are preserved, but, as will be shown, is necessary in order to achieve useful results.

2.2 Aspects

2.2.1 *Advice*

An aspect machine $A = (S_A, S_0^A, S_{ret}^A, R_A, L_A)$ over atomic propositions AP_A 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.

The above described state machine can be constructed either at the design stage, where a model of the aspect is built from the user requirements, or created from the code of the aspect (e.g., by tools like Bandera [14]). The atomic propositions AP_A include all the necessary information about the state of the base system, at the appropriate level of abstraction.

2.2.2 *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., using variants of regular expressions, as in [27]. Another choice for describing pointcuts might be LTL path formulas containing only past temporal operators. For example, the descriptor $\rho_1 = a \wedge \mathsf{Y}b \wedge \mathsf{YY}b$ 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 there exists a suffix of l *matched* by ρ.

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

With appropriate choice of the atomic predicates, past-LTL formulas are expressive enough to describe any AspectJ pointcut. Note that the set AP may contain predicates meaning the computation is now *just before*, or *just after*, some event, thus enabling us to model before and after advice of AspectJ, where the event of interest is usually a method call. Around advice can also be modeled as a combination of a before and an after advice, and in case a method call is to be replaced (in AspectJ terms, no proceed statement is executed), the before part of the advice will be responsible for the appropriate behavior (and the after part will not be executed, as its join-point will not be reached).

2.2.3 *Specifications*

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

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

The form of the specification is an instance of the *assume-guarantee* paradigm but generalized to relate to global properties of the system. The assumption of an aspect can include information on what is expected to be true at join-points, global invariants of the underlying system, or assumed properties of instances of classes or variables that may be bound to various parameters of the aspect when it is woven. The result assertion can include both new properties added by the aspect, and those properties of the basic system that are to be maintained in a system augmented with the aspect. Both parts of aspect specification are expressed in linear temporal logic. Such a specification captures the intension of the aspect.

Definition 4 An aspect is *correct* with respect to its assume-guarantee specification if, whenever it is combined (by itself) with a system that satisfies the assumption, the result will satisfy the guarantee.

With some abuse of notation we will denote by A the aspect as a whole, and not only its advice machine.

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:

- advice machine $A = (S_A, S_0^A, S_{ret}^A, R_A, L_A)$ over $AP_A \supseteq AP$,
- 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 $B + A = (S_{B+A}, S_0^{B+A}, R_{B+A}, L_{B+A}, F_{B+A})$.

The 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 $B + A$.

2.3.1 Constructing a pointcut-ready machine

A 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:

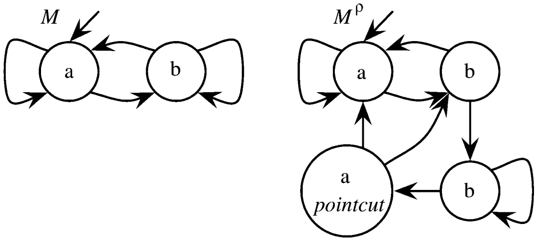


Fig. 1 Constructing a pointcut-ready machine M^ρ for the given M and LTL past formula pointcut descriptor $\rho = a \wedge \mathsf{Y}b \wedge \mathsf{YY}b$

- $S_{B^\rho} \supseteq S_B$
- L_{B^ρ} is a function from S_{B^ρ} to $\mathcal{P}(AP_B \cup \{pointcut\})$
- For all finite-length paths $\pi = s_0, \dots, s_k$ in B^ρ such that $s_0 \in S_0^{B^\rho}$, we have $label(\pi) \models \rho \Leftrightarrow s_k \models pointcut$.
- 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 $label(\pi_B) = l$.

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. The desired separation of the paths matching ρ from all the others in B^ρ is achieved by state splitting, which leads to path unwinding of the relevant paths.

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.

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. In the worst case, the point-ready state machine might grow by two to the nesting depth of past temporal operators in a past-LTL description of the pointcut.

2.3.2 Constructing an augmented machine

We construct the components of augmented machine $B + A = (S_{B+A}, S_0^{B+A}, R_{B+A}, L_{B+A}, F_{B+A})$ as follows:

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

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. Note that the program counter is part of the base system state, which enables us to model the return of the aspect to the appropriate place in the code of the base system. Thus the typical situations where an aspect returns to the point in the code where it was activated, but with some possibly different values for variables, can be correctly modeled by using the program counter and a variable indicating whether the aspect has already been applied.

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

From the definition of F_{B+A}, a path is fair in $B + A$ 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_{B+A} has a successor according to R_{B+A}.

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 P. In this case, the augmented system contains portions with unknown behavior, and is difficult to reason about in a modular way.

Definition 5 An aspect A with pointcut ρ is said to be *weakly invasive* for a base machine B if, for all states in S_{B^ρ} that are reachable in $B + A$ along a fair path are also reachable in B^ρ along a fair path.

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 [18]), the aspect can be shown weakly invasive for any base machine satisfying its assumption P, 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).

2.5 Interference among aspects

Given a library of reusable aspects, each of which is correct w.r.t. its assume-guarantee specification, it is important to check that the aspects will still function properly when woven all together into the same base system.

Definition 6 Given a set of correct aspects, $\mathcal{A}$, we say that $\mathcal{A}$ is *interference-free* if for any subset $\{A_1, \dots, A_n\} \subseteq \mathcal{A}$ the following holds: Whenever $A_1, \dots, A_n$ are woven in any order into a base system that satisfies all the assumptions of the aspects $(P_1, \dots, P_n)$, the augmented system obtained after this weaving satisfies the guarantees of all the aspects in the subset $(R_1, \dots, R_n)$.

Thus if a library of individually correct aspects is proven interference-free, any subset of aspects from this library can be chosen to be added to a given base system, and the augmented system will function properly provided the base system satisfies the assumptions of all the chosen aspects. Moreover, we can decide to add or remove aspects from the system later on, in any order needed during the evolution of our system, and the interference freedom will guarantee proper behavior of the resulting system. Note that some aspects can change the values of variables used by other aspects from the library even if they do not interfere, as long as the correctness of the specification is unchanged.

Remark We demand that the base system into which we would like to weave our aspects satisfies the assumptions of all the aspects in the library, whereas in practice there might be a situation when application of aspect A is possible and desired only after some other aspects, e.g., B and C, have been added to the base system. In such a case we might say that there is a relationship of cooperation between A, B and C, rather than interference. Our method can be easily extended to treat cooperation as well. However, there are many cases when aspects do not depend on the presence of each other in the system, and we concentrate on them in this paper.

2.6 Verification process

Given a library of reusable aspects (with their assume-guarantee specifications), our goal is twofold:

- Verify that each of the aspects in the library is correct. (See algorithm in Sect. 3.)
- Establish the relationship between the aspects in the library: prove interference freedom, or detect interference among some of the aspects. (The algorithm appears in Sect. 4.)

The second part of the verification process is performed incrementally, by adding the aspects one by one to the "verified subset" of the library, until all the aspects are included.

The MAVEN tool, described in Sect. 5, is used to automate both algorithms.

3 Verifying aspect correctness

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

Algorithm

Given:

- set of atomic propositions AP;
- aspect machine A over $AP_A \supseteq AP$ and pointcut descriptor ρ over AP;
- assumption P_A for base systems, an LTL formula over AP_A; and
- desired result R_A for augmented systems, an LTL formula over AP_A.

Perform the following steps:

0. For all $a \in AP$, if P_A does not include a, augment P_A with a clause of the form $\cdots \wedge (a = a)$, so that P_A contains every $a \in AP$, without altering its meaning.
1. Construct T_{P_A}, the tableau for P_A. Since P_A contains every AP, the result of Theorem 1 will hold when all labels in AP are considered.
2. Restrict T_{P_A} to only those states reachable via a fair path.
3. Weave A into T_{P_A} according to ρ, obtaining $T_{P_A} + A$.
4. Check the system $T_{P_A} + A$ for deadlock states. If none exists, continue to the next step. Otherwise, report the deadlock state found.
5. Perform model checking in the usual way to determine if $T_{P_A} + A \models R_A$.

