Analysis and Detection of Interactions Among Aspects

Emilia Katz
Analysis and Detection of Interactions Among Aspects

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Emilia Katz

Submitted to the Senate of the Technion — Israel Institute of Technology
Cheshvan 5771 Haifa November 2010
The research thesis was done under the supervision of Prof. Shmuel Katz in the Computer Science Department.

I wish to express my sincere gratitude to my supervisor, Prof. Shmuel Katz, for his guidance and kind support.

The generous financial support of the Technion and the AOSD-EUROPE Network of Excellence is gratefully acknowledged.
Contents

Abstract 1

1 Introduction 3

2 Background on LTL, NuSMV and MAVEN 8
  2.1 LTL 8
  2.2 Representation of systems 10
    2.2.1 Base system 10
    2.2.2 Aspects 10
  2.3 Specification and Verification for Weakly Invasive Aspects 13
    2.3.1 Weakly Invasive Aspect Specification 13
    2.3.2 Modular Verification of Weakly Invasive Aspects 14

3 Modular Verification of Strongly Invasive Aspects 17
  3.1 Verification theory for strongly invasive aspects 17
    3.1.1 Refined Aspect Specification 17
    3.1.2 Refined Tableau Construction 18
  3.2 Algorithms 26
    3.2.1 Computing $L_A$ Automatically 26
    3.2.2 Determining the Aspect Category 27
    3.2.3 Verifying the Aspect 31
    3.2.4 Base System Correctness Verification 31
  3.3 Example 33
  3.4 Related Work 37

4 Incremental Analysis of Interference Among Aspects 39
  4.1 Semantic Interference Among Aspects 39
4.2 Interference analysis ........................................... 40
  4.2.1 Proving Interference Freedom ............................ 41
  4.2.2 Direct interference-freedom proofs ...................... 45
  4.2.3 Feasible aspect composition .............................. 45
  4.2.4 Error Analysis .......................................... 46
  4.2.5 Joint Weaving ........................................... 48
4.3 Proof Implementation ........................................ 51
4.4 Related Work ................................................ 54

5 User Queries for Specification Refinement Treating Shared
Aspect Join Points ................................................. 56
  5.1 Semantics of aspect behavior at a common join point .... 56
  5.2 Specification of aspects with possibly shared join-points .. 64
    5.2.1 Guided Specification Construction .................... 64
    5.2.2 Using the Refined Specification ....................... 69
  5.3 Related Work ............................................... 69

6 Additional Extensions of Modular Aspect Verification ............ 71
  6.1 Aspects With Local Memory .................................. 71
  6.2 Aspects with Full Past-LTL Pointcuts ...................... 72
  6.3 Aspects that Might Have Shared Join-Points ............... 74

7 Example Library ................................................ 76
  7.1 Aspect Specification ....................................... 76
    7.1.1 Aspect Descriptions .................................. 78
    7.1.2 Specification Refinement ............................. 82
  7.2 Interference Check of the Library .......................... 86
    7.2.1 Encrypting Passwords and Retrieving Forgotten Pass-
         words .................................................. 87
    7.2.2 Three-way interference ................................ 89
    7.2.3 Interference Detection in the Presence of Shared Join-
         Points ................................................ 90

8 Conclusions ................................................................ 92

Abstract in Hebrew ................................................ x
List of Figures

3.1 Refined assume-guarantee specification for strongly invasive aspects. \( R_A \) is guaranteed in the woven system if \( PR_A \) holds in the reachable part of \( S \), and \( L_A \rightarrow PU_A \) holds in the computations starting from previously unreachable states of \( S \).

4.1 Incremental interference-detection procedure for a library of correct aspects.

5.1 Pointcut occurrence example.

5.2 Cases of influence of aspects at a shared join-point.

6.1 Example of aspect with local memory: logging messages before they are sent.

7.1 Specification refinement for L4.

7.2 Interference checks summary.
Abstract

Aspect-oriented programming is becoming a common approach to extend object systems with modules that cross-cut the usual class hierarchy. Aspects encapsulate treatment of concerns that otherwise would be scattered within an underlying application, and tangled with code treating other concerns.

Several issues are treated that extend the applicability of proof methods for aspects, with emphasis on interference among aspects.

One extension is for treatment of aspects that might bring the computation to a state previously unreachable in the base system. Such aspects are termed strongly invasive and they were not covered by previous verification methods. An extended specification for aspects, and a new verification method based on model checking are presented. They are used to establish the correctness of strongly invasive aspects, independently of any particular base program to which they may be woven.

Often, insertion of several aspects into one system is desired and in that case the problem of interference among the aspects might arise, even if each aspect individually woven is correct relative to its specification. In this type of interference, one aspect can prevent another from having the required effect on a woven system. Such interference is defined and an incremental proof strategy is presented based on off-line model checking of pairs of aspects for a generic model expressing the specifications. When an aspect is added to a library of non-interfering aspects, only its interaction with each of the aspects from the library needs to be checked. Such checks for each pair of aspects are proven sufficient to detect interference or establish interference freedom for any collection of aspects in a library.

Additional extension is needed for the case when multiple aspects can share a join-point. In this case they may, but do not have to, semanti-
cally interfere, and a specification refinement might be necessary to enable modular verification and interference detection among aspects even in the presence of shared join-points. An in-depth analysis of aspect semantics and mutual influence of aspects at a shared join-point is presented, in order to enable distinguishing between potential and actual interference among aspects at shared join-points. An interactive semi-automatic procedure for specification refinement is described, that will help users define the intended aspect behavior more precisely.

Extensions to the MAVEN system implementing the theory are described, and a case study of specification and interference checks for a library of aspects is given.
Chapter 1

Introduction

Aspect-oriented software development is becoming a common approach to extend object systems with modules (aspects) that cross-cut the usual class hierarchy, and enable modular treatment of different concerns, such as debugging, security, fault tolerance, and many others. The treatment of such concerns, without aspects, would have been scattered over different classes or methods of the base system, or would have been mixed with code treating other concerns. An aspect consists of two parts: the code associated with concern treatment (called advice), and a predicate defining when advice should be applied during system executions (called a pointcut descriptor, or - more briefly - a pointcut). A pointcut can refer both to static and dynamic information about the state of the system, i.e., identify not only the place in the code, or other conditions that can be statically evaluated, but also runtime conditions under which the advice should be invoked (such as values of system variables, or call stack contents). Thus, when a computation arrives at one and the same place in the code several times, sometimes the corresponding state may be matched by the pointcut, and sometimes it may not. The points in the execution that are identified by a pointcut are called join-points. The process of combining aspects with a system is called weaving, the original system is then referred to as the base system, and the result - as the woven system. The best-known aspect-oriented language is the AspectJ [27] extension of Java, but there are many aspect-oriented languages and software development techniques (see, e.g., [14]). In this work we model aspects and systems as state machine abstractions, that can either come from early designs or be derived from code. In this way we are able
to support full AspectJ aspects, including ones with local memory. This is explained in more detail in Section 2.2.

The possibility of adding an aspect, or even multiple aspects, to systems gives rise to many important questions of applicability and correctness, such as: to which base systems can a given aspect be applied, what is its intended influence on a base system, how should the potentially applicable advice pieces be woven at a join-point, and how do multiple aspects woven into the same base system interact with each other? Answering these questions requires:

1. formal definitions of the semantics of an individual aspect and of non-harmful aspect interactions,
2. formal specifications of aspects,
3. a verification procedure to check that an individual aspect is correct,
4. showing that there are no harmful interactions among multiple aspects intended to be woven into the same base system.

Our work has contributions in all the areas above.

In order to be able to define the intended behavior of an aspect, and describe the possible base systems into which this aspect can be woven, a formal specification for aspects is needed. A modular treatment of aspects, without referring to a particular base system into which they will be woven, is desired, in order to reduce the size of verified models, and to allow convenient reuse of aspects in a library. Such an approach requires that the aspect itself have an independent specification that can be shown to hold. In one form or another, the specification of an aspect describes an assumption about any base system to which the aspect can be woven, and a guarantee about the resultant system after the aspect is woven. In previous work [16] an approach to verification was developed: the aspects are shown correct relative to their specification, and then, for each system to be constructed with the aspects, the base system is shown to satisfy the assumptions of the needed aspects. The construction of a model of the entire concrete woven system (which might be considerably larger than either of those used in the modular verification) and its direct verification do not have to be carried out at all.
However, so far, when aspects are treated separately from a specific weaving, it has been necessary to add a restriction: that the aspect returns control to the base system in a state that already existed for some computation of the base system without the aspect woven into it. Such aspects are called weakly invasive in [24], where the other categories of aspects mentioned in this work are also defined (See also Chapter 2). In our work we remove this restriction and show that a modular approach can be realized even for aspects of the most general category - so-called strongly invasive aspects that do return control to the base system in new states that were unreachable in the base system executing alone (for example, due to some base system invariant that is invalidated by the aspect). To do this, we take advantage of the usual organization of model checkers for linear time systems, and of the facilities they commonly provide. An extension of the maven [16] aspect verification system is presented, that can treat strongly invasive aspects, and an example of a bonus aspect for student grades is described. This aspect offers a possibility to give a bonus for assignments and/or exams, which sometimes leads to grades exceeding the usual 0..100 range. This is a violation of a base system invariant, thus the aspect is strongly invasive.

The basic idea of the new approach is to add to the specification an assumption about the base system that restricts the computation segments that may become reachable after a strongly invasive aspect is woven. We then show once-and-for-all that when the aspect is woven into any base system with a reachable part that satisfies the previous type of assumption and an unreachable part that satisfies the added one, the result of the weaving will satisfy the guarantee. For a particular base system, we then have to show that the assumptions are true for both the reachable and unreachable parts (or at least the unreachable part that may become reachable after weaving). These tasks are made feasible due to the fact that many model checkers actually generate a state transition system that includes the unreachable parts of the computation, as a side-effect of the construction, and that marking the reachable states is a built-in operation.

Often, multiple aspects are woven together into the same base system. There even exist libraries of reusable aspects, e.g., in [28], where a library of aspects was created to implement the ACID properties for transactional objects. Once the individual aspects are known to be correct relative to their
specifications, a second verification stage is needed to determine 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. We give a precise definition of semantic interference among aspects using the specification of the aspects as the interference criterion, show how to detect it, and how to use examples of interference to modify the aspects or their specifications. We define an incremental proof strategy based on model-checking that establishes whether there exists a legal underlying system in which the aspects interfere. Our interference detection procedure performs pairwise interference checks of aspects in the library, and these pairwise checks are shown sufficient to imply non-interference of all the possible subsets of aspects in the library. When an interference check of a pair of aspects fails, error analysis is proposed to find out which of the aspects was responsible for the failure, and what steps can be taken to eliminate the interference, depending on the type of the failure. Sometimes interference cannot be eliminated, and then the user is guided not to weave the two aspects together in the same system (either at all, or in some particular order). Our verification technique is most appropriate for establishing usage guidelines for reusable aspects, especially as libraries of reusable aspects are developed.

All the above verification methods rely on a correct formalization of aspect specification. However, writing a specification in a precise and formal way is not an easy task, as it is hard for the user to think of all the relevant details. After the specification is almost completed, there still might be some delicate special cases not treated by it, not due to the lack of information, but from the fact that the user might not notice that this information is indeed relevant, and also might not immediately know how to formalize it. The goal of the next part of our work is to connect informal knowledge of the user with the input expected by formal verification tools, especially model checkers, verifying Temporal Logic specifications. We present a semi-automatic interactive procedure for specification refinement, that will help the user refine the specification in such cases.

In the procedure we propose, the user is asked a series of questions in natural language, regarding the parts of system behavior that are important for the delicate case treated. Based on the answers, statements in Temporal Logic are automatically created, to be added to a previously existing
specification. The semi-automatic creation of properties uses parametrized formulae, which are a result of previous analysis of the semantics of the special case in question. In this work we apply our approach to aspect semantics and mutual influence of aspects at a shared join-point.

The possibility of aspects woven into the system to have common join-points is widely recognized as potentially problematic, and gives rise to several key questions and problems, from understanding how the potentially applicable advice pieces should be woven at such a common join-point (as they cannot be applied all at the same time), to conflict detection and resolution, since application of one advice might interfere with the computation or even applicability of another. Application of our approach to this delicate situation results in an easily automatizable interactive procedure. It helps users specify their intentions for aspect behavior in a specific system more precisely, and this, in turn, enables our above-mentioned interference-detection method to check whether there is actual interference with respect to this specification.

The last part of the thesis describes the implementation of these methods and their application to examples.

The rest of this work is organized as follows: In Chapter 2, the necessary background information is presented. Chapter 3 describes our extension of aspect specification and modular verification to strongly invasive aspects. In Chapter 4 our modular interference-detection procedure appears. In Chapter 5 we describe the user-guided specification refinement procedure treating the case of shared aspect join-points. Chapter 6 presents additional improvements to the MAVEN aspect verification tool, and in Chapter 7 we discuss an example aspect library and application of our verification, specification and interference detection procedures to it. We conclude in Chapter 8. The related work for each topic is discussed at the end of the chapter devoted to that topic. Chapter 3 has appeared in [21], while Chapter 4 was partially covered in a preliminary work [20] and in [15], where, additionally, parts of Chapter 2 and Chapter 7 are covered. Chapter 5 is partially covered in [23]. Parts of Chapter 7 also appear in [20, 21, 23].
Chapter 2

Background on LTL, NuSMV and MAVEN

This chapter gives some general background on LTL and model-checking, followed by background on aspects, and, finally, by a description of the basic approach to specification and model-checking of an individual aspect from [16].

2.1 LTL

The specifications of aspects we consider are written in Linear Temporal Logic [32]. 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. For both assumptions and desired properties to be verified we consider formulas in LTL because, as will be explained later, the tableau associated with verification of LTL properties is central to the verification method used in the original MAVEN tool and in this thesis. The temporal modalities we use to define properties of a single computation are:

- “G” (meaning, “Globally”, from now on). Given a computation $\pi$ of a system $S$, a state $s$ in it and a temporal logic formula $\varphi$ (built from predicates over state variables of $S$, and temporal logic operators), we say that the temporal logic formula $G\varphi$ holds at $s$ in $\pi$ (denoted by

\[G \varphi\]
• “F” (meaning, “Finally”, eventually). Given a computation $\pi$ of a system $S$, a state $s$ in it and a temporal logic formula $\varphi$, we say that the temporal logic formula $F \varphi$ holds at $s$ in $\pi$ (denoted by $\pi, s \models_{S} F \varphi$) if some state of $\pi$ in which $\varphi$ holds can be eventually reached from $s$ (notice that it can be the state $s$ itself).

• “O” (meaning, “Once”) - dual to “Finally”. Given a computation $\pi$ of a system $S$, a state $s$ in it and a temporal logic formula $\varphi$, we say that the temporal logic formula $O \varphi$ holds at $s$ in $\pi$ if there exists a state $s_1$ that occurs before $s$ in $\pi$ such that $\varphi$ holds in $s_1$.

• “X” (meaning, “neXt”, in the next state). Given a computation $\pi$ of a system $S$, a state $s$ in it and a temporal logic formula $\varphi$, we say that the temporal logic formula $X \varphi$ holds at $s$ in $\pi$ (denoted by $\pi, s \models_{S} X \varphi$) if $\varphi$ holds at the state $s_1$ of $\pi$, where $s_1$ is the next state after $s$ in $\pi$. A property described by $X \varphi$ is called a next-state property.

• “U” (meaning, “Until”). Given a computation $\pi$ of a system $S$, a state $s$ in it and two temporal logic formulas, $\varphi$ and $\psi$, we say that the temporal logic formula $\varphi U \psi$ holds at $s$ in $\pi$ if there exists a state $s_1$ that appears after $s$ in $\pi$ such that $\psi$ holds in $s_1$ and $\varphi$ holds at every state between $s$ and $s_1$ (including $s$).

• “W” (meaning, “Weak until”). Given a computation $\pi$ of a system $S$, a state $s$ in it and two temporal logic formulas, $\varphi$ and $\psi$, we say that the temporal logic formula $\varphi W \psi$ holds at $s$ in $\pi$ if either $G \varphi$ or $\varphi U \psi$ hold at $s$. Meaning that either $\varphi$ holds at every state of $\pi$ from $s$ and on, or there exists a state $s_1$ that appears after $s$ in $\pi$ such that $\psi$ holds in $s_1$ and $\varphi$ holds at every state between $s$ and $s_1$ (including $s$).

We say that a computation $\pi$ 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 $\pi$. A system $S$ is said to satisfy an LTL formula $f$ ($S \models f$) if every computation of $S$ satisfies $f$.  