Note: The check in step 4 is optional, and can be used as an additional indication of possible non-weakly-invasiveness of the aspect. A deadlock state is reachable in $T_{P_A} + A$

only in some return state of aspect A, because there are no deadlocks in T_{P_A}, by construction. The meaning of such a deadlock is that A is not weakly invasive for every base system satisfying P_A.

The above algorithm gives us a sound proof method: whenever the model check of the constructed augmented tableau (in step 5 above) succeeds, then for any base system satisfying P_A, applying aspect A according to pointcut descriptor ρ will yield an augmented system satisfying R_A. This is expressed below:

Theorem 2 *Given AP, P_A, R_A, and A as defined, if $T_{P_A} + A \models R_A$, then for any base program M over a superset of AP such that aspect A is weakly invasive for M, if $M \models P_A$ then $M + A \models R_A$.*

As mentioned in Sect. 1, here we assume that A has been shown weakly invasive for M.

The proof of Theorem 2 relies on the definitions and lemmas that appear below together with the intuition for the proof:

In order to prove the theorem we need to show that if the result of weaving A into T_{P_A} satisfies R_A, then for every base system M satisfying P_A, the result of weaving A into M satisfies R_A. For this purpose it is enough to show that for every infinite fair path σ in the woven system $M + A$ there exists a corresponding infinite fair path π in the woven tableau, $T_{P_A} + A$, such that $label(\sigma) \mid_{AP} = label(\pi) \mid_{AP}$. In that case indeed in order to prove that every path in the woven system satisfies R_A, it is enough to show that every path in the woven tableau satisfies this property.

To simplify the notation, let us denote T_{P_A} by T. The task of finding a fair path in $T + A$ that corresponds to a given fair path of $M + A$ will be divided into steps according to prefixes of σ, and at each step a longer prefix will be treated. The following lemma will help to extend the treated prefixes:

Lemma 1 *Let S be a system, and let $s_0, \ldots, s_k$ be states in S such that s_0 and s_k are reachable by a fair path from some initial state of S (the paths and the initial states for s_0 and s_k might be different), and for each $0 \leq j < k$, the transition (s_j, s_{j+1}) exists in S. Then there exists a fair computation in S which contains the sequence of states $s_0, \ldots, s_k$.*

Proof A computation is *fair* if it visits states from the *Fairness set* of the system model infinitely often. Let π_0 and π_k be fair computations in S in which s_0 and s_k occur, respectively. Then $\pi_0 = \sigma_0 \cdot s_0 \cdot \ldots$, and $\pi_k = \ldots \cdot s_k \cdot \sigma_k$ for some σ_0 and σ_k. Let us take $\pi = \sigma_0 \cdot s_0 \cdot s_1 \cdot \cdots \cdot s_k \cdot \sigma_k$. This is obviously a path in S, and it starts from an initial state, as did σ_0. Moreover, π is a fair computation, because it has the same infinite suffix, σ_k, as the fair computation π_k. □

The following definition will be useful for identifying the "interesting" prefixes of the path σ:

Definition 7 Any infinite path π in a transition system can be represented as a sequence of *path segments*—$\pi = \pi^0 \cdot \pi^1 \cdot \ldots$, where each path segment π^i is a sequence of states such that:

- If $i = 0$, the first state of π^i is the initial state of π
- If $i > 0$, the first state of π^i is either an initial state of an advice or a resumption state of the base system (i.e., a state in the base system into which the computation arrives after an advice execution is finished)

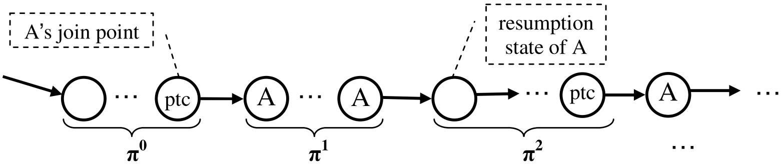


Fig. 2 Example of division of π to path segments

- The last state of π^i is either a pointcut state or a last state of an advice (after which the computation returns to the base system)
- There are no pointcut states and no last states of advice inside π^i (i.e., in the states of π^i that are not the first or the last state)
- π is the concatenation of the path segments of π in the order of their indices

An example of division of a path to path segments is presented in Fig. 2. Note that the decomposition of a path to path segments is unique, and that, because of loops, there can be resumption states within a segment. Note also that we could have an infinite (last) segment—either in (the reachable part of) the base, or in the aspect. In our case all the paths in question are infinite, so the last state of each finite path segment will be either a pointcut or a last state of an advice. Every resumption state is reachable in the system before weaving, as we assumed A to be a weakly invasive aspect with respect to M.

Now if we are given a path of $M + A$, $\sigma = \sigma^0 \cdot \sigma^1 \cdot \cdots$ where σ^i-s are the path segments of σ, for each finite prefix of σ consisting of a number of path segments we define the set of *corresponding path-segment prefixes* of fair paths in $T + A$:

$$\begin{aligned}\Pi_i = \{\pi^0 \cdot \pi^1 \cdot \cdots \cdot \pi^i \mid \\ &label(\pi^0 \cdot \cdots \cdot \pi^i)\mid_{AP} = label(\sigma^0 \cdot \cdots \cdot \sigma^i)\mid_{AP}, \\ &\exists \pi \text{ fair path in } T + A \text{ such that } \pi = \pi^0 \cdot \cdots \cdot \pi^i, \ldots\}\end{aligned}$$

Each element in Π_i is a prefix of an infinite fair computation of $T + A$ corresponding to the i-th prefix of σ, thus the following lemma will show that for every finite prefix of σ there exists a corresponding prefix of a fair computation in $T + A$:

Lemma 2 *Given a fair computation σ of $M + A$, and sets of prefixes Π_i-s as defined above, $\forall i \geq 0. \Pi_i \neq \emptyset$.*

Proof The proof is by induction on i.

Base: $i = 0$ To show that Π_0 is not empty we need to show the existence of π^0 such that $label(\pi^0)\mid_{AP} = label(\sigma^0)\mid_{AP}$ and π^0 is a prefix of some fair path π in $T + A$. σ^0 is the first path-segment of a fair path in $M + A$, thus there is no advice application before σ^0 or inside it. So σ^0 is also the first path segment of a fair computation in M. According to the assumption on M, $M \models P_A$, thus for every fair path starting from an initial state of M there exists a corresponding fair path in T. In particular, there exists a fair path $\pi = t_0, \ldots, t_k, \ldots$ in T such that $label(\sigma^0)\mid_{AP} = label(t_0, \ldots, t_k)\mid_{AP}$. Then again, as $t_0, \ldots, t_k$ is a beginning of a fair path in T, and there are no pointcuts in it, except maybe for the last state, it is also a beginning of a fair computation in $T + A$. So let us take $\pi^0 = s_0, \ldots, s_k$. We are left to show that π^0 is indeed a path-segment, and then it will follow that $\pi^0 \in \Pi_0$, meaning that Π_0 is not empty.

label(t_0) = *label*$((\sigma^0)_0)$, thus t_0 is an initial state of $T + A$. There is no pointcut inside σ^0, because it is a path-segment, so the last state of σ^0 (if exists) cannot be a return state of advice application, which means that it has to be a pointcut state. Due to the agreement on labels, the last state of π^0 will also be marked as a pointcut state (or, if σ^0 is infinite, there will be no last state in π^0). For the same reason, there are no pointcut states among $t_0, \dots, t_{k-1}$, which, in the same way as for σ^0, implies that there are no advice return states also. Thus both ends of π^0 are legal ends of a path-segment, and there are no pointcut states and no advice return states inside π^0, which makes it, indeed, a legal path-segment.

Induction step Let us assume that for every $0 \le i < k$, $\Pi_i \neq \emptyset$. We need to prove that $\Pi_k \neq \emptyset$.

The induction hypothesis holds, in particular, for $i = k - 1$, thus there exists some prefix $\pi^0 \cdot \pi^1 \cdot \dots \cdot \pi^{k-1}$ of a fair computation of $T + A$, corresponding to the prefix $\sigma^0 \cdot \sigma^1 \cdot \dots \cdot \sigma^{k-1}$ of $M + A$'s computation, σ. Let us denote by *s_first*(i) the first, and by *s_last*(i) the last state of i-th path-segment of σ (σ^i), and symmetrically for the states of path segments of $T + A$—by *t_first*(i) the first, and by *t_last*(i) the last state of i-th path-segment. There are two possibilities for *s_last*$(k - 1)$:

1. *s_last*$(k - 1)$ is a pointcut. Then *t_last*$(k - 1)$ is also a pointcut, because due to the induction hypothesis *label*(*s_last*$(k - 1)$) $|_{AP}$= *label*(*t_last*$(k - 1)$) $|_{AP}$. Then in every continuation of the computation both in $M + A$ and in $T + A$ the advice of the aspect is performed, thus the k-th path-segment is in both cases the application of the same advice from the same state, and the agreement on the labels of the k-th path-segments is trivially achieved. Moreover, for the same reason the existence of an infinite fair path with the prefix $\pi^0 \cdot \pi^1 \cdot \dots \cdot \pi^{k-1}$ implies the existence of an infinite fair path with the prefix $\pi^0 \cdot \pi^1 \cdot \dots \cdot \pi^k$, because every continuation of the first prefix had to be an advice application. From the above it follows that in this case $\Pi_k \neq \emptyset$.
2. *s_last*$(k - 1)$ is a last state of the advice. This, in particular, implies that *s_last*(k) is a pointcut state, and no advice has been applied between *s_last*$(k - 1)$ and *s_last*(k). Aspect A is weakly invasive with respect to M, thus *s_last*$(k - 1)$ is a reachable state in M (more precisely, the state reachable in M is the projection of *s_last*$(k - 1)$ on AP). As no advice is applied between *s_last*$(k - 1)$ and *s_last*(k), we have that the whole path-segment σ^k is in the reachable part of M. Moreover, due to Lemma 1, as both *s_last*$(k - 1)$ and *s_last*(k) are reachable by some fair paths from some initial states of M, we also have that there exists a fair computation of M containing the sequence *s_last*$(k - 1)$, *s_first*$(k), \dots,$ *s_last*(k). All the fair computations of the reachable part of M are represented in the tableau T. Thus, in particular, the above fair path has a corresponding path in T, and, as there was no pointcut and no advice application inside the sequence *s_last*$(k - 1)$, *s_first*$(k), \dots,$ *s_last*(k), there are also no pointcuts and no advice applications in the corresponding sequence in the computation of T, and thus there exists a corresponding sequence of states in $T + A$, π^k. The last state of π^{k-1}, *t_last*$(k - 1)$, is reachable from the initial state of $T + A$ by some fair path, as Π_{k-1} is not empty. Moreover, all the possible prefixes of such fair pathes appear in Π_{k-1}, thus at least one of them continues to the sequence π^k. So indeed we obtain that there exists a sequence of states π^k corresponding to σ^k in the woven tableau, for which a fair continuation exists. We are left to see that the sequence of states, π^k, is indeed a path segment in the woven tableau computation. But this is true due to the agreement on labels of the states, *label*(π^k) $|_{AP}$= *label*(σ^k) $|_{AP}$: the path segment σ^k started from a return state of

an advice, ended by a pointcut, and had no advice applications in the internal states, so the same is true for π^k and thus π^k is a path segment.

Thus, indeed, the set of possible continuations, Π_i, is never empty. □

Proof of Theorem 2 Now let us return to the proof of Theorem 2. Let us be given an infinite fair path σ in the woven system $M + A$. From Lemma 2 it follows that there exists an infinite path π in the woven tableau corresponding to the given path σ—all the prefixes of π appear in the Π_i-s above, and due to the lemma, the Π_i-s are all non-empty. So in order to complete the proof of the theorem we need only to notice that every path constructed from the prefixes in Π_i-s above is fair, for the following reason: There are two possibilities for the infinite suffix of π. It either has infinitely many advice applications, or there exists some infinite suffix in which no aspect state is visited. If there are infinitely many advice applications, some state of the advice must be visited infinitely often, and all the states of the advice are defined as fair. If there is no advice application after some state, then there are only a finite number of path segments of π, and the last path segment is infinite. But, as we know, this path segment belongs to some fair path in $T + A$, so this must be a fair suffix, and so the computation π is indeed fair. This completes the proof of Theorem 2. □

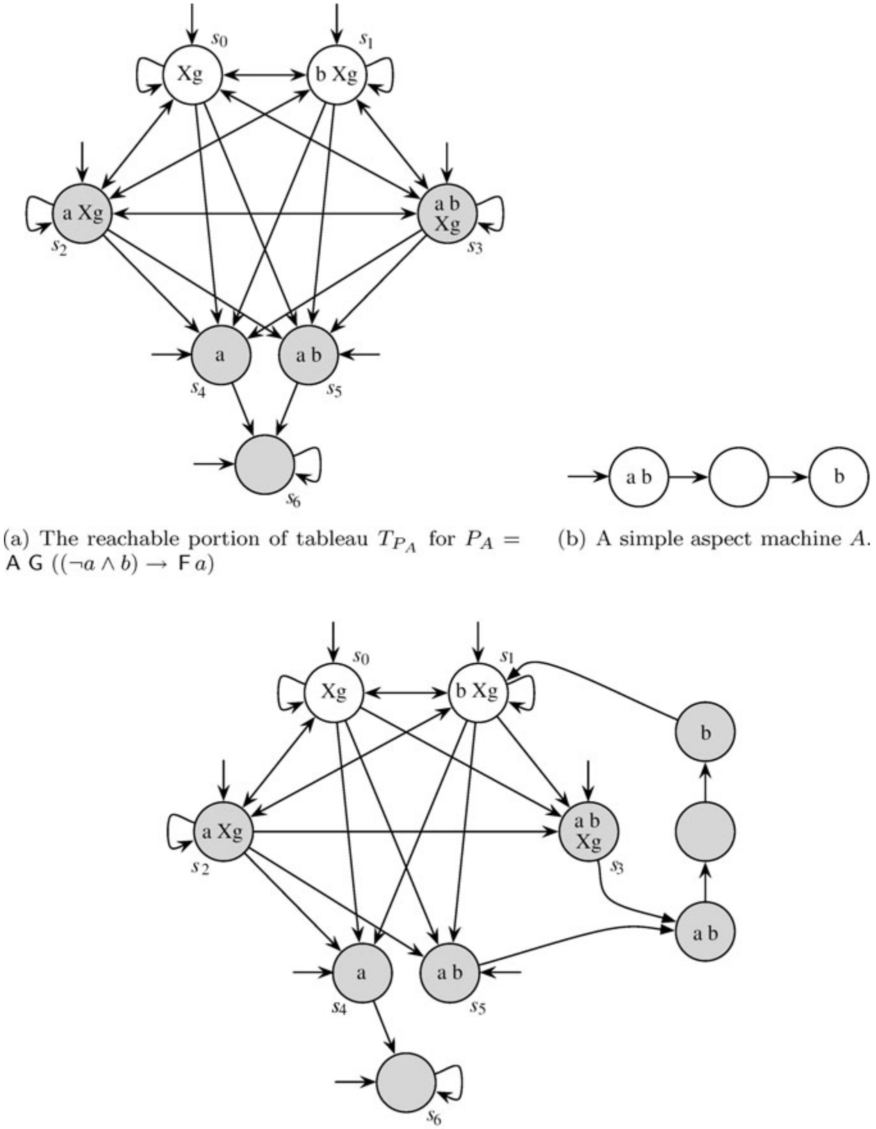
Although in our verification algorithm we make use of the entire reachable part of tableau T_{P_A}, it does not serve as the mechanism for performing LTL model checking, but rather forms (part of) the system to be checked. The tableau for even a complex assumption formula is likely to be much smaller than models of concrete base 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 $P_A = \mathsf{AG}((\neg a \wedge b) \rightarrow \mathsf{F}a)$—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 P_A will give an augmented system satisfying result $R_A = \mathsf{AG}((a \wedge b) \rightarrow \mathsf{XF}a)$—that is, any state satisfying $a \wedge b$ will eventually be followed by a later state satisfying a.