$\pi, s \models_{S} G \varphi$ if $\varphi$ holds at every state of $\pi$ from $s$ and on, including $s$ itself.
2.2 Representation of systems

2.2.1 Base system

We refer to systems given as a tuple $M = (S_M, S^M_0, Rel_M, Lab_M, F_M)$, where $S_M$ is a set of all the states in $M$, $S^M_0$ is the set of the initial states of $M$, $Rel_M$ is the transition relation, $Lab_M$ is the labeling function, and $F_M$ is the set of fair state sets. A computation $\pi$ of $M$ is a fair path in the state-transition graph of $M$, where a fair path is a path that visits each set in $F_M$ infinitely many times.

The labeling function, $Lab_M$, matches each state $s \in S_M$ with its label — the set of all the atomic predicates from $AP_M$ that hold at $s$. This set is also denoted by $label(s)$. A label of a path $\tau$, $label(\tau)$, is defined to be the sequence of the labels of the states of $\tau$, so that if $\tau = s_0, s_1, s_2, \ldots$, then $label(\tau) = (label(s_0), label(s_1), label(s_2), \ldots)$.

2.2.2 Aspects

Advice

An aspect machine $A = (S_A, S^A_0, S^A_{ret}, Rel_A, Lab_A)$ over atomic propositions $AP_A$ is defined as usual for a state machine with no fairness constraint, with the following addition:

**Definition 1** $S^A_{ret}$ is the set of return states of $A$, where $S^A_{ret} \subseteq S_A$ and for any state $s \in S^A_{ret}$, $s$ 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 [17]). The atomic propositions $AP_A$ include all the necessary information about the state of the base system, at the appropriate level of abstraction. Aspects that have a local memory can also be modeled, as explained later in Section 6.1.

The following definition is useful when we need to refer only to some of the labels of a state (for example, in a state of the advice examine only atomic propositions relevant to the base system):

**Definition 2** Given a path $\pi$ with $label(\pi) = l_0, l_1, \ldots$, and a set of labels $Q$, let $l|Q = m_0, m_1, \ldots$ where for each $i \geq 0$, $m_i = l_i \cap Q$. 10
Pointcuts

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 [39]. Another choice for describing pointcuts might be LTL path formulas containing only past temporal operators. For example, the descriptor \( ptc_1 = a \land Y b \land YY b \) 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 \( ptc \) 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 \( ptc \).

We define \( l \models ptc \) to mean that finite label sequence \( l \) is matched by pointcut descriptor \( ptc \) 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).

Weaving

Weaving is the process of combining a base with an aspect according to the aspect pointcut descriptor. In [16], only state predicates were allowed as pointcut descriptors. In case more complex, history dependent pointcuts were needed, it was assumed that a manual state splitting has been applied to the base system to make it “pointcut ready”, meaning that after the splitting every state would either always or never be matched by
the pointcut. However in Chapter 6, among the other improvements to the 
verification process, we describe a technique that automatically does the 
necessary state-splitting and makes it possible to handle general past-LTL 
pointcut descriptors.

When the base system and the aspect advice are given as state-transition 
systems as defined above, the weaving process performed at this level of 
abstraction is as follows:

- Every join-point \( s \) in the base state machine \( M \) is connected to the 
corresponding initial state of the advice (a state \( s_1 \in S^A_0 \) such that 
\( Lab_M(s) = Lab_A(s_1)|_{AP_M} \)), instead of its former next states, and the 
links to the former next states are removed.

- Every state \( s \in S^A_{ret} \) (i.e., every last state of the advice) is connected to 
all the corresponding states in the base state machine (\( s_1 \in S_M \) such 
that \( Lab_A(s)|_{AP_M} = Lab_M(s_1) \)).

Categories of Aspects

Aspects can be divided into categories with respect to their possible influence 
on the computations of the base systems into which they can be woven. 
In [24], four categories of aspects are defined: spectative, regulative, weakly 
invasive and strongly invasive.

Spectative aspects merely gather information about the base system, 
regulative aspects can also rule out some paths in the base system, weakly 
invasive aspects can, in addition to the above, return from the advice exec-
tution to any reachable state of the base system, thus creating new paths. 
Strongly invasive aspects are allowed to return to any state in the base sys-
tem, even if it was not reachable before. Each category is contained in 
the following one, and the strongly invasive category includes them all. We want 
to categorize an aspect by the lowest category it is in, thus many aspects 
can be called weakly invasive (although of course they also are included in 
the strongly invasive category).

The following definition is most important for us:

Definition 4 An aspect \( A \) is strongly invasive relative to a model \( M \) if a 
state of \( M \) that was unreachable in \( M \) becomes reachable in the woven system 
\( M+A \) and transitions of \( M \) are applied to it.
The last part of the definition is needed to ensure that the aspect advice (sometimes) finishes in a state of M that was previously unreachable, and then the code of M is applied to the new state.

The definition of aspect categories is of a semantic nature, but often syntactic techniques are enough to determine the aspect category. As described in [24, 38, 43, 2], dataflow techniques can be used for this purpose. After the category of an aspect is identified, results from [24] can be applied to derive temporal properties of the base system that are guaranteed to be preserved by the aspect, according to its category, thus significantly simplifying the task of aspect correctness verification.

LTL Tableau

Intuitively, the tableau $T_f$ 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.

**Theorem 1** (from [7], 6.7, Theorems 4 & 5) Given a tableau $T_f$ for formula $f$, for any Kripke structure $M$, for all fair paths $\pi'$ in $M$, if $\pi' \models_M f$ then there exists a fair path $\pi$ in $T_f$ such that $\pi$ starts in $S_{0}^{T}$ and $label(\pi')|_{AP} = label(\pi)$.

2.3 Specification and Verification for Weakly Invasive Aspects

2.3.1 Weakly Invasive Aspect Specification

A specification of a weakly invasive aspect A is a pair of LTL formulas, $(P_A, R_A)$. $P_A$ is the assumption of A about (the reachable part of) any base system into which A may reasonably be woven. $R_A$ is the guarantee of A about the result of weaving A into any appropriate base system.

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. Such a specification captures the intension of the aspect.

**Definition 5** 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.

### 2.3.2 Modular Verification of Weakly Invasive Aspects

Given a weakly invasive aspect $A$ with its assume-guarantee specification $(P_A, R_A)$, the MAVEN tool presented in [16] can be used for modular verification of $A$'s correctness. Modular verification in this context means that the aspect is checked independently of any concrete base system. After an aspect is shown correct, it can be reused without additional proofs in every base system satisfying its assumption.

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. The input format of MAVEN is an extension of the NuSMV finite state machine language, and is named FSMA, for “finite state machine aspects.” It 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. 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.

**INIT** Initial states of the aspect machine are defined by the conjunction of all the INIT directives. These directives are 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.

**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.

The verification algorithm builds a tableau from base assumption $P_A$ using the ltl2smv component of NuSMV and weaves $A$ with this tableau according to pointcut descriptor $ptc_A$. Then NuSMV is applied to model-check the augmented tableau to verify the desired result $R_A$. 

15
The soundness of the verification process in MAVEN follows from the following key property: the tableau $T_{P_A}$ of $P_A$ contains all the possible computations satisfying $P_A$, and thus all the possible computations of all the base systems into which $A$ can be woven (i.e., those base systems satisfying $P_A$). It is then shown that all the computations of any woven system with $A$ are represented in the woven tableau $T_{P_A + A}$. Thus if the guarantee of $A$ holds in $T_{P_A + A}$, then it also holds in every woven system with $A$.

There are several restrictions to this method that are removed in the current work:

- The aspects treated are weakly invasive
- The pointcuts are assumed to be state predicates only, and not full past-LTL formulas
- The case when an aspect might return to the same base state from which it started its execution is not treated
- Aspects cannot have local memory

The MAVEN tool is available as part of the Common Aspects Proof Environment (CAPE) [25] 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. An example of MAVEN specification can be found in Chapter 6, where our improvements to MAVEN are described.
Chapter 3

Modular Verification of Strongly Invasive Aspects

3.1 Verification theory for strongly invasive aspects

3.1.1 Refined Aspect Specification

The assumption of a strongly invasive aspect has to contain more information than the assumption of a weakly invasive one: it sometimes needs to define restrictions on the behavior of the unreachable part of the base system into which the aspect can be woven, in order to ensure an appropriate behavior of the woven system from the states that are made reachable by the strongly invasive aspect.

The specification of aspect $A$ is still a pair of assumption and guarantee, $(P_A, R_A)$, where $P_A$ is the assumption about the base system and $R_A$ is the result assertion guaranteed to hold in the woven base with the aspect. But now the assumption $P_A$ is a pair of LTL formulas, $(PR_A, PU_A)$, in which $PR_A$ is the assumption about the reachable part of the base system (the kind of assumption that was possible in [16]), and $PU_A$ is a new kind of assumption. This statement is an LTL formula defining restrictions on the unreachable part of the base system which is made reachable by completing an aspect advice fragment. The restriction is posed on computations of the base system that start in the states that might be reached by completing the
aspect advice, which were previously unreachable. We now may define the correctness of an aspect relative to such a specification, relating to a base system $S = S_{reach} \cup S_{unreach}$ where $S_{reach}$ represents the reachable part, and $S_{unreach}$ consists of all the paths starting from states in $S \setminus S_{reach}$. Note that the paths in $S_{unreach}$ might contain reachable states of $S$, thus the union above is not necessarily disjoint.

**Definition 6** An aspect $A$ is correct with respect to its refined assume-guarantee specification $((P_{RA}, P_{UA}), R_A)$ if, whenever it is woven (by itself) into a system $S = S_{reach} \cup S_{unreach}$, where $S_{reach}$ satisfies $P_{RA}$ and the paths from the states of $S_{unreach} \setminus S_{reach}$ that might become reachable after weaving satisfy $P_{UA}$, the result will satisfy the guarantee, $R_A$.

The property of $S_{unreach}$ is relevant only for computation segments starting from a state that can be the last state of an advice execution. The reason is that only by an advice execution can a computation of the woven system pass from a state that was reachable in the base system to a state which was unreachable in the base system. Thus in order to check that the computations of $S_{unreach}$ satisfy the requirements of the aspect it is enough to verify a formula of the form $L_A \rightarrow P_{UA}$ on them, where $L_A$ is a state formula describing the set of all the possible last states of the advice state machine, projected on the base system variables.

With some abuse of notation, we denote by $L_A$ the set of possible last states of aspect $A$ (identifying the unary predicate with the set it describes). Note that this set consists exactly of all the states in the base system into which a computation can arrive after finishing advice execution.

**Definition 7** Given an aspect $A$ with refined assume-guarantee specification $((P_{RA}, P_{UA}), R_A)$ where $P_A = (P_{RA}, P_{UA})$, we say that a system $S$ satisfies the assumption of $A$, denoted by $S \models P_A$, iff $((S_{reach} \models P_{RA}) \land (S_{unreach} \models L_A \rightarrow P_{UA}))$

Now Definition 6 can be seen as a generalization of the basic correctness of an aspect in Definition 5. This definition is illustrated in Figure 3.1.

### 3.1.2 Refined Tableau Construction

Given an aspect $A$ and its refined specification, $((P_{RA}, P_{UA}), R_A)$, we need to construct a refined tableau to serve as a representation of all the base
systems into which our aspect will possibly be woven. But now in order to build the tableau of the assumption of the aspect, it is not enough to build the tableau of $PR_A$: we need to restrict the unreachable part of the tableau. The tableau needs to represent the systems, the reachable part of which satisfies $PR_A$, and the unreachable part of which satisfies $L_A \rightarrow PU_A$, where $L_A$ is the predicate defining the set of all the possible return states of the advice. The refined tableau, $T$, is constructed in three steps:

**Step 1:** Automatically construct the predicate $L_A$. The construction is shown in Section 3.2.1.

**Step 2:** Use the ltl2smv module of the NuSMV model checker to build the tableau $T_1$ of the LTL formula $(PR_A \lor (L_A \land PU_A))$.

**Step 3:** Take the tableau $T$ to be the same as $T_1$ except for the initial states definition. To obtain the $INIT_T$ predicate of $T$, restrict the $INIT_{T_1}$ predicate of $T_1$ to include only states that should be reachable in the base system: $INIT_T = INIT_{T_1} \land PR_A$.

Note that $T_1$ is the tableau of $(PR_A \lor (L_A \land PU_A))$ and not of $(PR_A \lor
(L_A \rightarrow PU_A)), because the only way to reach the part of the base system that does not satisfy PR_A is by application of an aspect advice, and this will bring the computation to a state in which L_A must hold. This intuition will be justified during the proof of Theorem 2.

Let us denote the refined tableau constructed as above by $T_{(PR_A, (PU_A, L_A))}$.

**Theorem 2** Let A be an aspect with the refined assume-guarantee specification $((PR_A, PU_A), R_A)$, and let $L_A$ be a formula describing the set of all the possible last states of A. Then A is correct with respect to $((PR_A, PU_A), R_A)$ if the result of weaving A into $T_{(PR_A, (PU_A, L_A))}$ satisfies $R_A$.

We delay the proof of the theorem until after bringing some helpful definitions and lemmas needed for the proof. They 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 $T_{(PR_A, (PU_A, L_A))}$ satisfies $R_A$, then for every base system M such that its reachable part satisfies $PR_A$ and the unreachable part satisfies $L_A \rightarrow PU_A$, the result of weaving A into M satisfies $R_A$. For this purpose it is enough to show that for every infinite fair path $\sigma$ in the woven system $M + A$ there exists a corresponding infinite fair path $\pi$ in the woven tableau, $T_{(PR_A, (PU_A, L_A))} + A$, such that $\text{label}(\sigma)|_{AP_A} = \text{label}(\pi)|_{AP_A}$ (recall that $AP_A$ is the set of all the atomic propositions appearing in the specification and advice of A, containing all the necessary information about the state of the base system). 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 $T_{(PR_A, (PU_A, L_A))}$ 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 $\sigma$, 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 $s_0, \ldots, s_k$. 20
Proof.
A computation is fair if it visits states from the Fairness set of the system model infinitely often. Let $\pi_0$ and $\pi_k$ 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 $\sigma_0$ and $\sigma_k$. Let us take $\pi = \sigma_0 \cdot s_0 \cdot s_1 \cdot \ldots \cdot s_k \cdot \sigma_k$. This is obviously a path in $S$, and it starts from an initial state, as did $\sigma_0$. Moreover, $\pi$ is a fair computation, because it has the same infinite suffix, $\sigma_k$, as the fair computation $\pi_k$.
Q. E. D. (Lemma 1)

Definition 8 Any infinite path $\pi$ 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 $\pi^i$ is a sequence of states such that:

- If $i = 0$, the first state of $\pi^i$ is the initial state of $\pi$

- If $i > 0$, the first state of $\pi^i$ 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)

- The last state of $\pi^i$ is either a pointcut state or a last state of an advice (after which the computation returns to the base system), or the last state of the path, if $\pi$ is finite

- There are no pointcut states and no last states of advice inside $\pi^i$ (i.e., in the states of $\pi$ that are not the first or the last state)

- $\pi$ is the concatenation of the path segments of $\pi$ in the order of their indices

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 - in the reachable part of the base, or in the unreachable part, or even 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. A resumption state might be unreachable in the system before weaving - in case of a strongly invasive aspect.

21
Now if we are given a path of M+A, \( \sigma = \sigma^0 \cdot \sigma^1 \cdot \ldots \) where \( \sigma^i \)-s are the path segments of \( \sigma \), for each finite prefix of \( \sigma \) consisting of a number of path segments we define the set of corresponding path-segment prefixes of fair paths in T+A:

\[
\Pi_i = \{ \pi^0 \cdot \pi^1 \cdot \ldots \cdot \pi^i | \\
\quad \text{label}(\pi^0 \cdot \ldots \cdot \pi^i) \mid_{APA} = \text{label}(\sigma^0 \cdot \ldots \cdot \sigma^i) \mid_{APA}, \\
\quad \exists \pi \text{ fair path in } T+A \text{ such that } \pi = \pi^0 \cdot \ldots \cdot \pi^i, \ldots \}
\]

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

**Lemma 2** Given a fair computation \( \sigma \) of M+A, and sets of prefixes \( \Pi_i \)-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 \( \Pi_0 \) is not empty we need to show the existence of \( \pi^0 \) such that \( \text{label}(\pi^0) \mid_{APA} = \text{label}(\sigma^0) \mid_{APA} \) and \( \pi^0 \) is a prefix of some fair path \( \pi \) in T+A. \( \sigma^0 \) is the first path-segment of a fair path in M+A, thus there is no advice application before \( \sigma^0 \) or inside it. So \( \sigma^0 \) is also the first path segment of a fair computation in \( M \). According to the assumption on \( M, M \models PR_A \), 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 \( \text{label}(\sigma^0) \mid_{APA} = \text{label}(t_0, \ldots, t_k) \mid_{APA} \).