Figure 3(a) shows the reachable portion of the tableau for the assumption P_A. 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 3(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. 3(c) gives the weaving of A with T_{P_A} according to ρ. Model checking this augmented tableau will indeed establish that it satisfies the desired property R_A. This result follows neither from the aspect nor base machine behavior directly, but from their combined behavior mediated by ρ. And since $T_{P_A} + A \models R_A$, any $M \models P_A$ will yield $M + A \models R_A$.

Figure 4(a) depicts a particular base machine M satisfying P_A, 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.



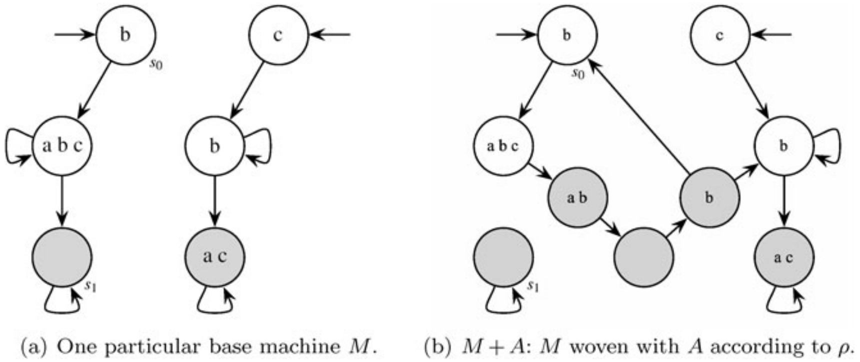
(a) The reachable portion of tableau T_{P_A} for $P_A = \mathsf{A\ G}\ ((\neg a \wedge b) \rightarrow \mathsf{F}\ a)$

(b) A simple aspect machine A.

(c) Augmented tableau $T_{P_A} + A$, satisfying $R_A = \mathsf{A\ G}\ ((a \wedge b) \rightarrow \mathsf{X\ F}\ a)$.

Fig. 3 Example augmented tableau

From Fig. 4(b), one sees it is indeed the case that the augmented machine $M + A$ satisfies R_A—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



(a) One particular base machine M. (b) $M+A$: M woven with A according to ρ.

Fig. 4 Example weaving where $M \models P_A$ and $M + A \models R_A$

augmented tableau (involving only P_A, ρ, and A) and a concrete augmented system with advice over a full base machine would be substantial. This can be seen in Sect. 6, where a library of aspects is described, in which there is a need to verify more realistic aspects and check interference among them.

4 Interference analysis

In a straightforward approach, to be able to establish interference-freedom of a library of aspects in terms of Definition 6 one would have to check all the possible subsets of the library and all the possible orderings of weaving of the aspects in any subset. But in our method, as proven later, it is enough to perform pairwise interference-freedom checks between the aspects in the library in order to ensure interference-freedom of a library as a whole. To simplify the discussion below, we define the following properties of aspect interaction:

Definition 8 Given two correct aspects, A and B, we say that the KP_{AB} *property* holds if for every system S satisfying the assumptions of both A and B weaving A into S preserves the assumption of B. We say that the KR_{AB} *property* holds if the guarantee of A is preserved when weaving B into any system $(S + A)$, that resulted from weaving A into a system S satisfying the assumptions of both A and B. More formally,

$$KP_{AB} \triangleq \forall S[(S \models P_A \land P_B) \rightarrow (S + A \models P_B)]$$

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

$$KR_{AB} \triangleq \forall S[(S \models R_A \land P_B) \rightarrow (S + B \models R_A)]$$

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

The statements KP_{BA} and KR_{BA} are defined symmetrically. If all the four statements—KP_{AB}, KR_{AB}, KP_{BA} and KR_{BA}—are true, A and B are semantically non-interfering. (Theorem 3 below shows this is a special case of Definition 6 for $n = 2$.) In this paper we are only interested in systems for which both A and B are weakly invasive, but the statements are written in more general form, as they will be relevant also for strongly invasive aspect verification and interference detection.

Notice that it might happen that weaving A before B results in violation of the desired properties (KP_{AB} and/or KR_{AB}), but weaving B before A does not, or vice versa: in many cases the result of weaving A before B, $(S + A) + B$, will differ from the result of weaving B before A, $(S + B) + A$, as the advice of the aspect woven first may not apply to the one woven afterwards. For the same reason both orderings above might differ from the result of simultaneous, AspectJ-like, weaving—the relation between them will be discussed in Sect. 4.5. The order of weaving will matter, for example, in the Composition Filters model [3], and in languages with dynamic aspect introduction. Moreover, even in AspectJ, if we first weave A into S and compile the program, and then weave B into the obtained Java bytecode, we do not get the same result as if A and B were woven into S at the same time by the AspectJ weaver.

4.1 Proving interference freedom

The following theorem shows that the condition for non-interference following Definition 8 is a special case of Definition 6 for $n = 2$.

Theorem 3 *Let A and B be two aspects with the specifications* (P_A, R_A) *and* (P_B, R_B) *respectively, and assume that both aspects are correct relative to their specifications. Then to prove that A and B do not interfere, it is enough to show that the statements* KP_{AB}, KR_{AB}, KP_{BA} *and* KR_{BA} *hold.*

Proof According to Definition 6, aspects A and B do not interfere if the following two statements are true:

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

and

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

Let us show that if A and B are correct aspects, and the KP_{AB} and KR_{AB} statements hold, then OK_{AB} holds. The KP_{AB} statement means that the weaving of A into a system S satisfying the assumptions of both aspects does not invalidate the assumption of B, if both aspects are weakly invasive in S. Such an S, in particular, satisfies the assumption of A. We know that after weaving A into a system S that satisfies P_A, R_A is true, for it is assumed that A satisfies its specification. Thus together we have that $(S + A)$ will satisfy not only the assumption of B, but also the guarantee of A, and the following statement will be true:

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

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

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

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

The proof that OK_{BA} follows from KP_{BA} and KR_{BA} is symmetric. Together we obtain that if all the premises of the theorem hold, A and B do not interfere. □

Theorem 4 *Let $A_1, \dots, A_N$ be aspects with the specifications $(P_1, R_1), \dots, (P_N, R_N)$ respectively, and assume all these aspects satisfy their specifications. Assume also that for every pair of indices i, j $KP_{i,j}$ and $KR_{i,j}$ are true. Then the set $\mathcal{A} = \{A_1, \dots, A_N\}$ is interference-free.*

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

Lemma 3 *For every set of $n \geq 2$ aspects $\{A_1, \dots, A_n\}$ satisfying their specifications $(P_1, R_1), \dots, (P_n, R_n)$, if for every pair of indices i, j $KP_{i,j}$ is true, then for every base system S such that $S \models P_1 \wedge \cdots \wedge P_n$, the following holds: For every $0 \leq k < n$, $(\dots(S + A_1) + \cdots + A_k) \models P_{k+1} \wedge \cdots \wedge P_n$ (where the case of $k = 0$ means that no aspects are woven into the system S).*

Proof The proof is by induction on k.

Basis: $k = 0$. We need to show that $S \models P_1 \wedge \cdots \wedge P_n$, but this statement is one of the premises of the lemma.

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

Now let us prove Theorem 4. Let us be given a subset of $1 \leq n \leq N$ aspects from $\mathcal{A}$, and a permutation $(i_1, \dots, i_n)$ of their indices, indicating the chosen weaving order. Without loss of generality, we can call them $(1, \dots, n)$. (Clarification: We can always permute the aspects in the library so that for every j, aspect number i_j will stand on the j-th place. Then the order $1, \dots, n$ on the permuted library will give the same sequence of aspects as the order $i_1, \dots, i_n$ on the original one.) We need to prove that for every base system S, if $S \models (P_1 \wedge \cdots \wedge P_n)$ then $(\dots(S + A_1) + \cdots + A_n) \models R_1 \wedge \cdots \wedge R_n$. The proof is by induction on n.

Basis: If $n = 1$, there is only one aspect, A_1. Let S be a system satisfying P_1. The aspect A_1 satisfies its specification, thus the statement $(S \models P_1) \rightarrow (S + A_1 \models R_1)$ holds.

Induction step: We assume that the statement holds for any $1 \leq k < m$ aspects from the n given, and prove it for $k = m$. Let us be given a base system S satisfying $P_1 \wedge \cdots \wedge P_n$. We will denote by S' the system $(\dots(S + A_1) + \cdots + A_{m-1})$. From the induction hypothesis we have that $S' \models R_1 \wedge \cdots \wedge R_{m-1}$. Lemma 3 is applicable here, so we also have that $S' \models P_m \wedge \cdots \wedge P_n$. In particular, $S' \models P_m$. Thus, as A_m is correct according to its specification, $S' + A_m \models R_m$. And for every $i \neq m$, $1 \leq i \leq n$, the $KR_{i,m}$ property holds, thus from the fact that $S' \models P_m \wedge R_i$ it follows that indeed $S' + A_m \models R_i$. Together we get that indeed $(\dots(S + A_1) + \cdots + A_m) \models R_1 \wedge \cdots \wedge R_m$. □

4.2 Direct interference-freedom proofs

An interference-freedom proof that uses Theorem 3 for pairwise interference-freedom proofs is called an *incremental* proof. Alternatively, we could prove the OK_{AB} and OK_{BA} statements directly, without checking that the assumption of the first woven aspect and the guarantee of the second woven aspect are preserved. However, as opposed to the incremental proofs assumed in Theorem 4, a direct proof of non-interference among pairs of aspects does not generalize to weaving of more than two aspects. As described in Sect. 6.2.2, even if aspects A, B, and C are pairwise interference-free, and are correct relative to their assumptions and guarantees, weaving of all three into a system with $P_A \wedge P_B \wedge P_C$ does not guarantee $R_A \wedge R_B \wedge R_C$ in the resulting system.

Thus the incremental method is essential for showing interference-freedom among groups of aspects of any size. However, the method is incomplete in that there could be aspects that are interference free, but the method will not allow proving it. In particular, by demanding that aspect B will preserve R_A when woven into *any* system that satisfies $R_A \wedge P_B$, we pose too strong a restriction, because we are interested in this statement only for base systems in which aspect A is present.

4.3 Feasible aspect composition

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

Definition 9 Given two aspects A and B with specifications (P_A, R_A) and (P_B, R_B) respectively, the composition of A before B is *feasible* iff all the following formulas are satisfiable: $P_A \wedge P_B$, $R_A \wedge P_B$, $R_A \wedge R_B$.

If a composition of A before B is not feasible, it means that A has to interfere with B. Thus as a first step in detection of interference, a *feasibility check* can be performed—i.e., a satisfiability check on the appropriate formulas. It is recommended to perform a feasibility check before starting the full verification process described in Sect. 5, because this check is much easier and quicker, and then proceed to the verification only if the composition of the aspects is feasible. However, this is not an obligatory stage of the verification process, because if some contradiction exists, the verification process will also detect interference and provide a counterexample.

4.4 Error analysis

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

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

A typical error analysis will be shown for the interference detected in the example of Sect. 6.2.1.

4.5 Joint weaving

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

The following definitions will be useful to us:

Definition 10 Let A and B be two aspects, and S be a system. The result of simultaneous weaving of A and B into S, $S + (A, B)$, is the following system: The set of initial states of $S + (A, B)$ is the same as in S. For each state s in each computation of $S + (A, B)$, if s is a join-point matched by A (and/or B), then the advice of A (and/or B) is executed at s, otherwise one of the enabled transitions of S is executed at s.

Definition 11 Let A and B be two aspects, and S be a system. Let us denote by J the set of all the join-points that are matched by B in S, and by J'—the set of all the join-points that are matched by B in $(S + A)$. We say that A *creates a join-point* matched by B if there exists a join-point $j_1 \in J'$ such that j_1 is not in J (that is, J' is not included in J). We also say that A *removes a join-point* of B if there exists a join-point $j_2 \in J$ such that j_2 is not in J' (that is, J is not included in J').

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

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

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

Proof Since A and B do not have common join-points, at each join-point only one of the advices is inserted, and thus there is no possibility of either changing the order of application of the advices, or achieving different interleavings of their operations (where each interleaving could result in a different combination of operations of the advices in a computation and might violate the specification in some of the cases, and satisfy it in the others).

Moreover, let us notice that the aspects do not create or remove join-points matched by each other, so it does not matter in what order the weaver explores those join-points. The result will be always the same: at each join-point one and only one advice will be applied. Thus indeed $S + (A, B) \cong (S + A) + B \cong (S + B) + A$. □

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

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

Proof Aspect A might create join-points matched by B inside A's advice only, and there is no other modification of the join-points by any of the aspects. Thus all the join-points that appear in the base program will be found and correctly attributed by any weaving. No join-points of A can appear inside the advice of B, so the only potentially problematic join-points are those appearing inside the advice of A. Let us see what will happen to them in each of the weavings.

When the simultaneous weaving is performed, the join-points of B inside A will be identified, and the advice of B will be woven in. The same will happen if we weave B after A. However, when weaving A after B the new join-points will not be identified—the weaver will not look for them, because from its point of view all the aspects except for A have been treated (or do not exist) by the time it comes to weave A in. Thus, indeed, in this case we can only say that $S + (A, B) \cong (S + A) + B$. □

In order to check that the above lemmas can be applied, we need to establish that A and B do not match common join-points. For that purpose existing tools mentioned in Sect. 7 can be used ([9, 16]).

5 MAVEN

The verification algorithm defined in Sect. 3 has been implemented in a prototype system called MAVEN, for "Modular Aspect VerificatioN." This tool is available as part of the Common Aspects Proof Environment (CAPE) [19] developed by the Formal Methods Lab of AOSD-Europe, an EU Network of Excellence. The CAPE is an extensible framework for aspect verification and analysis tools.

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 [6]. 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 the base system variables appear. The possible types of the variables are those defined in NuSMV. Only the variables mentioned in the aspect specification or updated in the advice should be defined.

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

POINTCUT Describes the aspect's pointcut. One predicate appears after each POINTCUT directive. Only current-state expressions are allowed; (past) LTL syntax is not permitted. In order to define a pointcut ρ that contains past LTL operators,the user is required to defined a new state variable (e.g., "ρ_*state*") and divide the pointcut definition into two parts: add the statement "POINTCUT ρ_*state*" to the list of POINTCUT directives, and add the assumption "G(ρ_*state* $\leftrightarrow \rho$)" to the assumptions of the aspect. The complete pointcut definition is the disjunction of all POINTCUT directives; this allows the user to specify multiple logical pointcuts for the aspect. One or more POINTCUT directives should be present.

GLOBINIT Initialization of the aspect variables in the woven system is defined by the conjunction of all the GLOBINIT directives. These directives are optional, and if no directive is given for some aspectual variable, no restriction will be posed on its initial value in the woven system.

LOCINIT Initial states of the aspect machine are defined by the conjunction of all the LOCINIT directives. These directives are optional.

LOCMEM A list of aspect local memory variables, separated by ",", follows this directive. The values of these variables will be preserved between the executions of the advice. This list is optional.

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. These are the only restrictions on the transitions of the aspect machine, thus any pair of states for which all the TRANS predicates hold is considered to be included in the aspect transition relation (exception—RETURN states, see below).

RETURN One state predicate (involving aspect and/or base system variables) appears after each RETURN directive. The disjunction of all the RETURN directives defines the state(s) in which the control should return from the advice back to the base system. These states have no outgoing transitions in the advice machine, even if some transitions are permitted by TRANS.

ONRET One next-state statement follows each ONRET directive. If the value of an aspect variable should be changed while returning from the advice machine to the base system (for example, reset to its initial value), the ONRET directive is used. The conjunction of all the ONRET directives defines the next state of the aspect variables after returning from the advice.

LTLSPEC --BASE Defines an LTL formula that should hold in the base system as part of the aspect requirements. The conjunction of all the LTLSPEC --BASE directives is the complete precondition of the aspect, used to construct the assumptions tableau.

LTLSPEC --AUGMENTED Defines an LTL formula that should hold in the woven system as part of the guarantee of the aspect. The conjunction of all the LTLSPEC --AUGMENTED directives is the complete resulting assertion of the aspect, which will be model-checked.

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 textual representation of a corresponding tableau state machine. Note that during the construction of the assumption tableau, it is automatically made pointcut-ready due to the additional assumption statements inserted for past-LTL pointcut definitions. 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.