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 \( \pi^0 \) is indeed a path-segment, and then it will follow that \( \pi^0 \in \Pi_0 \), meaning that \( \Pi_0 \) is not empty.

\( \text{label}(t_0) = \text{label}(\sigma^0)_0 \), thus \( t_0 \) is an initial state of \( T+A \). There is no pointcut inside \( \sigma^0 \), because it is a path-segment, so the last state of \( \sigma^0 \) 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 \( \pi^0 \)
will also be marked as a pointcut state. For the same reason, there are no pointcut states among \( t_0, \ldots, t_{k-1} \), which, in the same way as for \( \sigma^0 \), implies that there are no advice return states also. Thus both ends of \( \pi^0 \) are legal ends of a path-segment, and there are no pointcut states and no advice return states inside \( \pi^0 \), which makes it, indeed, a legal path-segment.

**Induction step** Let us assume that for every \( 0 \leq i < k \), \( \Pi_i \neq \emptyset \). We need to prove that \( \Pi_k \neq \emptyset \).

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

1. \( s_{\text{last}}(k-1) \) is a pointcut. Then \( t_{\text{last}}(k-1) \) is also a pointcut, because due to the induction hypothesis \( \text{label}(s_{\text{last}}(k-1))|_{AP_A} = \text{label}(t_{\text{last}}(k-1))|_{AP_A} \).
   
   Then in every continuation of the computation both in \( M + A \) and in \( T + A \) the advice of the aspect will be performed, thus the \( k \)-th path-segment will in both cases be the application of the same advice from the same state, and the agreement on the labels of the \( k \)-th path-segments will be trivially achieved. Moreover, for the same reason the existence of an infinite fair path with the prefix \( \pi^0, \pi^1, \ldots, \pi^{k-1} \) implies the existence of an infinite fair path with the prefix \( \pi^0, \pi^1, \ldots, \pi^k \), 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. \( s_{\text{last}}(k-1) \) is a last state of the advice. This, in particular, implies that \( s_{\text{last}}(k) \) is a pointcut state, and no advice has been applied between \( s_{\text{last}}(k-1) \) and \( s_{\text{last}}(k) \). Here are again two possibilities:

   The first case is that \( s_{\text{last}}(k-1) \) is a reachable state in \( M \) (more precisely, the state reachable in \( M \) is the projection of \( s_{\text{last}}(k-1) \) on \( AP_A \)). As no advice is applied between \( s_{\text{last}}(k-1) \) and \( s_{\text{last}}(k) \), we have that the whole path-segment \( \sigma^k \) is in the reachable part of \( M \). Moreover, due to Lemma 1, as both \( s_{\text{last}}(k-1) \) and \( s_{\text{last}}(k) \) 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 $s_{last(k-1)}, s_{first}(k), \ldots, s_{last}(k)$. All the fair computations of the reachable part of M are represented in the tableau of $PRA$, which is exactly the reachable part of T. Thus, in particular, the above fair path has a corresponding path in T, and, as there was no pointcut or advice application inside the sequence $s_{last(k-1)}, s_{first}(k), \ldots, s_{last}(k)$, there are also no pointcuts and advice applications in the corresponding sequence in the computation of T, and thus there exists a corresponding sequence of states in $T+A$, $\pi^k$. The first state of $\pi^k$, $t_{last(k-1)}$, is reachable from the initial state of $T+A$ by some fair path, as $\Pi_{k-1}$ is not empty. Moreover, all the prefixes of such fair paths appear in $\Pi_{k-1}$, thus at least one of them continues to the sequence $\pi^k$. So indeed we obtain that there exists a sequence of states $\pi^k$ corresponding to $\sigma^k$ in the woven tableau, for which a fair continuation exists. We are left to see that the sequence of states, $\pi^k$, is indeed a path segment in the woven tableau computation. But this is true due to the agreement on labels of the states, $\text{label}(\pi^k) |_{PA} = \text{label}(\sigma^k) |_{PA}$: the path segment $\sigma^k$ 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 $\pi^k$ and thus $\pi^k$ is a path segment.

The last case left is that $s_{last(k-1)}$ is unreachable in M. Additionally, $s_{last(k-1)}$ is the last state of the advice, thus $s_{first}(k)$ is the return state of the advice, and also is unreachable in M, because according to the weaving algorithm

$\text{label}(s_{last(k-1)}) |_{PA} = \text{label}(s_{first}(k)) |_{PA}$. From the fact that $s_{first}(k)$ is unreachable in M, together with the assumption on the unreachable part of M, we have that $L_A \rightarrow PU_A$ holds in the suffix of any path starting from $s_{first}(k)$. But from the agreement on labels with $s_{last(k-1)}$ we also have that $s_{first}(k) \models L_A$. Together we obtain that $PU_A$ holds in the suffix of any computation in M starting from $s_{first}(k)$, and, in particular, for the computation $\sigma t$ containing the next path segment of $\sigma$, $\sigma^k$ (because there is no advice application inside $\sigma^k$, all its states are states of the original system, M - either in the reachable or the unreachable part). Now let us examine the states of the woven tableau. The tableau of $L_A \land PU_A$ is included in the refined tableau T, thus every computation satisfying $PU_A$ that
starts from a state satisfying $L_A$ is represented in $T$ (though its initial state might be unreachable before the aspect is woven into $T$). Let $\pi t$ be a computation that corresponds to the suffix of $\sigma t$ that starts from $s_{\text{first}}(k)$. The first state of $\pi t$ agrees on its label with $s_{\text{first}}(k)$, and thus with $s_{\text{last}}(k-1)$, which, according to the induction hypothesis, implies agreement on labels with $t_{\text{last}}(k-1)$. According to the weaving algorithm, the last state of the advice is connected to all the states in the underlying system with which it agrees on labels. Thus, in particular, $t_{\text{last}}(k-1)$ (which is the last state of the advice, in the same way as $s_{\text{last}}(k-1)$), is connected to the first state of $\pi t$. So we can take the first state of $\pi t$ to be the first state of $\pi^k$. Let us then take $\pi^k$ to be the first path-segment of $\pi t$. It is indeed a path segment of a fair computation (due to Lemma 1), it is connected to $\pi^{k-1}$ and agrees on labels with $\sigma^k$, so we found what we needed.

Thus, indeed, the set of possible continuations, $\Pi_i$, is never empty. Q. E. D.(Lemma 2)

**Theorem 2 proof:** Now let us return to the proof of Theorem 2. Let us be given an infinite fair path $\sigma$ in the woven system $M + A$. From Lemma 2 it follows that there exists an infinite path $\pi$ in the woven tableau corresponding to the given path $\sigma$ - all the prefixes of $\pi$ appear in the $\Pi_i$-s above, and due to the lemma, the $\Pi_i$-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 $\Pi_i$-s above is fair, for the following reason: There are two possibilities for the infinite suffix of $\pi$. 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 $\pi$, 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 $\pi$ is indeed fair. This completes the proof of Theorem 2

Q. E. D.
3.2 Algorithms

3.2.1 Computing $L_A$ Automatically

Given a model of the aspect, $A$, in MAVEN format, we would like to automatically compute the state formula defining the set of all the possible last states of A’s advice. The algorithm we propose consists of four steps:

**Step 1:** Construct a formula $\varphi$ defining the pointcut of the aspect: take $\varphi$ to be the disjunction of all the POINTCUT expressions in $A$.

**Step 2:** Construct a system in which all the possible computations of the aspect are represented, by starting from all the possible join-point states, and weaving the advice into them all. This construction is performed by running MAVEN on a model $A'$ which is the same as $A$ except for a change in the specification. The assumption of the aspect is replaced by $\varphi$, and the guarantee of the aspect is replaced by $true$. Our goal then is achieved in the following way:

- At the first step of its work, MAVEN will automatically construct the tableau of the new assumption of the aspect, $\varphi$, using the ltl2smv module of NuSMV. Note that in this tableau, $T_\varphi$, only the initial states are restricted, and the initial states are exactly all the possible join-points of the aspect.

- At the second step, MAVEN will perform the weaving of the aspect into the constructed tableau. The obtained woven system, $T_\varphi + A$, will contain all the possible computations of the aspect, because the initial states of the tableau are all the possible pointcut states that can occur in either reachable or unreachable parts of the base systems into which A will be woven (as the ranges of all the base variables as defined in the aspect model definition are the maximal possible, and the combinations of variables values are restricted only by the formula $\varphi$).

Note that if we added other restrictions on the computations of the tableau $T_\varphi$, we may not be able to guarantee that all the possible runs of the advice of A will appear in the woven tableau. For example, if we demand that the computations of the tableau should satisfy $PR_A$, then after the weaving we would not obtain the runs of the aspect from the states that were unreachable.
in the base system. Since in the unreachable part of the base system which becomes reachable after the weaving there might be join-points of A, we have to model the computations of the advice starting from these states. However, there are cases when additional restrictions might be posed on the computations of the tableau built. For example, there might be some invariant that holds both in the reachable and the unreachable parts of the base system, and then it could be added to \( \varphi \). Additionally, there might exist an assertion that holds for all the pointcut states, but is not explicitly written as part of the pointcut. Then it would be possible to restrict the initial states of the constructed tableau by this assertion.

**Step 3**: Take the woven system obtained in Step 2, \( T_{\varphi} + A \), and use the built in functionality of NuSMV to compute the set of all the reachable states of this model, \( (T_{\varphi} + A)_{\text{reachable}} \). For each of the states in \( (T_{\varphi} + A)_{\text{reachable}} \), check whether it satisfies any of the RETURN conditions of the aspect. If it does, add it to the set \( L_A \).

**Step 4**: Now \( L_A \) is the set of all the possible last states of A. What is left is only to construct the predicate describing this set. This is done by taking the disjunction of all the predicates describing the states in \( L_A \).

Sometimes it might be easy to see a compact description of the possible last states of the aspect. For this case we provide the user a possibility to supply a manually constructed predicate \( L \). But such a predicate should be checked before use, because the intuition of the user might be wrong. Then we use the above algorithm to construct the full \( L_A \) predicate, and check that the supplied predicate \( L \) is implied by \( L_A \). If indeed \( L_A \rightarrow L \) holds, the verification using \( L \) will still be sound, because it just might check additional paths, but no relevant path will be left unverified.

### 3.2.2 Determining the Aspect Category

Before applying the full verification technique it is very desirable to determine the category of the aspect. If the aspect is of the weakly invasive category (or a simpler category included within the weakly invasive one), then the simpler method described in [16] is applicable to it. Otherwise, the full verification method described in Section 3.2.3 should be used.

Some ways of determining the category of the aspect using code analysis, dataflow techniques and semantic definitions are described in [24, 38, 43, 9,
2]. If none of them gives a positive answer, the algorithm presented below can help to determine whether the aspect is uniformly strongly invasive, i.e., is always strongly invasive for every possible base system to which it can be woven. But first some definitions and observations are needed:

Observe that from Definition 4 in Section 2.2.2 it immediately follows that for any system $M$ in which all the states not reachable from the initial state by some fair path have been removed, if an aspect $A$ is strongly invasive relative to $M$, there is a deadlock in the system $M + A$: Let $s$ be a last state of advice execution such that there exists no reachable state $s'$ in $M$ for which $\text{label}(s') = \text{label}(s) |_{AP_A}$. Then this state is a deadlock state in the woven system.

**Lemma 3** Let aspect $A$ have the specification $((PR_A, PU_A), RA)$, where $AP_A$ is the set of all the atomic propositions appearing in the specification and $TP$ denotes the tableau of $PR_A$. Aspect $A$ is strongly invasive with respect to $PR_A$ if when $A$ is woven into $TP$, there exists a state $s$ in $TP + A$ such that:

- $s$ is the last state of advice execution, and
- there exists no state $s'$ in $TP$ such that $s'$ is reachable by some fair computation of $TP$ and $\text{label}(s') = \text{label}(s) |_{AP_A}$

**Proof.**
Immediate from the observation above.

**Definition 9** Given a tableau $T$ of an LTL formula $\phi$, the tableau $TP$ obtained from $T$ by removing all the states that are not reachable from the initial state of $T$ by any fair path (and only them) is called the pruned tableau of $\phi$.

Note that the above defined pruned tableau is equivalent, in terms of the fair computations set, to a tableau obtained from $T$ by removing all the states and transitions that only lead to deadlock states.

**Lemma 4** Aspect $A$ with the specification $(PR_A, RA)$ is strongly invasive relative to $PR_A$ iff there exists a deadlock in the system $TP_{PR_A} + A$, where $TP_{PR_A}$ is the pruned tableau of $PR_A$. 

28
Proof.
The conditions of the observation above Lemma 3 hold, in particular, for $M = T_{PR_A}$, so there will be a deadlock state in $T_{PR_A} + A$.

On the other hand, if there exists a deadlock in the system $T_{PR_A} + A$, let $s$ be the deadlock state. Let us denote by $s'$ the state of $T_{PR_A}$ such that $\text{label}(s') = \text{label}(s) \upharpoonright_{AP_A}$. There are two possibilities: If $s'$ is reachable in $T_{PR_A}$, then there exists some infinite computation $\pi = s', s_2, \ldots$ from $s'$ in $T_{PR_A}$, because $T_{PR_A}$ is a pruned tableau. In particular, there exists a state $s_2$ in $T_{PR_A}$ (the second state of $\pi$) to which $s'$ is connected. However, in $T_{PR_A} + A$ the state $s$ is no longer connected to $s_2$. According to the construction of $T_{PR_A} + A$, the only reason could be that an advice is applied at $s$. But if an advice was applied at $s$, $s$ would not be a deadlock state. Thus when we assumed that the projection of $s$ on $AP_A$ is reachable in $T_{PR_A}$ we obtained a contradiction. So we conclude that $s'$ is unreachable in $T_{PR_A}$.

But could $s'$ still be reachable in $T_P$? This can only be if $s'$ has been removed from $T_P$ during the construction of the pruned tableau. This means that all the paths starting from $s'$ led to some deadlock states, and thus $s'$ couldn’t be reached by any fair computation of $T$. But according to Lemma 3 this exactly means that the aspect $A$ is strongly invasive relative to its assumption.

Q. E. D.

According to Lemma 4, in order to check whether the given aspect is strongly invasive relative to its assumption, the following algorithm is possible: First, detect and rule out all the deadlock states, and all states such that all the paths from them lead only to deadlock states in the tableau of the assumption. In such a way a deadlock-free tableau is obtained which contains exactly the same fair paths as the original one. As a second step, weave $A$ into the deadlock-free tableau, and, at last, check wether there are any deadlocks in the resulting woven system. This algorithm can be automatically executed using MAVEN and NuSMV in the following way:

1. Construct the pruned tableau $T_{PR_A}$ from the tableau of the assumption of $A$. This is done automatically, by an iterative procedure that we have added to MAVEN. The procedure is as follows:
   
   • Run NuSMV to detect deadlock states in the tableau.
   • If a deadlock state is detected, construct a predicate describing
this state, $p$

- Rule out the deadlock state: Add the negation of $p$ to the initial state definition, and to the predicate defining possible next states of the transitions.

Repeat the procedure until there are no more deadlocks in the tableau.

2. Use MAVEN to weave the aspect into the above constructed tableau.

3. Run NuSMV to check whether there are deadlocks in the woven tableau. If a deadlock is detected, the aspect is strongly invasive relative to its assumption. Otherwise, the aspect $A$ is weakly invasive relative to $PR_A$.

Note that the algorithm presented here gives a positive answer only if the aspect is strongly invasive relative to the tableau of its assumption, but not relative to a concrete base system. Thus if the algorithm gives a positive answer, the aspect is strongly invasive relative to all the possible base systems into which it might be woven. But if the algorithm gives a negative answer, there might exist a base system satisfying the assumption of the aspect, with respect to which our aspect is still strongly invasive.