A usage guide for MAVEN and a movie demonstrator are available at [5], as well as a detailed description of some implemented examples. One of the examples is a variant of aspect E described in Sect. 6 (the example on the web page is called "Security Aspect").

Our interference detection method is based on Theorem 3, and it also uses MAVEN as a subsystem. To show that weaving aspect A before B does not lead to interference, perform the following steps:

1. For KP_{AB}, build a tableau that corresponds to the conjunction of the assumptions of the aspects, $P_A \wedge P_B$, weave the advice of A and show that the assumption of B, P_B, is true of the result. That is, run MAVEN to show $T_{P_A \wedge P_B} + A \models P_B$.
2. For KR_{AB}, build a tableau that corresponds to the conjunction of the assumption of B and the guarantee of A, $R_A \wedge P_B$, weave the advice of B, and show that the guarantee of A, R_A, still holds for the result. That is, run MAVENto show $T_{R_A \wedge P_B} + B \models R_A$.
3. If in both cases the woven models built are deadlock-free, and both verifications succeed, then aspect A can be woven before B.

The incremental proof that B can be woven before A is symmetric. The verification can be preceded by a feasibility check, for error analysis (see Sect. 4.4).

Note: in order to be able to rely on MAVEN verification results the following two assumptions are needed: the basic assumption that A and B are both weakly invasive for any base system S to which they will be woven, and an additional assumption that though aspect A is allowed to add join-points for aspect B, and even contain join-points of B inside the advice, aspect B remains weakly invasive in the system $S + A$. (Additional discussion on restriction of the possible influence of one aspect on join-points of another appears in Sect. 4.5.) These two assumptions are sufficient for all the verification tasks above. The case of KP_{AB} check is obvious, as, by Theorem 2, $T_{P_A \wedge P_B} + A$ contains all the computations of $S + A$. The soundness of the KR_{AB} check follows from the next two observations: First, if $S + A$ satisfies both R_A and P_B, then all its computations are represented in $T_{R_A \wedge P_B}$. And second, due to the assumption that B remains weakly invasive in $S + A$, Theorem 2 is applicable again. Thus indeed all the computations of $(S + A) + B$ are represented in the woven tableau $T_{R_A \wedge P_B} + B$.

The above incremental method is sound, due to the above note and Theorems 3 and 4, but, as already noted at the end of Sect. 4.2, not complete. In addition to the inherent incompleteness, there are practical reasons for not always succeeding in proving non-interference. First, the model checking itself may not succeed. If the model is infinite, or finite but too large, the model-checking will collapse without providing any answer. So, as always when model-checking, the models and the properties should be described at a sufficient level of abstraction. Second, the specification of some aspect may not be as general as possible. Specifically, the assumption of aspect B, P_B, may not be the weakest possible, or the guarantee of A, R_A, may not be the strongest possible. In the first case, as P_B is not the weakest possible, aspect A might not preserve the assumption of aspect B, but assures some other property, P'_B, that is enough for aspect B to operate correctly. Then the KP_{AB} check fails, but the OK_{AB} is true. In the second case, symmetrically, it might happen that we cannot prove that aspect B preserves the guarantee of A, because the assumption $R_A \wedge P_B$ is not strong enough to ensure R_A after B is woven, but the OK_{AB} property is true because A actually guarantees a stronger statement, R'_A, and with this assumption B is able to preserve R_A (for every system S, if $S \models R'_A \wedge P_B$, then $S + B \models R_A$).

6 Example: aspect library verification

In this section we present some parts of the verification process for a library of reusable aspects. A typical library for reusable aspects could deal with concerns like communication security, or system backup for fault-tolerance. Some aspects from such a library, and their verification process, are described below.

In order to be able to check whether one aspect preserves an assumption or a guarantee of another one, we need to be able to model the influence of the operations of the advice on the variables to which the assumption or the guarantee refers. By default, we assume that advice of an aspect does not change values of the variables that did not appear in its own specification and description. But as variables that appear in our models and specifications are abstraction of actual system variables, sometimes different variables have a semantic connection which we need to preserve in our specifications and models. For our examples, the following connections were identified and used:

- ($login\_psw\_send \rightarrow psw\_send$) (sending a password from a login screen is a special case of sending a password in the system)
- ($(psw\_in\_usr \wedge send\_usr) \rightarrow psw\_send$) (sending user data that contains a password implies that a password is sent in the system).

6.1 Aspect descriptions

We will consider four aspects.

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

$$P_E \triangleq \mathsf{G}(psw\_send \leftrightarrow login\_psw\_send),$$

$$R_E \triangleq \mathsf{G}(psw\_send \rightarrow encrypted\_psw)$$

where the predicate *psw_send* means that a message containing a password is being sent, and *login_psw_send* means that the password is being sent from the login screen. The pointcut of E can be given as a state predicate *login_psw_to_send*, that becomes true each time before a message is to be sent from a login screen. It is related to the *login_psw_send* predicate in a way that each state where *login_psw_send* holds is preceded (but maybe not immediately) by a unique state in which *login_psw_to_send* is true. This relationship is part of our general knowledge about the base system, and is expressed in the following addition to the assumption of the aspect:

$$\mathsf{G}(\mathit{login\_psw\_send} \rightarrow ((\neg \mathit{login\_psw\_send})\ \mathsf{S}\ (\mathit{login\_psw\_to\_send} \wedge \neg \mathit{login\_psw\_send})))$$

Aspect M provides the possibility to "remember" the user's password in the system, so that the user will not have to type the password during subsequent log-ins. To add this functionality to the system, M should add some introductory operation, e.g., a new checkbox which, when checked, indicates that the password should be stored for the user. The advice of M might add a private field "password" to the User class, and store the password there after the checkbox is checked. A partial specification for M thus is:

$$P_M \triangleq \mathit{true}$$

(M does not need to assume anything about the base system, as the checkbox is added by M itself), and

$$R_M \triangleq [\mathsf{G}(\mathit{req\_store\_usr\_psw} \rightarrow (\mathit{req\_store\_usr\_psw} \mathsf{U}(\mathit{psw\_in\_usr})))]$$

where *req_store_usr_psw* means that "remember my password" has been checked, and *psw_in_usr* means that a password appears as part of the user data. The moment the predicate *req_store_usr_psw* becomes true is the pointcut of the aspect. Note that before M is woven into a base system, objects of the User class might, or might not, contain the password as part of the base system activity, but after weaving M they surely do.

Aspect B can "back up" user data, to increase fault-tolerance: it sends all the user data to the backup server upon request. The aspect does not need to assume anything about the base system, and guarantees that if there is a request for backup, all the data of the user will be sent. More formally:

$$P_B \triangleq \mathit{true},$$

$$R_B \triangleq [\mathsf{G}(\mathit{req\_backup} \rightarrow (\mathit{req\_backup} \mathsf{U}(\mathit{send\_usr})))]$$

where the predicate *req_backup* means that there is an unprocessed request for sending user data, and *send_usr* means that all the user data is sent. The moment the predicate *req_backup* becomes true is the pointcut of the aspect.

Aspect F provides a list of security questions to the user, and if the questions are answered correctly, F guarantees that the user will get his password via an e-mail, and thus retrieves a forgotten password. Like aspect M, aspect F might add a new button—"Forgot my password"—to the system so that we can define the pointcut of F as the moment when this button is pressed. F's advice then provides the dialog with questions, checks the answers, and in case all the answers are correct—sends an e-mail to the user. More formally, F's assumption is

$$P_F \triangleq \mathit{true}$$

And F's guarantee is

$$R_F \triangleq [\mathsf{G}((\mathit{button\_pressed} \wedge \mathit{quest\_answered}) \rightarrow \mathsf{F}(\mathit{psw\_send}))]$$

where *button_pressed* is the flag that means forgetting the password has been reported and not yet treated. The moment the predicate *button_pressed* becomes true is the pointcut of the aspect.

Intuitively, the three aspects E, M, and B together interfere, and cannot be woven together in one system, because aspect B violates the guarantee of E, by sending passwords unencrypted after M saves them. Weaving both E and F can also be problematic, as will be shown in detail later. Although the interferences are easy to see here, when large libraries are considered, automatic interference analysis is needed.