Given a base system $S$, there is one more way for us to check whether the given aspect, $A$, is strongly invasive relative to this system. Intuitively, what we would like to do is to look at all the unreachable states of the base system, and check whether there are last states of our aspect among these unreachable states. For that purpose we can check satisfiability of the following formula: $\varphi = S_U \land L_A$, where $S_U$ is the formula defining the set of all the unreachable states of $S$, and $L_A$ is the formula defining the set of all the possible last states of $A$. $\varphi$ can be constructed automatically: the way to construct $L_A$ automatically is shown in Section 3.2.1, and the way to construct $S_U$ automatically is shown in Section 3.2.4. Then the satisfiability of $\varphi$ can be automatically checked using a SAT solver (such as, for example, Chaff [33]). If $\varphi$ is found unsatisfiable, it means that there are no last states of the aspect $A$ in the unreachable part of $S$, so $A$ is weakly invasive relative to $S$, and the simpler model check in [16] can be used. If $\varphi$ is found satisfiable, it doesn’t necessarily imply that $A$ is strongly invasive relative to $S$, because the predicate $L_A$ is an over-approximation: it contains all the possible last states of the aspect, but maybe some of them will never
occur in the computations of the woven system $S + A$, and thus will not bring the computation to states that were unreachable in $S$. But this over-approximation is a safe one: if we declare some aspect as strongly invasive when it is weakly invasive, we will just have to work harder to prove its correctness than we would if we knew its exact category, but the verification results will be sound.

3.2.3 Verifying the Aspect

Given an aspect $A$ and its refined assume-guarantee specification, $((PR_A, PU_A), R_A)$, the verification of correctness of $A$ with respect to $((PR_A, PU_A), R_A)$ is performed as follows:

1. Construct the refined assumption tableau for $A$ as shown in Section 3.1.2 - the $T(PR_A, (PU_A, L_A))$.

2. Use MAVEN to weave $A$ into $T(PR_A, (PU_A, L_A))$ and to run the NuSMV model checker on the resulting system and check the $R_A$ property on it.

3.2.4 Base System Correctness Verification

Non-optimized solution

Given a base system $S$, we need to verify that it satisfies the refined assumption of our aspect, $(PR_A, PU_A)$:

- Verify that the reachable part of $S$, $S_{reach}$, satisfies $PR_A$

- Verify that all the computations starting from the unreachable part of $S$, $S_{unreach}$, satisfy $L_A \rightarrow PU_A$.

The first verification task can be done by usual model-checking of $S$ versus $PR_A$. The meaning of the second task is as follows: we need to examine the model of $S_{unreach}$ and check all the fair computations that start from states satisfying $L_A$ (note that a computation starting from a state in $S_{unreach}$ might return to the reachable part of $S$ at some state). All these computations should satisfy $PU_A$. The verification is performed in three steps:
1. Automatically compute the state formula $S_U$ defining the set of all the unreachable states of $S$: $S_U$ is the negation of the formula $S_R$ defining all the reachable states of $S$, and in NuSMV there exists a possibility to compute $S_R$ automatically for a given system $S$.

2. In the model of the base system, $S$, automatically replace the initial states definition by the formula $S_U \land L_A$.

3. Run NuSMV on the obtained model and the formula $PU_A$. If the verification succeeds, it means that the given base system satisfies the restriction on the unreachable part.

**Optimization**

In some cases, the requirement in the second part of the verification process can be relaxed due to the structure of $PU_A$. For example, in case $PU_A$ is some safety property, i.e., $PU_A$ has the form $G \varphi$, we do not have to verify that $\varphi$ holds all along the computations starting from resumption states in the unreachable part of the system. We need to check only the segments between a resumption state and the next join-point (if exists). So if we denote by $ptc(A)$ the predicate defining the pointcut of the aspect, then it is enough to verify the following formula on the unreachable part of the system: $L_A \rightarrow (\varphi W (ptc(A) \land \varphi))$. The reason is that when the computation reaches a join-point, in the woven system the advice will be executed at that point, so the information about the possible continuations of the computation in the base system from that point is useless. And after the advice returns, if the resumption state is still unreachable, it is one of the initial states of $S_{unreach}$, thus all the computations starting from it are verified. Note that in the special case when $PU_A$ is a state invariant that appeared also in $PR_A$, the verification process can be relaxed even further: it is then enough to check the segments between a resumption state and either the next join-point or the next reachable state (the one that comes first). The reason is that after a computation arrives at a reachable state, it either stays forever in the reachable part of the system (and thus all its states satisfy $\varphi$ due to $PR_A$), or stays in the reachable part till advice execution takes it to an unreachable resumption state (and from there on the $\varphi$ invariant has already been checked). If we denote by $reachable$ the predicate defining the reachable
states of the base system, the formula we need to verify on the unreachable part of the system now becomes $L_A \rightarrow (\varphi \cup (reachable \lor (ptc(A) \land \varphi)))$.

As an example of the situation described above, we can consider an aspect that is in charge of the scheduling policy of a semaphore-guarded resource. The purpose of the aspect is to implement a possibility of a waiting queue for the semaphore. As a result, the semaphore that could previously have only values 0 or 1 can now have negative values (according to the number of waiting processes). Thus the aspect is indeed strongly invasive. But there is a part of the system invariant that we need to extend to the unreachable part of the base system: regardless of the semaphore value and the concrete scheduling algorithm, we demand that no two processes hold the guarded resource at the same time. So if the formula $\psi$ encodes the fact that two processes hold the resource at the same time, the assumption of the aspect about the unreachable part of the base system should be $U = G \neg \psi$. But when verifying the computations starting in the unreachable part of the base system, it is enough to check that after each possible last state of the aspect the computation satisfies $\neg \psi$ until it arrives to a pointcut state or to a reachable state.

3.3 Example

In this example we discuss an aspect that can be used in any grades-managing system. The aspect $GR$ provides a way of giving bonus points for assignments and/or exams (thus making it possible to have assignment/exam grades that are more than 100), but still keeping the final grade within the $0..100$ range.

The aspect has two kinds of pointcuts, and two corresponding pieces of advice. The first pointcut of $GR$ is the moment when an assignment or exam grade is entered to the system. At this point the original system would accept only grades between 0 and 100, but the aspect offers a possibility of giving a bonus on the grade, and stores the new grade successfully even if it exceeds 100. The second pointcut of $GR$ is the moment when the final grade calculation of the base system is performed. Then if the calculation resulted in a grade that exceeds 100, the aspect replaces this grade by 100 (otherwise keeping the grade unchanged).

Aspect $GR$ is strongly invasive in the systems into which it can reason-
ably be woven, because its operation results in states in which some grades are more than 100, which is impossible in the base systems without bonus policies. And this example, though simple, is still of interest to us, because the aspect here exhibits a typical behavior we would like to treat: when it is woven into a system, the calculations there are performed partly in the aspect, and partly in the base system code, but using new inputs, that were impossible before the aspect was woven in.

The specification of $GR$ can be formalized as follows:

- The assumption on the reachable part of the base system is that all the grades appearing in the grading system - homework assignment grades ($hw_i$), exam grades ($exam_j$), final grade ($f$) - are between 0 and 100, and after the final grade is ready ($f_{ready}$) (i.e., all the assignments and exams that comprise the grade have been checked, and the final grade has been calculated from them according to the base system grading policy), the final grade is published ($f_{published}$). The result of the final grade calculation of the base system is represented by $calc$.

$$PR_{GR} = [G(f_{ready} \rightarrow ((f = calc) \land F f_{published}))
\land G(f_{published} \rightarrow f = calc) \land
G(0 \leq f \leq 100) \land
G(\forall 1 \leq i \leq 10(0 \leq hw_i \leq 100)) \land
G(\forall 1 \leq j \leq 2(0 \leq exam_j \leq 100))]$$

Here, for modeling purposes, we have to provide some bounds on the number of assignments and exams, so we assume that there are no more than 10 home assignments and no more than 2 exams in each course. We also show the specification for the grades of a single student (because the grades of different students are independent, and calculations involving them can be viewed as orthogonal). When the model of the aspect is built, the ranges of all the variables - both the aspect variables and the relevant base system ones - are defined. Let us assume, for example, that our aspect gives bonuses in range of 0..20 points, then all the grade variables defined in the model of GR are in the range 0..120.

- The assumption on the unreachable part of the base system that be-
comes reachable after adding $GR$ is in our case a weakening of $P_{GR}$.
We still want the final grades to be published after they are ready, but
now the final and the intermediate grades do not have to be bound by
100, but by 120. So we are left with the following property:

$$PU_{GR} = \left[ G(f\_ready \rightarrow ((f = calc) \land F f\_published)) \land 
G(f\_published \rightarrow f = calc) \land 
G(0 \leq f \leq 120) \land 
G(\forall 1 \leq i \leq 10(0 \leq hw\_i \leq 120)) \land 
G(\forall 1 \leq j \leq 2(0 \leq exam\_j \leq 120)) \right]$$

- The guarantee of the aspect now is that regardless of the existence of
bonuses on the components of the final grade, the final grade will be
the one calculated by the base system function, but rounded down to
100 if needed:

$$R_{GR} = \left[ G(f\_published \rightarrow f = \min(calc, 100)) \right]$$

The guarantee of the aspect might also include a statement about the
bonus policy it enforces, saying that the aspect calculates the bonuses
as desired. But to simplify the discussion, we omit it here.

- The pointcut of the aspect can be formalized using the following predi-
cates, which define the moments when the grades are entered into the
system: $enter\_hw\_i$ for homework grades, and $enter\_exam\_j$ for exam
grades.

$$Pointcut_{GR} = \left[ (\bigvee_{i=1}^{10}(enter\_hw\_i)) \lor 
(enter\_exam\_1) \lor (enter\_exam\_2) \lor 
(f\_ready \land (f > 100)) \right]$$

Let us follow the verification algorithm, applying it to aspect $GR$. The
first step is the refined tableau construction. It begins with calculating the
predicate $L_{GR}$, defining all the possible last states of $GR$. In our example,
we get

\[
L_{GR} = [(f_{ready} \rightarrow (f = 100) \land (calc > 100)) \land
\neg f_{published}] \land
\forall 1 \leq i \leq 10(\neg enter_{hw.i}) \land
\forall 1 \leq j \leq 2(\neg enter_{exam.j}) \land
(0 \leq f \leq 120) \land (0 \leq calc \leq 120) \land
\forall 1 \leq i \leq 10(0 \leq hw.i \leq 120) \land
\forall 1 \leq j \leq 2(0 \leq exam.j \leq 120)]
\]

And here is the explanation: All the combinations of exams and assignments grades values in range 0..120 are possible at the last state of the aspect, because all the grades of assignments and exams are independent. There is a connection between the final grade and the other grades, but only when the final grade is declared to be ready and still is not published. Then the final grade is equal to the minimum between the calculated value (\textit{calc}) and 100. In the other states of the computation the value of the final grade is not restricted (except by its range), so in those states effectively we have to enable any combination of the final grade value and the other grades. The values of the other system variables are restricted as follows: The variables \textit{enter_{hw.i}} and \textit{enter_{exam.j}} for all \(i\)-s and \(j\)-s are \textit{false}, because no grade is entered by the user at the last state of the advice. The variable \textit{f_{published}} is also \textit{false}, because the aspect does not publish the grades - even if it was called at the moment when the final grade was calculated, it just modifies the calculated grade, but does not publish it. Publishing the grades is done by the base system. The next variable to discuss is \textit{f_{ready}}. If the aspect was called at the moment of grades entering, the variable \textit{f_{ready}} is \textit{false} at the join-point. The final grade is not calculated by the aspect in this case, so the variable remains \textit{false} at the last state of the advice. However, if the aspect was called at a join-point when the final grade is calculated, the variable \textit{f_{ready}} is true there and remains true after the advice finishes its execution. In this case, as we said earlier, we will also have \(f = 100\) and \textit{calc} > 100.

Now after the predicate \(L_{GR}\) is constructed, the tableau of the \((PR_{GR} \lor (L_{GR} \land PU_{GR}))\) formula is created, its initial states are restricted to those
satisfying $PR_{GR}$ (that is, the refined tableau $T_{(PR_{GR},PU_{GR},L_{GR})}$ is built), and then GR is woven into the result. The last part of the verification process is running NuSMV on the woven tableau in order to check the $R_{GR}$ property on it. And for the above described aspect, with the specification given, the verification succeeds, so our algorithm shows that indeed it is correct with respect to its refined assume-guarantee specification. Intuitively, the reason for the success of the verification is that the base system performs only some arithmetic operations on the grades the aspect modifies, and thus we can expect that the result of performing old operations on the new arguments will be as anticipated, if only there is no overflow or type declaration problem. (By a type declaration problem we mean, for example, the case when the type of the grades variables is defined in the base code by some \texttt{typedef} to be 0..100, so that larger values cause a fatal type error.) But the assertion $PU_{GR}$ ensures that this will not happen, because $PU_{GR}$ will not hold for the base system in case such problems arise.

Note that the aspect does not restrict the grade calculation process of the base system, so this aspect is highly reusable, as long as the calculation can handle values greater than 100 (as seen in $PU_{GR}$). Moreover, this aspect can appear in a library of aspects providing different grading policies: different types of bonuses for homework assignments, or factors on the exam grades. All these aspects will have the same requirements from the base system as GR does, so when some grading system is checked for applicability of one of the aspects from this library, it is automatically inferred that all the other aspects from the library are also applicable to this base system. Thus the grading policy can be changed as needed at any time, by replacing the applied aspect, without any further checks on the base system.

3.4 Related Work

Several works have dealt with model checking of aspect systems [26, 40, 19, 16, 31, 20]. These works either treat a system with aspects woven in, or try to deal with the aspects modularly, relative to a specification. In the latter case, one or another form of an assume-guarantee specification is used.

So far, such a modular treatment was possible only for weakly invasive aspects. The reasoning behind the restriction is easy to understand: the aspect’s assumption about the base system only relates to those computa-
tion sequences and states (known as reachable states) that can occur for some fair execution of the base system without the aspect. When an aspect returns control to the base system code, but in a state of the base variables that does not occur for any computation of the base system that begins from a "normal" initial state, there is no restriction on the behavior of the continuation. Instructions from the base code are executed, but with values that were never expected or tested, and with no restriction on the outcome. Thus the overall behavior of such a system is hard to analyze in a modular manner, separating the reasoning about the base from the reasoning about the aspects to be woven. In such cases, modular reasoning was thought unfeasible.

On the one hand, this restriction still allowed treating most aspects, as many commonly used aspect examples are weakly invasive. Nevertheless, there are other aspects that definitely are strongly invasive, and that occur in real applications, e.g., the semaphore and the grading aspects described above, so that a more complete approach is desirable.

This chapter is based on our paper [21].
Chapter 4

Incremental Analysis of Interference Among Aspects

4.1 Semantic Interference Among Aspects

Given a library of reusable aspects, each of which is correct w.r.t. its assume-guarantee specification (in terms of Definition 6), it is important to check that the aspects will still function properly when woven all together into the same base system.

Definition 10 Given a set of correct aspects, \( A \), we say that \( A \) is interference-free if for any subset \( \{A_1, \ldots, A_n\} \subseteq A \) the following holds: Whenever \( A_1, \ldots, A_n \) are woven in any order into a base system that satisfies all the assumptions of the aspects, \( P_1, \ldots, P_n \) (in terms of Definition 7), the augmented system obtained after this weaving satisfies the guarantees of all the aspects in the subset \( R_1, \ldots, R_n \).

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.

When a library of aspects is not interference free, our analysis does not stop at getting this answer. Our goal in that case is to help the user to try to eliminate the interference, and to provide restrictions on aspect weaving if interference elimination is not possible. We then seek an answer for two questions: who is guilty in the interference, and what is to be done. The interference detection technique presented below enables an easy detection of the “guilty” aspect, and of the cause of the failure. In Section 4.2.4 we present an analysis of possible failure types, and provide recommendations for failure elimination attempts.

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 work.

### 4.2 Interference analysis

In a straightforward approach, to be able to establish interference-freedom of a library of aspects in terms of Definition 10 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, the following definition will be used:

**Definition 11** Given two correct aspects $A$ and $B$, we say that $A$ can be woven before $B$ if for every system $S$ satisfying the assumptions of both $A$ and $B$ the following two properties hold: Weaving $A$ into $S$ preserves the assumption of $B$ (the $KP_{AB}$ property), and weaving $B$ into the resulting system $(S+A)$ preserves the guarantee of $A$ (the $KR_{AB}$ property).
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 10 for $n = 2$).

4.2.1 Proving Interference Freedom

Recall that the assumption $P_A$ may now be divided to $PR_A$ for the reachable part and $PU_A$ for unreachable, and similarly for $P_B$.