6.2 Aspect verifications

First of all, the verification procedure described in Sect. 3 has been applied to the aspects, and they have been shown correct w.r.t. their assume-guarantee specifications. The detailed descriptions below refer to the pairwise interference checks performed to check interference freedom of the library.

6.2.1 Encrypting passwords and retrieving forgotten passwords

As part of our interference checks, we would like to verify that weaving the passwords-encrypting aspect (E) and the aspect retrieving a forgotten password (F) into the same system is possible.

Let us check OK_{EF} incrementally. F's assumption is *true*, thus E can not violate it. Thus in order to check the possibility of weaving F after E, we need to prove only that the weaving of F maintains the guarantee of E (the KR_{EF} statement):

$$\forall S[(S \models \mathsf{G}(psw\_send \rightarrow encrypted\_psw)) \quad \rightarrow$$
$$(S + F \models \mathsf{G}(psw\_send \rightarrow encrypted\_psw))]$$

This statement seems to be reasonable, and the feasibility check succeeds, but the advice of aspect F is implemented in such a way that the password sent from it is not encrypted. Thus when trying to verify the KR_{EF} statement, a counterexample is obtained. It is a computation in which at some state s_1 the predicate *button_pressed* became true, and at the same time the predicate *encrypted_psw* was false. Two states after that, at a state s_2, due to the operation of the aspect F, *quest_answered* became true (while *button_pressed* was still true), and in the next state, s_3, *psw_send* became true. But F does not encrypt the passwords, thus *encrypted_psw* was still false at s_3, contradicting the implication in R_E, so the verification failed.

In order to check the possibility of weaving E after F, we need to prove that the weaving of F to a system satisfying both assumptions maintains the assumption of E (the KP_{FE} statement):

$$\forall S[(S \models \mathsf{G}((psw\_send \leftrightarrow login\_psw\_send) \wedge (true)) \quad \rightarrow$$
$$(S + F \models \mathsf{G}(psw\_send \leftrightarrow login\_psw\_send))]$$

However, the implementation of the advice of F leads to a violation of the assumption of E, because F does not send the password from the login screen. Note that in this case, again, there is no contradiction in the specifications of E and F, so the feasibility check succeeds, and the interference is detected during the verification only. In this example, the conflicting aspects do not share any join-points, and the interference does not emerge from updating common variables.

At http://www.cs.technion.ac.il/ssdl/pub/SemanticInterference/ the whole cycle of verification for a variant of these aspects is presented: from AspectJ code to abstract models in

Table 1 Execution statistics for verifications and interference checks of aspects E and F. ‘Model size’ refers to the number of BDD nodes generated

Check type	Result	Model size	Verification time (msec)
$AspectE$	true	1127	26
$AspectF$	true	718	23
KP_{EF}	true	1374	22
KR_{EF}	false	1283	22
KP_{FE}	false	2375	30
KR_{FE}	true	2450	29

the MAVEN input format, followed by verification of each aspect alone w.r.t. to its assume-guarantee specification, and then interference checks for the two aspects.

Verification results described above and some statistics on running the example are summarized in Table 1. After obtaining the results, error analysis, as described in Sect. 4.4, was performed for the cases in which interference was detected. For example, for the case of KP_{FE} we discovered that in this example the composition of aspects is feasible. If it would be found unfeasible, we would know that a change of specification(s) is required, and in this case the specification(s) should have been weakened. But as the specifications are not contradictory, we do not have to change them. In such a case if neither of the assumptions is too strong, a change in one advice, or in both, is necessary. For instance, in our example we can change the advice of F to bring the user to a version of the login screen where the password can be changed, instead of sending the e-mail with the password. In this case, if E is woven after F, the password-sending operation of F is done by the user as another login-password send and thus will be a legal join-point of E. Therefore the advice of E will be performed and no password will be sent unencrypted. More formally: the specification of F can stay the same, but as a result of the change in the advice, whenever *psw_send* is true, so is *login_psw_send*. Aspect E and its specification will stay as before. Now the verification will be of F’s new code relative to the specifications, so that KP_{FE} and KR_{FE} now will hold. This means that the sequential weaving of first F and then E is possible. Notice, however, that weaving first E and then F would still be problematic.

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

Note that the detected interference does not mean that we can never add the above two aspects to the same base system, even if the aspects and their specifications are not changed. The result of our verification only means that we cannot do so without additional checks, because we only state here that it is not true that the two aspects can be woven together into *every* base system satisfying both of their assumptions. If we still want to add the two unmodified aspects together to a given base system, an in-depth analysis of the particular base system is required, and it might be the case that in this specific system the two aspects would succeed to work together.

6.2.2 Three-way interference

Recall the aspects E (encrypting passwords), M (remembering passwords in the user’s data), and B (backup of the user’s data). They have been shown correct w.r.t. their assume-guarantee specifications.

These aspects interfere when all three are woven. The incremental verification method succeeds to detect interference by pairwise checks only (see Table 2), in spite of the fact

Table 2 Pairwise interference checks results

Verification task	Result	Comment
E vs. B		
KP_{EB}	true	no need to check: P_B is *true*
KR_{EB}	false	invariant $psw\_send \rightarrow encrypted\_psw$ violated, and weaving fails
KP_{BE}	false	a state where a password is sent not from the login screen can now be reached
KR_{BE}	true	
E vs. M		
KP_{EM}	true	no need to check: P_M is *true*
KR_{EM}	true	
KP_{ME}	true	
KR_{ME}	true	
B vs. M		
KP_{BM}	true	no need to check: P_M is *true*
KR_{BM}	true	
KP_{MB}	true	no need to check: P_B is *true*
KR_{MB}	true	

that each pair is possible alone. By Theorem 4 we know it should, and indeed, there are two checks that fail for our example: The KP_{BE} check fails, as a state where a password is sent not from the login screen can now be reached, violating E's assumption. The KR_{EB} check fails as well, because R_E alone (i.e., the fact that the passwords are always sent encrypted when E is woven to the base system) does not imply that the passwords are not stored as part of the user data, and thus when aspect B sends all the user data, the passwords might be sent unencrypted. Note that with a slight change of specification, a variant of this example can be created where the direct verification method will fail to detect interference among the aspects, and only the incremental method will work.

Error analysis performed for this example showed that part of the above discovered interference could be repaired, e.g., by saving the password in an encoded form in aspect M.

7 Related work

7.1 Modular aspect verification

The first work to separately model check the aspect state machine segments that correspond to advice is [23], where the verification is modular in the sense that base and aspect machines are considered separately. The verification method also allows for join-points within advice to be matched by a pointcut and themselves advised. However, the treatment there is for aspects 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). Aspects treated in that paper can influence the control flow of the base system, for example, in the case of around advice. However, all the aspects treated are assumed not to modify data variables of the base system, thus not all possible weakly invasive aspects can be analyzed.

In [20], 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 [14] 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 [15] that generate state machine representations from code can be used to connect common high-level aspect languages to the state machines used here.

In [18] a semantic model based on state machines is given, and the treatment of code-level aspects and join-points 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 [8] and [29], among others, an assume-guarantee structure for aspect specification is suggested, similar to the specifications here, but model checking is not used. In [8], 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 [29], 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 [13] 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 [4], 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 [10].

7.2 Checking interference among aspects

The way in which assume-guarantee specifications of aspects described in Sect. 2.2 are used to define interference freedom is analogous to interference freedom among processes in shared-memory systems [25]. In that classic work, interference freedom among processes is defined in terms of whether independent and local Hoare-logic proofs of correctness for each parallel process are invalidated by operations from other processes. The individual proofs that each aspect is correct when woven alone correspond to the n local proofs of [25], while the interference-freedom checks for aspects correspond to the n^2 checks of interference-freedom among processes. A key point, also adapted here, is that the other processes may change the values of shared variables, but there is no interference as long as the independent proofs are not invalidated. The level of interleaving in shared memory systems is much finer than for aspects: every local assertion about memory values can be invalidated by another assignment by a different processor. The fact that the code of the aspect (the advice) is only activated at join-points means that less stringent conditions can be used, and that modular model checking can be used as a proof component.