To prove that A can be woven before B, we need to show that the following statements hold:

$$KP_{AB} \equiv \forall S([S_{\text{reach}} \models PR_A \land PR_B] \land [S_{\text{unreach}} \models (L_A \rightarrow PU_A \land L_B \rightarrow PU_B)])$$
$$\rightarrow [(S + A)_{\text{reach}} \models PR_B \land (S + A)_{\text{unreach}} \models L_B \rightarrow PU_B)]$$

(“Keeping the Precondition of B when weaving A before B”) and

$$KR_{AB} \equiv \forall S([S_{\text{reach}} \models R_A \land PR_B] \land [S_{\text{unreach}} \models L_B \rightarrow PU_B])$$
$$\rightarrow [(S + B)_{\text{reach}} \models R_A]$$

(“Keeping the Result of A when weaving A before B”) In the same way, to prove that B can be woven before A we need to show $KP_{BA}$ and $KR_{BA}$.

Notice that ”A can be woven before B” and ”B can be woven before A” are two distinct statements, as 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 Section 4.2.5. The order of weaving will matter, for example, in the Composition Filters model [5], 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.

The following theorem shows that Definition 11 is a special case of Definition 10 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 10, aspects \(A\) and \(B\) do not interfere if the following two statements are true:

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

and

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

Let us show that if \(A\) and \(B\) are correct aspects, and \(A\) can be woven before \(B\) (i.e., the \(KP_{AB}\) and \(KR_{AB}\) statements hold), then \(OK_{AB}\) holds.
By Definition 7, the \(OK_{AB}\) statement can be rewritten as follows, making the two parts of aspect assumptions explicit:

\[
OK_{AB} \triangleq \forall S\[((S_{reach} \models PR_A \land PR_B) \land (S_{unreach} \models (L_A \rightarrow PU_A \land L_B \rightarrow PU_B)))
\rightarrow ((S + A) + B \models R_A \land R_B)]
\]

Let \(S\) be a system such that \((S_{reach} \models PR_A \land PR_B) \land (S_{unreach} \models (L_A \rightarrow PU_A \land L_B \rightarrow PU_B)))\). In particular, \(S\) satisfies \((S_{reach} \models PR_A) \land (S_{unreach} \models L_A \rightarrow PU_A))\), which is the assumption of \(A\). Aspect \(A\) is correct, thus, whenever it is woven into a system satisfying its assumptions, \((PR_A, PU_A)\), the guarantee of \(A\), \(R_A\), holds in the resulting woven system. In particular, \(S + A \models R_A\), which is equivalent to \((S + A)_{reach} \models R_A\). This, together with \(KP_{AB}\), implies that

\[
(((S + A)_{reach} \models R_A \land PR_B) \land ((S + A)_{unreach} \models L_B \rightarrow PU_B))
\]

The above argument is correct for any system \(S\) satisfying the assumption, thus the following statement holds:

\[
KP'_{AB} \triangleq \forall S\[((S_{reach} \models PR_A \land PR_B) \land (S_{unreach} \models (L_A \rightarrow PU_A \land L_B \rightarrow PU_B)))]
\rightarrow [((S + A)_{reach} \models (R_A \land PR_B) \land (S + A)_{unreach} \models L_B \rightarrow PU_B)]
\]

42
KR\textsubscript{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\textsubscript{AB} satisfies RB, and we have

\[
KR'_{AB} \triangleq \forall S((S_{\text{reach}} \models RA \land PR_B) \land (S_{\text{unreach}} \models LB \rightarrow PU_B)) \\
\rightarrow [(S + B)_{\text{reach}} \models RA \land RB]
\]

Now we can combine KP\textsubscript{′}\textsubscript{AB} and KR\textsubscript{′}\textsubscript{AB} by substituting S+A instead of S into KR\textsubscript{′}\textsubscript{AB}. As a result we will obtain the desired property, OK\textsubscript{AB}.

The proof that OK\textsubscript{BA} follows from KP\textsubscript{BA} and KR\textsubscript{BA} is symmetric. Together we obtain that if all the premises of the theorem hold, A and B do not interfere.

Q. E. D.

**Theorem 4** Let A\(_1, \ldots, A_N\) be aspects with the specifications \((P_1, R_1), \ldots, (P_N, R_N)\) 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 \(A = \{A_1, \ldots, A_N\}\) is interference-free.

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

**Lemma 5** For every set of \(n \geq 2\) aspects \(\{A_1, \ldots, A_n\}\) satisfying their specifications \((P_1, R_1), \ldots, (P_n, R_n)\), if for every pair of indices \(i, j\) KP\(_{i,j}\) is true, then for every base system \(S\) such that \(S \models P_1 \land \ldots \land P_n\), the following holds: For every \(0 \leq k < n\), \((\ldots (S + A_1) + \ldots + A_k) \models P_{k+1} \land \ldots \land P_n\) (where the case of \(k = 0\) means that no aspects are woven into the system S).

**Proof (Lemma 5).**
The proof is by induction on k.

Basis: \(k = 0\). We need to show that \(S \models P_1 \land \ldots \land P_n\), 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 \(S \models P_1 \land \ldots \land P_n\). Let us denote the system \((\ldots (S + A_1) + \ldots + A_{m-1})\) by \(S'\). We need to show that \((S' + A_m) \models P_{m+1} \land \ldots \land P_n\). 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, \(S' \models P_m \land \ldots \land P_n\), and, in particular,
S′ |= P_m ∧ P_i. Together we have that indeed (S′ + A_m) |= P_i for every 
m + 1 ≤ i ≤ n.
Q. E. D. (Lemma 5)

Now let us prove Theorem 4. Let us be given a subset of 1 ≤ n ≤ N
aspects from A, and a permutation (i_1, . . . , i_n) of their indices, indicating the
chosen weaving order. Without loss of generality, we can call them (1, . . . , 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, . . . , n
on the permuted library will give the same sequence of aspects as the order
i_1, . . . , i_n on the original one.) We need to prove that for every base system
S, if S |= (P_1 ∧ . . . ∧ P_n) then ( . . . (S + A_1) + . . . + A_n) |= R_1 ∧ . . . ∧ 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 |= P_1) →
(S + A_1 |= R_1) holds.
Induction step: We assume that the statement holds for any 1 ≤ k < m
aspects from the n given, and prove it for k = m. Let us be given a
base system S satisfying P_1 ∧ . . . ∧ P_n. We will denote by S′ the system
( . . . (S + A_1) + . . . + A_{m−1}). From the induction hypothesis we have that
S′ |= R_1 ∧ . . . ∧ R_{m−1}. Lemma 5 is applicable here, so we also have that
S′ |= P_m ∧ . . . ∧ P_n. In particular, S′ |= P_m. Thus, as A_m is correct according
to its specification, S′ + A_m |= R_m. And for every i ≠ m, 1 ≤ i ≤ n, the
KR_{i,m} property holds, thus from the fact that S′ |= P_m ∧ R_i it follows that
indeed S′ + A_m |= R_i. Together we get that indeed ( . . . (S + A_1) + . . . + A_m) |=
R_1 ∧ . . . ∧ R_m.
Q. E. D. (Theorem 4)

When checking interferences between two aspects, in the general case,
when at least one of the aspects given is strongly invasive, or its category
is unknown, the assumption of both aspects should be given as a pair,
(PR, PU), as described in Definition 6. If one of the aspects, A, is known to
be weakly invasive, the user can provide its specification in the usual way,
as a pair (PR_A, R_A), and the PU_A part of the specification should be auto-
matically created and will be equal to False. In this case the requirement
from the unreachable part of the base system turns out to be L_A → False.
This requirement will be satisfied by the base system S if and only if aspect
A is weakly invasive with respect to S, because only for a weakly invasive
aspect there are no unreachable states of the base system which are also last states of the aspect, and then $L_A$ equals $False$ in the unreachable part of $S$.

4.2.2 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 Section 7.2.2, 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 $P_A \land P_B \land P_C$ does not guarantee $R_A \land R_B \land R_C$ 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 $R_A \land P_B$, we pose too strong a restriction, because we are interested in this statement only for base systems in which aspect A is present.

4.2.3 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 12** 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: $PR_A \land PR_B$, $(L_A \rightarrow PU_A) \land L_B \rightarrow PU_B$ $R_A \land PR_B$, $R_A \land R_B$
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 Section 4.3, 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.

4.2.4 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.

**Case 1:** The composition of the two aspects is not feasible (i.e., its feasibility check failed). This, in particular, means that the specifications of one or both aspects must be changed to enable them to co-exist in one system. There are several possibilities:

- The specifications of the aspects were unnecessarily demanding, and it is enough to refine the specifications, without changing the advice. For example, if an unnecessary assumption was erroneously added to the
specification of one of the aspects. It can also happen that the guarantee of aspect A demanded some property that does not, in fact, have to hold in a certain special case, and this special case was impossible before aspect B appeared in the system.

- One special case of a too demanding specification is when there is a cooperation relationship between the aspects. One special case of cooperation is dependency. Then aspect B is dependent on aspect A’s presence in the system, and aspect A, woven first, enables the functioning of aspect B, so $P_B$ does not need to be true at the beginning, and might even contradict $P_A$. In this case only $P_A$ has to be assumed to hold in the base system, and the verification process from Section 4.3 should be repeated with this new assumption. In some cases only $R_B$ will be required in the system where both aspects are present, because the only purpose of aspect A was to enable the operation of aspect B. Then $R_A$ and $R_B$ should also be allowed to contradict each other. And in case of a cooperation relationship which is not dependency, $R_A$ and $P_B$ might contradict each other, because what is important is only the result of the cooperation of A and B, but still $R_A \land R_B$ should be required to hold in $((S + A) + B)$ provided $P_A \land P_B$ was true in $S$. In any of the above cases, when an alternative, weakened specification is chosen, the appropriate verification steps from Section 4.3 should be run.

- A change in one or both aspects advice is needed. If this change is possible, it will also imply a change in specification, and then the composition will become feasible.

Case 2: Aspect A interferes with aspect B, but the composition of A and then B is feasible. This means that the specifications of the aspects were consistent with each other, and there are three possible causes of the interference:

- The specifications of the aspects, though not contradicting, were still too demanding, and changing the specification alone can eliminate the interference. For instance, consider the case of aspects A and B applied to a grades managing system. Let aspect B be the GR aspect from Section 3.3 responsible for giving bonuses and factors to grades,
and let aspect A be in charge of gathering statistics on grades, such as their average, median and distribution, and shows the appropriate histogram. Before we are aware of the possibility of grades exceeding 100 in the system (which appear only after aspect B is added), it is reasonable for aspect A to assume that all the grades are between 0 and 100, and thus guarantee that all the values it produces are also below 100, and all the grades in the histogram are distributed between 0 and 100. The assumption of A is violated in presence of aspect B, even though both aspects might work fine together. In this case widening the values margins for A would solve the problem.

- Another possibility is that the interference was caused only by the implementation of the advice of one of the aspects or of both and it is possible to change the implementation of the aspects in such a way that the interference will no longer exist, and such a change will be the desired solution in this case. This is the case in the example in Section 7.2.1.

- No reasonable change in the advice is able to eliminate the interference. The cause might be that the assumptions of the aspects were too weak, and thus the verification failed. In this case it might be reasonable to define some more narrow sub-class of base systems into which both aspects can be woven in the given order. This can be achieved by defining an additional constraint, $\varphi$, that should hold in the base system together with $P_A$ and $P_B$ in order to make the weaving possible. Then the verification process described in Section 4.3 should be repeated with the assumption $P_A \land P_B \land \varphi$ (instead of just $P_A \land P_B$). If it succeeds, the constraint $\varphi$ will be added to the description of the interaction between A and B.

A typical error analysis will be shown for the interference detected in the example of Section 7.2.1.

### 4.2.5 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 issue of possibly shared join-points is discussed in more detail in Chapter 5.

The following definitions will be useful to us:

**Definition 13** 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 14** 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 6  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 $S+(A,B) \cong (S+A)+B \cong (S+B)+A$.
Q. E. D.

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 7  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 $S + (A, B) \cong (S + A) + B$.

Q. E. D.

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 (e.g., [11, 18]) can be used.

4.3 Proof Implementation

Our interference detection method is based on Theorem 3, and uses MAVEN as a subsystem. To show that aspect A can be woven before B, perform the following steps:

1. For $KP_{AB}$, first construct a refined tableau of the conjunction of the assumptions of the aspects, $(PR_A \land PR_B, L_A \rightarrow PU_A \land L_B \rightarrow PU_B)$. Then use MAVEN to weave aspect A into this tableau. In order to verify that the woven tableau satisfies $P_B$, perform the procedure from Section 3.2.4 with the woven tableau as the base system checked, and $(PR_B, L_B \rightarrow PU_B)$ as the verified specification.

2. For $KR_{AB}$, only the reachable part of the resulting system should be checked, thus it is enough to apply the procedure from Section 3.2.3 to aspect B and the refined specification $((R_A \land PR_B, L_B \rightarrow PU_B), R_A)$, thus checking that the guarantee of A is preserved by the advice of B when it is woven into a refined tableau of $R_A \land P_B$.

3. If both verifications above succeed, then aspect A can be woven before B.
In the special case when both aspects A and B are weakly invasive, a simplified verification procedure can be applied:

1. For $KP_{AB}$, build a tableau that corresponds to the conjunction of the assumptions of the aspects, $PR_A \land PR_B$, weave the advice of A and show that the assumption of B, $PR_B$, is true of the result. That is, run MAVEN to show $T_{PR_A \land PR_B} + A \models PR_B$.

2. For $KR_{AB}$, build a tableau that corresponds to the conjunction of the assumption of B and the guarantee of A, $R_A \land PR_B$, weave the advice of B, and show that the guarantee of A, $R_A$, still holds for the result. That is, run MAVENTO show $T_{R_A \land PR_B} + B \models R_A$.

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 Section 4.2.4).

In Figure 4.1 the verification process is shown for the case when a new aspect is added to an existing aspect library. This process relies on Theorem 4. Note that the same process can be performed when a whole unchecked library is given: the “checked part” of the library can be incrementally built by adding aspects one by one. An example of applying the interference checks to a small library is given in Chapter 7.

The above method is sound, due to the above note and Theorems 3 and 4, but not complete. 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,
Figure 4.1: Incremental interference-detection procedure for a library of correct aspects.

because the assumption $R_A \land P_B$ 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 $S \models R'_A \land P_B$, then $S + B \models R_A$). Notice some non-symmetry in the above statement - we have to assume $R'_A$, but can guarantee only $R_A$, because that is the property proven by the successful OK$_{AB}$ check. In fact, by demanding that aspect $B$ will preserve $R_A$ when woven into any system that satisfies $R_A \land P_B$, we pose too strong a restriction, because we are interested in this statement only for base systems in
which aspect A is present.

4.4 Related Work

The way in which assume-guarantee specifications of aspects described in Section 3.1.1 are used to define interference freedom is analogous to interference freedom among processes in shared-memory systems [37]. 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 [37], 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 [20] and as part of the paper [15], 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 Section 4.3, 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 [11, 18]. 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 tech-
niques 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 [38, 43]. Slicing techniques for aspects [44, 3, 41] can also be used for such detection. Since such potential influence is often harmless, many false positives can result.

This chapter is an extension of our paper [20] and of the part on interference detection from the paper [15]. These papers only treated interference among weakly invasive aspects. Here this treatment has been extended to the case when strongly invasive aspects can also be in the library.
Chapter 5

User Queries for Specification Refinement
Treating Shared Aspect Join Points

5.1 Semantics of aspect behavior at a common join point

In AspectJ, the most commonly used aspect language, aspects at a common join-point are applied one after another, and each time before performing an advice the pointcut condition is re-checked. As a result of such a semantics, when a base system arrives at a join-point matched by an aspect A, it is not necessarily the case that the advice of A is immediately executed. Other aspects in the system might also match this join-point, and thus some other advices might be executed before the advice of A, changing the state of the system in which A will be applied. Moreover, A’s advice might not be executed at all, in case one of the previously executed aspects left the system in a state which is no longer a join-point of A.

Thus the execution of the woven system from the moment it arrives at a join-point matched by some of its aspects is determined not only by the set of matching aspects, but also by the order of their application at this point. If this order of application is not explicitly prescribed by the user, the non-
determinism of aspect application may result in different states. However, the fact that different orders of advice application lead to different resulting states does not necessarily mean that the aspects semantically interfere. Consider the following example, from [1]: Several aspects are defined for systems in which messages of type String are sent between objects. Two of these aspects are Logging and Encryption. Both aspects are applied at the same join-points - when a message is sent in the system - and different orders of their application will result in different states of the system. If Logging is executed before Encryption, the logged message will be the original one, otherwise it will be the encrypted message produced by the Encryption aspect. In [1] such a situation is considered interference between the two aspects, but in fact the decision on whether it is interference should depend on the aspects’ specifications. In our example, the goal of the Encryption aspect is to encrypt every message before it is sent to the server. Consider the following possible specifications of the Logging aspect, described more formally in Section 7.1.1:

1. The log should record all the messages sent as they were originally written by the user, so that the user can view the contents of the messages.