The work on interference detection presented here, expanding preliminary work presented at a workshop [17], is the first definition of semantic interference for aspects that uses the specification of the aspects as the interference criterion, and applies model checking to detect interference or establish noninterference among collections of aspects. The interference checks are performed on pairs of aspects, and the results of these pairwise checks are sufficient to determine interference freedom for all the aspects in the library. However, as shown in Sect. 5, to enable such incremental proofs we have to "pay" by additional incompleteness.

There has been previous work on detecting whether the pointcuts of aspects match common join-points or there are overlapping introductions [9, 16]. This is important because the semantics of weaving can be ambiguous at such points, and be the source of errors. However, as has been shown, aspects can interfere even if there are no common join-points. Some work has also been done in identifying potential influence by using dataflow techniques showing that one aspect changes (or may change) the value of some field or variable that is used and potentially affects the computation done by the advice of another aspect [26, 31]. Slicing techniques for aspects [2, 30, 32] can also be used for such detection. Since such potential influence is often harmless, many false positives can result.

8 Conclusions

When given a library of aspects, two questions immediately arise: The first question is whether each of the aspects by itself is correct, i.e., satisfies its assume-guarantee specification when woven alone into a suitable base system. The second question is whether the guarantee of some aspect can be invalidated as a result of weaving it together with additional aspects into the same base system (then we say that there is interference among these aspects). The current paper presents an automatic and modular way to answer the two questions above. Both the individual correctness check and the interference checks are modular, whereas the interference-freedom detection requires only pairwise checks in order to detect interference of any subset of aspects from the library.

Modularity of aspect verification means its independence from any concrete base system. Thus most of the verification can be done as the library is built, and aspects are added to it, rather than when a collection of aspects from the library is to be used in an application. When the user would like to weave one or more aspects from the library into some base system, the only check that should be performed is that the base system satisfies the assumptions of all the aspects that will be added to it.

Modularity is achieved by reusing the notion of a tableau containing all behaviors that satisfy a particular formula.

Both in verification of a single aspect, and in interference checks, the result is more informative than just "yes" or "no". If a check of an individual aspect fails, the model-checker provides a counterexample, which is a possible computation of a system with this aspect that violates the guarantee. And interference checks of a library do not only result in stating whether or not the current library is interference-free. For each aspect we know with which aspects it does not interfere, and also for every aspect with which an interference exists, we know what is the cause of the interference, and in which order of weaving it occurs. All this information can serve as usage guidelines for the developers who would like to use aspects from the verified library. In case the library as a whole is not interference-free, a developer might chose some interference-free subset of the library (recall that pairwise interference-freedom of the aspects in any set is enough to guarantee interference-freedom of the set as a whole), or decide on an appropriate weaving order of the aspects to prevent interference.

As future work we plan to apply our verification method and develop effective usage guidelines for a larger existing library of aspects (e.g., [22]). Also in the future we plan to remove some of the restrictions in the present system, e.g., to treat strongly invasive aspects, and shared join-points. The present version already provides an effective tool for building provably correct reusable libraries of aspects, and for detecting semantic errors and interference among aspects.

References

1. Abraham E, de Boer F, de Roever W, Steffen M (2005) An assertion-based proof system for multithreaded Java. Theor Comput Sci 331(2–3):251–290
2. Balzarotti D, D'Ursi A, Cavallaro L, Monga M (2005) Slicing AspectJ woven code. In: Proc of foundations of aspect languages workshop (FOAL05)
3. Bergmans L, Aksit M (2001) Composing crosscutting concerns using composition filters. CACM 44:51–57
4. Blundell C, Fisler K, Krishnamurthi S, Hentenryck PV (2004) Parameterized interfaces for open system verification of product lines. In: Proc 19th IEEE international conference on automated software engineering (ASE'04), pp 258–267
5. Common aspect proof environment (CAPE) (2008). http://www.cs.technion.ac.il/~ssdl/research/cape
6. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NuSMV: a new symbolic model verifier. In: CAV'99. Lecture notes in computer science, vol 1633. Springer, Berlin, pp 495–499. http://nusmv.itc.it
7. Clarke EM Jr., Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
8. Devereux B (2003) Compositional reasoning about aspects using alternating-time logic. In: Proc of foundations of aspect languages workshop (FOAL03)
9. Douence R, Fradet P, Sudholt M (2004) Composition, reuse, and interaction analysis of stateful aspects. In: Proc of 3th intl conf on aspect-oriented software development (AOSD'04). ACM, New York, pp 141–150
10. Guelev DP, Ryan MD, Schobbens PY (2004) Model-checking the preservation of temporal properties upon feature integration. In: Proc 4th intl workshop on automated verification of critical systems (AVoCS'04). Electron Notes Theor Comput Sci 128(6):311–324
11. Filman RE, Elrad T, Clarke S, Akşit M (eds) (2005) Aspect-oriented software development. Addison-Wesley, Boston
12. Goldman M, Katz S (2007) MAVEN: Modular aspect verification. In: Proc of TACAS 2007. Lecture notes in computer science, vol 4424. Springer, Berlin, pp 308–322
13. Grumberg O, Long DE (1994) Model checking and modular verification. ACM Trans Program Lang Syst 16(3):843–871
14. Hatcliff J, Dwyer M (2001) Using the Bandera Tool Set to model-check properties of concurrent Java software. In: Larsen KG, Nielsen M (eds) Proc 12th int conf on concurrency theory, CONCUR'01. Lecture notes in computer science, vol 2154. Springer, Berlin, pp 39–58
15. Havelund K, Pressburger T (2000) Model checking Java programs using Java PathFinder. Int J Softw Tools Technol Transf (STTT) 2(4)
16. Havinga W, Nagy I, Bergmans L, Aksit M (2007) A graph-based approach to modeling and detecting composition conflicts related to introductions. In: AOSD '07. ACM, New York, pp 85–95
17. Katz E, Katz S (2008) Incremental analysis of interference among aspects. In: FOAL '08. ACM, New York, pp 29–38
18. Katz S (2006) Aspect categories and classes of temporal properties. Trans Aspect-Oriented Softw Dev 1:106–134. LNCS 3880
19. Katz S, Faitelson D The common aspect proof environment. Submitted for publication, 2009. See http://www.cs.technion.ac.il/~ssdl/research/cape/index.html
20. Katz S, Sihman M (2003) Aspect validation using model checking. In: Proc of international symposium on verification. Lecture notes in computer science, vol 2772. Springer, Berlin, pp 389–411
21. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of AspectJ. In: Proc ECOOP 2001. Lecture notes in computer science, vol 2072. Springer, Berlin, pp 327–353. http://aspectj.org
22. Kienzle J, Duala-Ekoko E, Gélineau S (2009) AspectOptima: A case study on aspect dependencies and interactions. Trans Aspect-Oriented Softw Dev 5:187–234
23. Krishnamurthi S, Fisler K (2007) Foundations of incremental aspect model-checking. ACM Trans Softw Eng Methodol (TOSEM) 16(2)

24. Manna Z, Pnueli A (1991) The temporal logic of reactive and concurrent systems: specification. Springer, Berlin
25. Owicki S, Gries D (1976) An axiomatic proof technique for parallel programs. Acta Inform 6:319–340
26. Rinard M, Salcianu A, Bugrara S (2004) A classification system and analysis for aspect-oriented programs. In: Proc of international conference on foundations of software engineering (FSE04)
27. Sereni D, de Moor O (2003) Static analysis of aspects. In: AOSD, pp 30–39
28. Sihman M, Katz S (2003) Superimposition and aspect-oriented programming. BCS Comput J 46(5):529–541
29. Sipma HB (2003) A formal model for cross-cutting modular transition systems. In: Proc of foundations of aspect languages workshop (FOAL03)
30. Storzer M, Krinke J (2003) Interference analysis for AspectJ. In: Proc of foundations of aspect languages workshop (FOAL03)
31. Weston N, Taiani F, Rashid A (2007) Interaction analysis for fault-tolerance in aspect-oriented programming. In: Proc workshop on methods, models, and tools for fault tolerance, MeMoT'07, pp 95–102
32. Zhao J (2002) Slicing aspect-oriented software. In: IEEE international workshop on programming comprehension, pp 251–260