2. The log should record all the messages as they were actually sent to the server in order to compare the sent messages to the received ones (as received) and verify that no messages got lost or garbled.

3. The goal of the Logging aspect is to measure the network activity of the system. Thus, though the contents of the messages are written to the log, they are of no importance to the user, and what matters is only, e.g., the times of the messages sent and the number of lines in the log.

4. The logging records all the attempts to send a message, even if they are aborted for whatever reason. It logs each message as it was attempted to be sent.

All the cases above can happen in our example system, and different orders of application of the two aspects at their common join-point lead to different resulting states, but not in all the cases above do the aspects interfere. The
requirements from the Encrypting aspect are never violated by Logging, no matter in what order they are executed, but in variants (1) and (4), the goal of Logging will not be reached if the Encrypting aspect is applied first, but the opposite order is permissible. In variant (2), only applying Encrypting after Logging will cause a problem. Moreover, in variant (3) applying the aspects in any order will not violate the requirements from Logging or from Encrypting, thus there is no interference. As will be shown later, an Authorization aspect can also be applied, further complicating the situation.

The above example shows the need to analyze possible semantic effects of sharing a join-point more deeply. We consider the AspectJ operational semantics with one restriction: no join-points of one aspect inside advice of another are possible. Under it, the weaving is performed by a three-step strategy: First all the places in the code of the base program that are matched by the static part of some aspect’s pointcut, are identified. Such places are called \textit{shadow join-points}. Note that a shadow join-point is usually defined by a place in the code of the program, but sometimes can contain additional information. Second, after shadow join-point identification, at each such join-point the weaving order of the potentially applicable aspects is defined (an aspect is considered potentially applicable if the static part of its pointcut matches the current shadow join-point). The weaving order can potentially be determined dynamically, upon arrival of the computation at a join-point. Finally, when a computation arrives at a join-point, each of the potentially applicable aspects, one by one and in the previously defined order, is checked for full applicability and immediately executed if indeed applicable (i.e., if both static and dynamic parts of the pointcut are matched by the current state). All the rest of the paper refers to this semantics, and if a different semantics is chosen, different reasoning might be needed. This operational semantics shows the need to reason about the part of computation between the first moment it arrives at some shadow join-point and the moment it leaves this join-point, which includes all the aspect applications performed at the join-point. We need some new terminology to make this reasoning easier. First of all, we need a name for the period of interest:

\textbf{Definition 15} A sequence of states \(s_1, \ldots, s_k\) in a computation of the woven system is called a pointcut occurrence of aspect \(A\) if \(s_1\) is the state when a join-point of \(A\) is first reached (that is, \(s_1\) is matched by the full
pointcut descriptor of A, and the previous state is not), and \( s_k \) is the state when the computation is about to leave the corresponding shadow join-point, after application of all the appropriate aspect advices according to the current weaving policy (that is, \( s_k \) is matched by the static part of the pointcut descriptor of A, and the next state is not).

Two aspects share a join-point if they have at least one overlapping pointcut occurrence. Note that overlapping pointcut occurrences do not have to coincide, as it might be a case that an execution arrives at a state \( s \) matched by the pointcut descriptor of aspect B, and by the static part of pointcut descriptor of A and B, but not matched by the dynamic part of A’s pointcut descriptor, and only the execution of aspect B at \( s \) will result in a state in which both static and dynamic parts of A’s pointcut hold. In such a case the pointcut occurrence of A will be contained in the pointcut occurrence of B, but not vice versa. For example, consider the case of aspects A and B applied to a grades managing system. Let aspect B be the GR aspect from Section 3.3 responsible for giving bonuses and factors to grades, and let aspect A be in charge of enforcing a required grades format, by rounding non-integer grades and replacing all the grades above 100 by 100. Both aspects are applied before the publication of the grades, so the static parts of their pointcuts are the same. However, aspect A should be applied only if the grade to be published is not an integer or exceeds 100. Clearly, a computation might arrive at a place before grade publishing with an integer grade below 100, thus matched by the dynamic part of GR’s pointcut only (and not of A’s), but as a result of GR’s modifications, a non-integer grade or a grade above 100 is obtained, bringing the computation to a state that is matched by A’s pointcut as well.

Previously, two kinds of join-points have been examined: shadow join-points and actual join-points. A shadow join-point of aspect A, as mentioned above, is a place in the code of the base program that is matched by the static part of A’s pointcut. An actual join-point of A is a state in a computation of the system at which the advice of A is actually applied. However, for the purpose of our analysis, a third type, arrival join-points, is needed:

**Definition 16** A state \( s \) in a computation of the woven system is called an arrival join-point of aspect A if \( s \) is the first state of a pointcut occurrence of A - the state when a join-point is first reached in that occurrence.
Note that an arrival join-point differs from a shadow join-point because it is matched also by the dynamic part of A’s pointcut descriptor. It also differs from an actual join-point because other aspects can intervene and even prevent A from reaching an actual joinpoint.

Figure 5.1 presents an example for the above definitions. The state $s_1$ is a shadow join-point of aspects A, B and C. It is also matched by the dynamic parts of A’s and B’s pointcuts, thus both for A and for B the sequence $s_1, \ldots, s_4$ is a pointcut occurrence, and $s_1$ is an arrival join-point. However, their actual join-points differ: A is applied at $s_1$, and B - at $s_2$. For aspect C, the sequence $s_3, s_4$ is a pointcut occurrence, meaning that the dynamic part of the pointcut of C becomes true only after B is executed. $s_3$ is also the state where C is executed, thus being both the arrival and the actual join-point of C.

The pointcut of A identifies states either before or after some events of interest, but the definitions above are applicable for both cases. And if A is an around advice (i.e., advice that augments the event of interest, and sometimes even replaces the functionality of the base system), it either has a proceed statement, and then can be viewed as a combination of two advice pieces - one before, and one after the corresponding event, or A has no proceed, and then can be viewed as a before advice, one of the effects of which is a change in the program counter of the base system. If indeed A’s advice changes the program counter of the base system, the end of its execution is also the end of the current pointcut occurrence - both according to our intuition and to the definitions above.

Now let a state $s$ be a join-point matched by aspect A, that appears
inside pointcut occurrence $\pi$. We distinguish between four possible cases of other aspects’ behavior that can influence the result of weaving $A$ into a system:

1. Aspect $B$ executed before $A$ in $\pi$ changes a value of some variable used by $A$ as an input to its computations.
2. Aspect $C$ executed after $A$ in $\pi$ changes a value of some variable updated by a computation of $A$.
3. Aspect $D$ executed before $A$ in $\pi$ brings the system to a state $s'$ which is not a join-point of $A$ any more.
4. Aspect $E$ executed after $A$ in $\pi$ invalidates the condition on which the join-point predicate depended, thus removing a join-point of $A$ after $A$ has already been executed at it.

The following analysis of those cases enables us to determine whether the above influences actually cause an interference: Let the specification of $A$ be given by an assumption-guarantee pair $(P_A, R_A)$, where $R_A$ is the guarantee of $A$ that must hold in any woven system containing $A$, provided the system into which $A$ has been woven satisfied the assumption of $A$, $P_A$. Note that $A$ can be woven into a system that does not satisfy $P_A$, but then $R_A$ is not guaranteed to hold in the resulting system. We denote by $V_{in}(A)$ a set of variables $A$ uses as input to its computations, and by $V_{out}(A)$ - a set of variables in which $A$ stores the result of its computations. An intuition for the discussion below is demonstrated in Figure 5.2.

**1. Change Before (CB).** In case an aspect $B$ executed before $A$ at $s$ changes a value of a $v \in V_{in}(A)$, the result of $A$’s calculations might differ from the one we would get if the value of $v$ has not been changed from the moment the computation arrived at $s$ till the moment the advice of $A$ was executed. If the guarantee of $A$ is formulated in terms of a specific connection between the value of $v$ when we arrive at a join-point and the value of $v$ after the computation of $A$ is finished, $R_A$ will be violated. (This can happen, for instance, in variant (1) of the Logging and Encrypting example above: we anticipate that the message string written to the log is the one created by the user and readable by the user, but if Encrypting is executed before Logging, what we actually get in the log is the encrypted message, because
Figure 5.2: Cases of influence of aspects at a shared join-point
the contents of the message was changed by the Encrypting aspect.) Note that if the requirement for correctness of A’s calculations binds the values at the end of A’s execution only to the values at the beginning of execution of A (as in variants (2) and (3) of the Logging and Encrypting example, with Encryption before Logging), it will not be violated in this case (for variant (2) only the final message contents are important, and for variant (3) message contents are not important at all).

2. Change After (CA). In case some aspect C executed after A at \( s \) changes a value of a \( v \in V_{out}(A) \), the guarantee of A will be violated if it required preservation of the result of A’s computation till some future point in the execution where the value of \( v \) is used. (As in variant (2) of the example, when Logging occurs before Encryption). Otherwise, as in variant (3) of the example with Logging before Encryption, the guarantee of A will not be influenced. If, indeed, a requirement for preservation of the value of \( v \) till some state \( use_v \) is part of A’s guarantee, then part of A’s assumption should be that in the base system the value of \( v \) is not modified from the actual place of application of A’s advice till arrival to the \( use_v \) state.

3. Invalidation Before (IB). In this case there is no state in A’s pointcut occurrence at which A is executed. Such a situation happens, for example, with Logging and Authorization aspects from Section 7.1.1 when the Authorization aspect is applied before Logging and the authorization of the user fails, thus preventing message sending, and removing the join-point of the Logging aspect. In variant (4) of the Logging specification, this leads to violation of the guarantee of Logging, as a message was prepared for sending and should have been logged, but the Logging aspect never has a chance to be applied, because the authorization failure finishes the pointcut occurrence.

4. Invalidation After (IA). In this case A is executed at some point at which it shouldn’t have been applied, because when arriving at the point of interest, the weaver “does not know” that the reason for A’s application will be removed by one of the aspects coming after A. If the specification of A requires that it is applied only if the join-point is followed by some event, and this following event is removed by another aspect, then the specification of A is violated. This is the case, for example, in variants (1), (2) and (3) if the Authorization aspect is applied after Logging and the authorization of the user fails. Note that in variant (4), on the other hand, the guarantee of
Logging is not violated if Logging precedes Authorization.

5.2 Specification of aspects with possibly shared join-points

5.2.1 Guided Specification Construction

An assume-guarantee specification of aspect A is a pair of LTL formulas, \((P_A, R_A)\), where \(P_A\) is the assumption of the aspect about all the base systems into which it can reasonably be woven, and \(R_A\) is the guarantee of the aspect, that must hold after A is woven into any system that satisfied the assumption \(P_A\). Generally, the basic specification of the aspect is clear, as will be seen in the examples in Section 7.1.1. The refinement of the basic specification for the case of possibly shared join-points, on the other hand, is not obvious, thus the procedure presented below is useful.

From the analysis in Section 5.1 the need for the following predicates arises:

- \(at(ptc(A))\): assuming that \(ptc(A)\) is the predicate defining A’s pointcut, the predicate \(at(ptc(A))\) means that the computation has just arrived at a join-point of A. It is useful for reasoning about what happened in the computation after the moment it arrived at a possibly shared join-point. In fact, this is the predicate marking the arrival join-points of A.

- \(in_{ptc\_occ}(A)\): this predicate marks all the states of A’s pointcut occurrence. It becomes true at the same time as \(at(ptc(A))\), and then continues to hold while the computation stays at the same shadow join-point. When the computation leaves the shadow join-point, \(in_{ptc\_occ}(A)\) becomes false again.

- \(after_{prev\_asp}(A)\): this predicate becomes true at the moment the weaver has applied all the aspects that preceded A at the current shadow join-point, according to the algorithm of the current weaver. This predicate is used to refine the definition of A’s pointcut because now A should only be applied at states satisfying both \(ptc(A)\) and \(after_{prev\_asp}(A)\), (which matches the definition of the set of all the
actual join-points of A). In addition, the predicate is used in assumptions added to A’s specification to express the desired behavior of the system from the moment its computation arrives at a join-point of A till the moment it leaves the current pointcut occurrence.

- \(\text{asp}_{-}\text{ret}(A)\): this predicate describes the possible return states of the aspect. This is needed for some of the cases below. Typically, the aspect return state has the same control location as the join-point state (the values can change, but not the program counter of the join-point). For the Logging aspect, for example, the base state is actually not changed, and only the log (local to the aspect) is modified. However, it does not have to be so in general. Thus in order to define the \(\text{asp}_{-}\text{ret}(A)\) predicate, the user is proposed a default predicate, automatically constructed by the system as described in Section 3.2.1. This default predicate can then be manually modified.

Using the above predicates, all the requirements mentioned in Section 5.1 can be expressed as additions to the assumptions of the aspects, though often not all of the predicates are needed in a given application. Below we present a way to express each of the additional requirements.

The construction of the refined specification can be automatic, but user-guided: questions are presented to the user, and the answers to these questions determine the new requirements. The construction process will thus be as follows:

**Step 1** Here we will treat the dependency of our aspect, A, on its input variables, in order to find out whether the values of the input variables need to be preserved between the arrival and the actual join-points of A (in order to be able to treat the “change before” case from Section 5.1). The user is asked the following question:

**Q. 1:** Are there any input variables of A for which the advice of A depends on the value as it is at the arrival join-point and not as it is when the advice of A actually starts its execution?

- If yes, the user should provide a list of variables for which such a dependency exists.
For each variable $v$ in the list, we add the following $CB$ (for “Change Before”) statement to A’s assumption:

$$CB(v) = G[(at(ptc(A)) \land v = V) \rightarrow (v = V \land [(after \_ prev \_ asp(A) \land v = V) \lor \neg in \_ ptc \_ occ(A)])]$$

where $V$ is a logical variable keeping the value of $v$ as it was at the arrival to the join-point. Note that no restriction is posed on the value of $v$ outside A’s pointcut occurrence.

- If there are no variables in the list, nothing is added to the specification of A at this step.

**Step 2** Here we treat the case when part of the effect of the aspect is modification of some state variables, and this effect should be preserved till some point in the future of the computation. This is important for the “change after” case from Section 5.1. The questions asked here are:

**Q. 2:** Are there any state variables of the system into which A is woven the value of which should be preserved after A’s execution is finished? (For example, variables modified by A, or variables that are semantically connected to A’s local variables.)

- If yes, the user is asked to fill in a table with two columns: the first column is the name of the variable, $v$, and the second is a state predicate $use_v$ describing the state of the woven system until which the value of $v$ should be preserved. For example, for variant 2 of the Logging aspect, that logs messages as they are sent to the server, the message should not change between the moment it has been logged and the moment it is actually sent. Thus the $use_v$ predicate will describe the moment of actual sending of the message (see Section 7.1.1 for more details). After the table is filled out, for each variable $v$ with state predicate $use_v$ in the table, we add the following $CA$ (for “Change After”) statement to the assumption of A:

$$CA(v) = G[(asp \_ ret(A) \land v = V) \rightarrow (v = V \land use_v \land v = V)]$$
where $V$ is a logical variable keeping the value of $v$ as it was at the end of the execution of A’s advice.

- If there are no variables in the list, nothing is added to the specification of A at this step.

Step 3 In this step we construct requirements corresponding to the “invalidation before” case in Section 5.1. Before the problem of common join-points in modular verification was considered, there existed an implicit assumption that all the arrival join-points of an aspect are its actual join-points. But when a join-point might be shared, this is not necessarily so, because the join-point can be invalidated; thus an additional explicit assumption of this possibility is needed. The user is asked the following question:

Q. 3: Does it have to be that each time an arrival join-point of A is reached, A is eventually executed at it? That is, is it an error if previously executed aspects invalidate the condition for A’s application?

- If no, nothing is added to the assumption of A in this step.
- If the answer was “yes”, the following $IB$ (for “Invalidation Before”) statement is added to the assumption of A:

$$IB \equiv G[\text{at}(ptc(A)) \rightarrow (\text{in}_\text{ptc}_\text{occ}(A) \cup (\text{after}_\text{prev}_\text{asp}(A) \land ptc(A)))]$$

Step 4 The goal of this step is to enable the verification process to treat the case of “invalidation after” from Section 5.1. We ask the user the following questions:

Q. 4.1: Does the reason for a state to be A’s join-point lie in the future of the computation? That is, does A’s pointcut descriptor refer to any event (immediately) following the join-point? For example, is the advice of A a “before” advice?

- If no, nothing is added to the assumption of A in this step.
- If the answer was “yes”, the next question is asked:

Q. 4.2: Is it an error if the advice of A is performed, but the presumably-following event does not follow? (For example, because the future computation was changed by other aspects)
• If the answer is “no”, nothing is added to A’s assumption in this step.

• If the answer is “yes”, the user is required to provide a state predicate, \(foll\_event\), meaning that the desired following event has just occurred. The user is then prompted to provide some optional restrictions on the values immediately after A’s execution, the values at the moment the desired event occurs, and the connections between them (including, for example, value preservation). The restrictions should be given in the form of two predicates: \(vals\_after\_asp\) and \(vals\_at\_foll\_event\). The default value for both predicates is \(true\).

• The following \(IA\) (for “Invalidation After”) statement is then added to the assumption of A:

\[
IA \triangleq G[(asp\_ret(A) \land vals\_after\_asp) \rightarrow (in\_ptc\_occ(A) \lor (foll\_event \land vals\_at\_foll\_event))]
\]

Some optimizations, that would enhance the verification process and/or simplify the formulas, can be performed in certain cases of combinations of answers to the above questions. For example, for technical reasons we would prefer to have the \(U\) operator, rather than \(W\), in aspect assumptions. One of the cases in which such a replacement can be obtained is when both the \(CB\) and \(IB\) statements are added to the specification. Then we can replace the \(W\) operator in \(CB\) by \(U\), and thus also simplify the formula to become:

\[
CB(v) = G[(at\_ptc(A)) \land v = V) \rightarrow (v = V \lor (after\_prev\_asp(A) \land v = V))]
\]

The reason is that due to \(IB\) we know that an actual join-point of the aspect has to appear in every pointcut occurrence.

After the above automatic modifications, the specification constructed both captures the requirements of the user regarding the desired effect of aspect application, and contains sufficient assumptions to make the modular verification results applicable to systems with aspects sharing join-points.
5.2.2 Using the Refined Specification

The refined specification of aspects is used during all the stages of the full specification and verification process for aspect libraries, where a library of aspects is a collection of reusable aspects grouped together for some common purpose, to be applied together to different systems. Given a library of aspects, two things are important for its usage: correctness of each aspect alone with respect to its assume-guarantee specification, and interference detection among the aspects. The question of possibly shared join-points is already important when the specification of individual aspects is defined. At this stage one of the tools for detection of potential interference at common join-points detection can be run, e.g. [1], and only if a potential interference is detected the specification refinement described in Section 5.2.1 has to be performed for the potentially interfering aspects. (If no tool for potential interference detection is run, all the aspect specifications should undergo the process from Section 5.2.1, to ensure the soundness of the verification process.)

After all the aspects in the library are specified and augmented as described above, tools for modular aspect verification (e.g., the tool described in Section 3) and interference detection (Section 4) can be run.

The effect of our specification-refinement procedure on the result of aspect verification and interference detection is twofold. First of all, after the specification of the aspect is refined, it is possible to prove more aspects correct with respect to their assume-guarantee specifications, because the assumption of the aspect might have been strengthened by additional assertions. Second, for the same reason, if two aspects interfere, the interference will in many cases be detected at an earlier stage of the interference-detection procedure, as violation of the assumption of an aspect might be detected, making it unnecessary to proceed to the guarantee-preservation check.

5.3 Related Work

Ways to detect shared join-points are described in [35, 36]. Several works study shared join-points as a source of possible conflicts, some (e.g., [35]) even see common join-points as the main source of interference among aspects. A language independent technique [13] makes it possible to check
whether an undesired order of aspect application at a shared join-point is possible, where the list of undesired orders has to be explicitly provided by the user. It is implemented in the “Secret” tool for Compose* [4, 34]. However, presenting the list of undesired orders requires a thorough analysis of the system, and also might not reflect all the intended behaviors, as at different states different orders of application might be possible. In [1] another tool for checking potential interference at common join-points is described, applicable for the Compose* language. It checks all the possible orders of aspect applications at a common join-point, and declares a conflict if different orders result in different resulting states. This method is fully automatic, but may lead to many false positives (for example, it would declare that the Encryption aspect interferes with all the variants of the Logging aspect, which is not true, as shown in Section 7.2.3). Additional tools that can be used to check interactions at shared join-points are described in Chapter 4. However, interactions found by these tools are, again, only potentially harmful.

Weaving techniques for conflict resolution at shared join-points appear in [35, 11, 12]. In [35], a first analysis of types of mutual influence of aspects applied at shared join-point appears. This analysis is extended in our paper, though used for a different purpose.

The intended semantics of weaving several aspects at a common join-point in a pre-defined order is addressed in papers on the semantics of aspects, such as [42, 8, 10].

However, as described in [30], not all the conflicts at shared join-points can be resolved by a clever weaving. Thus it is important for the user to be able to detect the conflicts and differentiate between real problems and false alarms.

This chapter is based on the paper [23] (its preliminary version appeared in [22]).
Chapter 6

Additional Extensions of Modular Aspect Verification

In this chapter, modifications to the MAVEN tool are presented, which extend the abilities of the tool. The extended MAVEN tool is available as part of the Common Aspect Proof Environment (CAPE) [25] 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.

6.1 Aspects With Local Memory

It is often the case that an aspect has local memory, i.e., there are variables of the aspect the values of which should be preserved between applications of the advice. In this section changes to MAVEN are described that enable modeling such a behavior.

The following keywords have been added to the MAVEN input format to be used as part of an aspect description:

**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.

**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.

The new **LOCMEM** and **ONRET** directives enable preservation of aspect variables values after the advice execution is finished. Aspect variables are not modified by base system transitions. Thus if variables listed in **LOCMEM** are preserved by **ONRET** as well, then when advice is re-entered their values will be the same as they were after the last advice execution. The **ONRET** directives also enable resetting aspect variable values when exiting the advice, and **LOCINIT** directives - when the advice is (re-)entered.

In the original MAVEN tool, only one kind of aspect variable initialization was possible: on entry to the advice. The division of the initialization to **GLOBINIT** and **LOCINIT** enables correct initialization of local aspect variables even if they are part of aspect local memory (and thus cannot be modified when entering the advice).

One example of an aspect with local memory is the Logging aspect mentioned in Section 5.1 (all its variants). The parts of aspect description that allow local memory treatment are highlighted in Figure 6.1.

### 6.2 Aspects with Full Past-LTL Pointcuts

One of the parts of aspect description in MAVEN format is a list of **POINT-CUT** directives, the disjunction of which is the complete pointcut definition of the aspect. Only current-state expressions are allowed in these directives, but the following technique makes it possible to describe full past-LTL pointcuts.
...
In order to define a pointcut \( \rho \) that contains past LTL operators, the user is required to define a new state variable (e.g., "\( \rho_{\text{state}} \)") and divide the pointcut definition into two parts: add the statement "POINTCUT \( \rho_{\text{state}} \)" to the list of POINTCUT directives, and add the assumption "\( G(\rho_{\text{state}} \leftrightarrow \rho) \)" to the assumptions of the aspect. In this way, when the tableau of an aspect assumption is built, its states are marked with one additional predicate, \( \rho_{\text{state}} \), and the states marked by this predicate are exactly the ones all the paths to which satisfy the past-LTL formula of the pointcut.

6.3 Aspects that Might Have Shared Join-Points

For the purpose of automatic modular verification and interference detection of aspects, the following corrections to the modeling process are performed in order to obtain a correct weaving of advice models into the tableau:

- As follows from the discussion in Step 3 of Section 5.2.1 (treating the case of "invalidation before"), the pointcut definition of \( A \) should be refined to be \( ptc'(A) = ptc(A) \land after\_prev\_asp(A) \), so that \( ptc'(A) \) marks actual and not arrival join-points of \( A \), because only at these points the advice of \( A \) is now executed. This change of the aspect model is done automatically. The \( after\_prev\_asp(A) \) flag is necessary only in the presence of possibly shared join-points. When no such possibility exists, the arrival join-point of \( A \) is also its actual join-point, thus by default \( after\_prev\_asp(A) \) is defined as equal to \( ptc(A) \).

- In order to model an aspect with possibly shared join-points, we need to be able to model returning of the advice to the join-point from which its execution started, so that the same advice will not be applied again at this point, but the other aspects will be able to execute. When several aspects can share a join-point, the weaver has to give them all exactly one chance to be applied at it. This can be viewed as fulfilling a promise to each one of these aspects. Thus a flag \( promise\_ful(A) \) is added to the variables of the weaver for each aspect \( A \). This flag is false when the computation arrives at a shadow join-point, becomes true at the moment the execution of \( A \)'s advice begins, and remains true until the computation leaves this shadow join-point. The aspect will be executed only when both its join-point is reached and the \( promise\_ful(A) \)
flag is \textit{false}. This flag also is important only in case multiple aspects can share A’s join-point. Otherwise, though using the flag would still be cleaner, another solution would be possible: when describing A, differentiate between the join-point state and the state “just after the join-point” in the base system computation. Then define the resumption state of A as the state “just after the join-point”, rather than the join-point state itself. Of course, such a solution would not work in case the join-point might be shared, as it would make impossible the analysis of interference caused by aspects that are executed at the join-point after A’s advice is performed (Cases 2 and 4 from Chapter 5).

• To be able to treat questions 1, 3 and 4 from Section 5.2.1, we need the means to define the $\text{in}_{ptc\_occ}(A)$ predicate. As follows from the definition of pointcut occurrence, $\text{in}_{ptc\_occ}(A)$ is true from the moment the actual join-point of A (marked by $ptc'(A)$ above) is reached, and till the moment the computation leaves the current shadow join-point. The latter is exactly the moment when the static part of A’s pointcut ceases to hold. Thus to complete the definition of $\text{in}_{ptc\_occ}(A)$ it was necessary to identify the static part of A’s pointcut. For that purpose, a new keyword has been added to the MAVEN input format:

\textbf{STATPTC} Describes the static part of the aspect’s pointcut. One predicate appears after each STATPTC directive. Only current-state expressions are allowed, as they are sufficient to represent all the static information. The complete definition of the static part of the pointcut is the disjunction of all STATPTC directives. Note that the set of states matched by STATPTC directives should include the states matched by the full pointcut definition (the POINTCUT directives).

The STATPTC directive is optional (for backward compatibility reasons). In case it is not present, we have no choice but to assume that the static part of the pointcut coincides with the full pointcut definition. This assumption will keep our verification method sound, while making it more incomplete due to the strengthening of the assumptions added for the above mentioned questions 1, 3 and 4.
Chapter 7

Example Library

In this chapter 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. Some of these aspects has been used in examples in [1]. Note that if this library is used in some grade-managing system, the GR aspect presented in Section 3.3 can be added to the library.

7.1 Aspect Specification

The first part of the verification process is to formalize the aspect specifications.

The following predicates will be used to describe the specifications of the aspects:

- $\text{msg\_attempt}$: a predicate which is true when a message is about to be sent, that is, when message sending is attempted. That is the moment before the message-sending procedure is actually called, and the parameters to the method call are represented by the variables $\text{msg\_c}$ and $\text{msg\_t}$, containing the two parts of the message to be sent: the contents and the creation time, respectively.

- $\text{msg\_send}$: a predicate which is true at the moment a message (with contents $\text{msg\_c}$ and creation time $\text{msg\_t}$) is sent.
• \textit{psw\_send}: this predicate is true at the moment a password-containing message is sent.

• \textit{login\_psw\_send}: a predicate that is true at the moment a password is sent from a login screen of the system.

• \textit{login\_psw\_to\_send}: a predicate that becomes true each time before a password is about to be sent from a login screen.

• \textit{in\_log (\textless str \textgreater)}: a predicate that is true if the string ”\textit{str}” appears in the log (useful for specifying a Logging aspect).

• \textit{encrypted(msg\_c)}: this predicate means that the contents of the sent message are encrypted (it is useful for the specification of an Encryption aspect below).

• \textit{permit\_usr\_send}: this predicate means that the current user has enough permissions to send the message (it is useful for the specification of an Authorization aspect below).

• \textit{req\_store\_usr\_psw}: a predicate meaning that “remember my password” button has been checked (and thus an aspect remembering passwords should be applied).

• \textit{psw\_in\_usr}: a predicate that is true if a password appears as part of the user data (for example, as a result of a password-remembering aspect execution).

• \textit{req\_backup}: a predicate meaning that there is an unprocessed request for sending user data for backup.

• \textit{send\_usr}: this predicate means that all the user data is sent (for example, as a result of a Backup aspect execution).

• \textit{button\_pressed}: a flag that means forgetting the password has been reported by the user and not yet treated (useful for an aspect that enables users to retrieve forgotten passwords).

• \textit{quest\_answered}: this predicate means that all the security questions asked during the password-retrieval procedure were answered correctly.
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:

- \( G(psw\_send \rightarrow msg\_send) \) (sending a password is a special case of sending a message in the system)
- \( G(login\_psw\_send \rightarrow psw\_send) \) (sending a password from a login screen is a special case of sending a password in the system)
- \( G(login\_psw\_to\_send \rightarrow msg\_attempt) \) (being about to send a password from a login screen is a special case of being about to send a message)
- \( G(((psw\_in\_usr \land send\_usr) \rightarrow psw\_send)) \) (sending user data that contains a password implies that a password is sent in the system)
- The \( login\_psw\_to\_send \) predicate is related to \( login\_psw\_send \) 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 expressed by the following formula: \( G(login\_psw\_send \rightarrow ((\neg login\_psw\_send) \land (login\_psw\_to\_send \land \neg login\_psw\_send))) \)
- \( G(\neg (msg\_attempt \land msg\_send)) \) (the “before call” pointcut cannot be true when the computation already entered the called function)

### 7.1.1 Aspect Descriptions

We will consider the following 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:

\[ P_E \triangleq G(psw_{send} \leftrightarrow login_{psw_{send}}), \]

\[ R_E \triangleq G(psw_{send} \rightarrow encrypted_{psw}) \]

The pointcut of E can be given as a state predicate \( login_{psw_{to\_send}} \).

**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:

\[ P_M \triangleq true \]

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

\[ R_M \triangleq [G(req_{store\_usr\_psw} \rightarrow (req_{store\_usr\_psw} \cup psw_{in\_usr}))] \]

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:

\[ P_B \triangleq true, \]
The moment the predicate \textit{req\_backup} becomes true is the pointcut of the aspect.

\textbf{Aspect }F\textbf{ }provides a list of security questions to the user, and if the questions are answered correctly, }F\textbf{ guarantees that the user will get his password via an e-mail, and thus retrieves a forgotten password. Like aspect }M\textbf{, aspect }F\textbf{ might add a new button - “Forgot my password” - to the system so that we can define the pointcut of }F\textbf{ as the moment when this button is pressed. }F\textbf{’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\textbf{’s assumption is

\[ P_F \triangleq true \]

And }F\textbf{’s guarantee is

\[ R_F \triangleq [G((button\_pressed \land quest\_answered) \rightarrow F(\text{psw\_send}))] \]

The moment the predicate \textit{button\_pressed} becomes true is the pointcut of the aspect.

\textbf{Authorization aspect (A)}\textbf{ }ensures that a message is sent to the server only if the current user has the needed permissions to communicate with the server. }A\textbf{’s guarantee can be

\[ R_A \triangleq G(msg\_send \rightarrow \text{permit\_usr\_send}) \]

A does not need to assume anything about the base system, and we can take

\[ P_A \triangleq true \]

\textbf{Logging aspect (L)}\textbf{ }logs the message - sending in the system. As described earlier in Chapter 5, there are four variants of the logging aspect in the library:

\textit{L\_1: }Logging all the sent messages as the user originally attempted to send them.

\textit{L\_2: }Logging all the messages that were actually sent to the server (the
message is logged as it was sent).

$L_3$: Logging the frequency of message sending.

$L_4$: Logging all the attempts to send a message (the message is logged as it was originally attempted to be sent by the user).

The pointcut of all the variants of the aspect is the moment before the message-sending procedure is called. More formally, \( \text{msg\_attempt} \) is true. The guarantees of the aspects emerge in the usual way from the purpose of each of them, and are written more formally below. If there would be no possibility of sharing join-points, it would be enough to have the assumption of the logging aspects be that if a message is sent, this very same message was the last one passed as a parameter to the message-sending procedure. More formally,

\[
P_L \triangleq G([\text{msg\_send} \land \text{msg\_c} = C \land \text{msg\_t} = T] \rightarrow ([\neg \text{msg\_attempt}]S(\text{msg\_attempt} \land \text{msg\_c} = C \land \text{msg\_t} = T))\)
\]

The following are possible guarantees of the logging aspect variants:

\[
R_{L1} \triangleq ([F(at(\text{msg\_attempt}) \land \text{msg\_c} = C \land \text{msg\_t} = T \land F(\text{msg\_send})]) \leftrightarrow [F(in\_log(<X,T>))])
\]

meaning that messages that appear in the log are all the sent messages, but as they were first attempted to be sent by the user. (Note that the fact that each message is accompanied by creation time information ensures a one-to-one correspondence between messages and lines in the log.)

\[
R_{L2} \triangleq ([F(\text{msg\_send} \land \text{msg\_t} = T \land \text{msg\_c} = C]) \leftrightarrow [F(in\_log(<C,T>))])
\]

meaning that a message appears in the log if and only if it has been, or will be, sent.

\[
R_{L3} = ([F(\text{msg\_send} \land \text{msg\_t} = T)] \leftrightarrow [F(in\_log(<T>))])
\]
meaning that the log contains all the creation-times of the sent messages.

\[
R_{LA} \triangleq ([F(at(msg\_attempt) \land msg\_c = C \land msg\_t = T)] \\
\quad \leftrightarrow [F(in\_log(<C,T>))])
\]

meaning that the log contains exactly the messages attempted to be sent by the user.

### 7.1.2 Specification Refinement

Aspects Encryption, Authorization and Logging appear to share (part of) their join-points: Logging and Authorization share all of their join-points, and the join-points of Encryption are a subset of the join-points of Logging and Authorization - a special case when not only a message is about to be sent, but it is about to be sent from the login screen. Thus we need to apply the specification-refinement procedure from Chapter 5 to these aspects. In a way of example, the result of applying this procedure to one of the Logging aspect variants is shown in Figure 7.1.

**Logging aspect (L)** The questions and responses for refining the computations of the variants of Logging are shown below.

**Specification refinement for \(L_1\):** The answers for the assumption-construction questions for \(L_1\) are as follows:

Q.1: “Yes”. The aspect depends on the contents and time information of the message as they were at the join-point, thus the values of the variables \(msg\_c\) and \(msg\_t\) should be preserved. Thus, substituting into the template \(CB(v)\), the following statements are added to the assumption of \(L_1\):

\[
CB(c) = G((at(msg\_attempt) \land msg\_c = C) \rightarrow \\
((msg\_c = C) \forall \\
[(after\_prev\_asp(L_1) \land msg\_c = C) \lor \neg in\_ptc\_occ(L_1)])
\]
... Aspect assumption

//general knowledge

LTLSPEC --BASE
G !(msg_attempt & msg_send);

//additional assumption: IB

LTLSPEC --BASE
G (msg_attempt & (Y !msg_attempt)) -> (msg_attempt U (after_prev_asp & msg_attempt));

//additional assumptions: CB(msg_c), CB(msg_t)
//(simplified due to IB)

LTLSPEC --BASE
G (msg_attempt & ! (Y msg_attempt) & msg_c = 0 & msg_t = 0) -> (msg_c = 0 & msg_t = 0) U (after_prev_asp & msg_c = 0 & msg_t = 0));

... //symmetric for all the other combinations of msg_c and msg_t values

//assumption that is common to all the Logging aspects, and
//would have been enough if shared join-points were not possible

LTLSPEC --BASE
G (msg_send & (msg_c = 0) & (msg_t = 0)) -> (msg_c = 0 & msg_t = 0) S (msg_attempt & (msg_c = 0) & (msg_t = 0));

LTLSPEC --BASE
... //symmetric for all the other combinations of msg_c
//and msg_t values

Aspect guarantee

LTLSPEC --AUGMENTED
(F in_log_0_0) <-> (F (msg_attempt & ! (Y msg_attempt) & msg_c = 0 & msg_t = 0));

LTLSPEC --AUGMENTED
... //symmetric for all the other combinations of msg_c
//and msg_t values

... Figure 7.1: Specification refinement for L4.
and

\[ CB(t) = G[(\text{at}(msg\_attempt) \land \text{msg} \_t = T) \rightarrow ((\text{msg} \_t = T) \mathcal{W} [(\text{after\_prev\_asp}(L_1) \land \text{msg} \_t = T) \lor \neg \text{in\_ptc\_occ}(L_1)])] \]

Q.2: “Yes”. The time information of the message should be kept intact till the moment the message is actually sent. There is one entry in the table: the variable \textit{msg\_t}, matched by the \textit{msg\_send} predicate. Thus the addition to the aspect assumption at this stage is:

\[ CA(t) = G[(\text{asp\_ret}(L_1) \land \text{msg} \_t = T) \rightarrow ((\text{msg} \_t = T) \mathcal{W} (\text{msg\_send} \land \text{msg} \_t = T))] \]

Q.3: “No”. If the message will not be sent, it should not appear in the log, thus the advice of \textit{L}_1 should not be applied for it. Nothing is added to the assumption of \textit{L}_1 at this stage.

Q.4.1: “Yes”. The advice of \textit{L}_1 is a “before” advice.

Q.4.2: “Yes”. It is an error if a message that is not sent and will not be sent appears in the log. The desired following event is the event of sending the message (defined by its creation time only, as that is what matters for the purpose of \textit{L}_1). Thus \textit{foll\_event} = \textit{msg\_send}, \textit{vals\_after\_asp} = (\textit{msg} \_t = T) and \textit{vals\_at\_foll\_event} = (\textit{msg} \_t = T). Substituting into the \textit{IA} template, we obtain the following addition to \textit{L}_1’s assumption:

\[ IA = G[(\text{asp\_ret}(L_1) \land \text{msg} \_t = T) \rightarrow (\text{in\_ptc\_occ}(L_1) \cup (\text{msg\_send} \land \text{msg} \_t = T))] \]

**Specification refinement for \textit{L}_2**: The construction of the assumption for \textit{L}_2 is performed similarly to that of \textit{L}_1, with only two differences: An additional variable, \textit{msg\_c}, should be preserved after the aspect finishes its computation (affecting the \textit{CA}(v) and \textit{IA} statements), and no values from arrival join-point should be kept (making \textit{CB}(v) true). Together we obtain that the addition to the assumption of \textit{L}_2 as a result of the guided
specification construction procedure consists of the following statements:

\[
CA(c) = G[(asp\_ret(L_2) \land msg\_c = C) \rightarrow (msg\_c = C) W (msg\_send \land msg\_c = C)]
\]

\[
CA(t) = G[(asp\_ret(L_2) \land msg\_t = T) \rightarrow (msg\_t = T) W (msg\_send \land msg\_t = T)]
\]

and

\[
IA = G[(asp\_ret(L_2) \land msg\_c = C \land msg\_t = T) \rightarrow (in\_pte\_occ(L_2) U (msg\_send \land msg\_c = C \land msg\_t = T))]
\]

**Specification refinement for** \(L_3\): The construction of the assumption for \(L_3\) is almost the same as for \(L_2\), except for the fact that the value of \(msg\_c\) need not be preserved after the aspect finishes its computation (thus giving the same \(CA(t)\) and \(IA\) statements as for \(L_1\)). Thus the addition to the assumption of \(L_3\) is

\[
CA(t) = G[(asp\_ret(L_3) \land msg\_t = T) \rightarrow (msg\_t = T) W (msg\_send \land msg\_t = T)]
\]

and

\[
IA = G[(asp\_ret(L_3) \land msg\_t = T) \rightarrow (in\_pte\_occ(L_3) U (msg\_send \land msg\_t = T))]
\]

**Specification refinement for** \(L_4\): The construction of the assumption for \(L_4\) is almost the same as for \(L_1\), with the following differences only:

- The answers to Question 2.1 and Question 4.1 are negative, as the logged message does not have to be sent, so \(CA = true\) and \(IA = true\) in this case.

- The answer to Question 3 is positive, as all the message sending attempts should be logged, including those aborted because of authorization failure.
Thus the additions to the assumption of $L_4$ are:

$$CB(c) = G[(at(msg\_attempt) \land msg.c = C) \rightarrow ((msg.c = C) \land (after\_prev\_asp(L_4) \land \neg in\_ptc\_occ(L_4)))]$$

$$CB(t) = G[(at(msg\_attempt) \land msg.t = T) \rightarrow ((msg.t = T) \land (after\_prev\_asp(L_4) \land \neg in\_ptc\_occ(L_4)))]$$

$$IB = G[at(msg\_attempt) \rightarrow (in\_ptc\_occ(L_4) \lor (after\_prev\_asp(L_4) \land msg\_attempt))]$$

**Encrypting aspect (E)** The additional assumption of E, constructed by the procedure in Section 5.2.1, emerges from the fact that the encrypted message value should be preserved till (and if) it is actually sent:

$$P_E = CA(msg.c) = G[(asp\_ret(E) \land msg.c = C) \rightarrow (msg.c = C \land (msg\_send \land msg.c = C))]$$

**Authorization aspect (A)** When constructing the assumption of A, all the answers to the questions asked happen to be negative, thus A does not need to assume anything about the base system, and we can leave

$$P_A \triangleq true$$

### 7.2 Interference Check of the Library

The detailed descriptions below refer to the pairwise interference checks performed to check interference freedom of the library after all the aspects are shown correct with respect to their (refined) assume-guarantee specifications.
7.2.1 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):

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

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):

$$\forall S[(S \models G((psw\_send \leftrightarrow login\_psw\_send) \land \neg psw\_in\_usr) \land (true)) \rightarrow (S + F \models G(psw\_send \leftrightarrow login\_psw\_send))]$$

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, note that the conflicting aspects do not share any join-points, and the interference doesn’t emerge from
updating common variables.

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

<table>
<thead>
<tr>
<th>Check type</th>
<th>Result</th>
<th>Model size</th>
<th>Verification Time(msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectE</td>
<td>true</td>
<td>1127</td>
<td>26</td>
</tr>
<tr>
<td>AspectF</td>
<td>true</td>
<td>718</td>
<td>23</td>
</tr>
<tr>
<td>KP_{EF}</td>
<td>true</td>
<td>1374</td>
<td>22</td>
</tr>
<tr>
<td>KR_{EF}</td>
<td>false</td>
<td>1283</td>
<td>22</td>
</tr>
<tr>
<td>KP_{FE}</td>
<td>false</td>
<td>2375</td>
<td>30</td>
</tr>
<tr>
<td>KR_{FE}</td>
<td>true</td>
<td>2450</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 7.1: Execution statistics for verifications and interference checks of aspects E and F. ‘Model size’ refers to the number of BDD nodes generated.

Verification results described above and some statistics on running the example are summarized in Table 7.1. After obtaining the results, error analysis, as described in Section 4.2.4, 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 \textit{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. This incompleteness of our method arises from the incompleteness of aspect specifications, as these specifications do not necessarily express the full functionality of the aspects. 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.

### 7.2.2 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 7.2), in spite of the fact 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
<table>
<thead>
<tr>
<th>Verification task</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E vs. B</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$KP_{EB}$</td>
<td>true</td>
<td>no need to check: $P_B$ is true</td>
</tr>
<tr>
<td>$KR_{EB}$</td>
<td>false</td>
<td>invariant $psw_send \rightarrow encrypted_psw$ violated, and weaving fails</td>
</tr>
<tr>
<td>$KP_{BE}$</td>
<td>false</td>
<td>a state where a password is sent not from the login screen can now be reached</td>
</tr>
<tr>
<td>$KR_{BE}$</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td><strong>E vs. M</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$KP_{EM}$</td>
<td>true</td>
<td>no need to check: $P_M$ is true</td>
</tr>
<tr>
<td>$KR_{EM}$</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>$KP_{ME}$</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>$KR_{ME}$</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td><strong>B vs. M</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$KP_{BM}$</td>
<td>true</td>
<td>no need to check: $P_M$ is true</td>
</tr>
<tr>
<td>$KR_{BM}$</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>$KP_{MB}$</td>
<td>true</td>
<td>no need to check: $P_B$ is true</td>
</tr>
<tr>
<td>$KR_{MB}$</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Pairwise interference checks results

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 when 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.

### 7.2.3 Interference Detection in the Presence of Shared Join-Points

Refinement of aspect specifications makes it possible to detect and classify several cases of interference among the aspects A, E and L, as shown in Figure 7.2. A cell in the table corresponding to aspects pair $< M; N >$ can have one of the following values:
• “—” means that there is no interference when executing first M and then N at shared join-points in any appropriate system;

• “X” - if the check is irrelevant, here we do not check interference among the aspect and itself.

• Otherwise, there is interference among the two aspects if M is executed before N at shared join-points, and the cause of the interference is written in the cell, according to the classification from Section 5.1: “CB” stands for Change Before, “CA” - for Change After, “IB” - for Invalidation Before, and “IA” - for Invalidation After.

\[
\begin{array}{cccccc}
\text{second} & E & A & L1 & L2 & L3 & L4 \\
\text{first} \\
E & X & --- & CB & --- & --- & CB \\
A & --- & X & --- & --- & --- & IB \\
L1 & --- & IA & X & --- & --- & \\
L2 & CA & IA & --- & X & --- & --- \\
L3 & --- & IA & --- & --- & X & --- \\
L4 & --- & --- & --- & --- & --- & X \\
\end{array}
\]

Figure 7.2: Interference checks summary.

For example, the cell \(< E, L_1 >\) is marked by \(CB\), meaning that the Encryption aspect, if woven first, invalidates the assumption of the first variant of Logging, and that the violated part of the assumption is related to the “change before” case from Section 5.1: changing parameter values between arrival and actual join-points of \(L_1\). And the cell \(< L_1, A >\) is marked by \(IA\) as the Authorization aspect, when woven after the first variant of Logging, invalidates the part of Logging specification related to the “Invalidation After” case from Section 5.1: removing a join-point of the aspect after its advice has already been applied. Note that we might want to add more than one variant of Logging to a system, and there is no interference among them, provided their log files are different (then instead of one \(in\_log\) predicate we would have four: \(in\_log_1 \ldots in\_log_4\).

91
Chapter 8

Conclusions

When given a library of aspects, three questions need to be addressed: First of all, the semantics of each aspect should be captured by a precise specification. Then every individual aspect should be checked for correctness with respect to its specification, i.e., that it behaves correctly when woven separately into a suitable base system. Last, it should be checked that the desired effect of one aspect is not invalidated as a result of weaving additional aspects from the library into the same base system. Our contributions lie in all the three areas.

We extend the notion of assume-guarantee specification to aspects of the most general category - strongly invasive aspects. We then show that strongly invasive aspects can be shown correct relative to their specification, independently of a particular base system. In this work for the first time we refer to properties of the unreachable part of the base system. Describing and checking these properties is reasonable because the possible transitions of the base are considered bottom up, independently of the initial states, thus generating the unreachable part of the base system as a byproduct of model checking. Strongly invasive aspects typically extend the functionality of the base system to situations not originally covered. The examples seen in Chapter 3, of a semaphore with negative values, and of aspects to give bonus points beyond the normal range, are typical. Often some invariants true in the base system alone will no longer hold after weaving such aspects, but other invariants will continue to hold, and are essential to the correctness of the woven system.

Now individual aspect verification is also possible for aspects with full
past-LTL pointcuts, and aspects that might have shared join-points.

Additional analysis of aspect semantics and possible interference in a problematic case when several aspects might share join-points resulted in an interactive semi-automatic procedure for aspect specification refinement. This is an example of a delicate situation in which the user possesses some important information, but is unaware of the fact that this information might be useful for a proof. The questions asked and the results of formal verification should help the user understand the fine points of such interactions, and how they could affect the correctness of their aspect systems.

We have also defined semantic interference among aspects relative to their assume-guarantee specifications and shown a modular and effective way to detect interference or prove interference freedom of any subset of aspects in a library. The interference-detection procedure requires only pairwise off-line checks of the aspects.

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 choose 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.

All the verification methods presented in our work are modular, and thus have an advantage over a straightforward non-modular verification of a woven system: the possibility of reuse without proof. There are two types of such reuse we see, both of which are demonstrated by the aspect described in Section 3.3 and the aspect library in Chapter 7. One case is when one and the same aspect (or even a library of aspects) is applicable to different base systems. Then the verification of every individual advice versus the
assume-guarantee specification is performed only once, and in order to be able to apply the aspect to a given base system we need only to perform the base system verification described in Section 3.2.4. In the same way, an interference freedom check of the library of aspects is also done only once, and if (a subset of) a library of aspects is shown interference-free, all its aspects can be woven without additional checks into any base system satisfying their assumptions. Another case is when (a subset of) a library of aspects is given, where all the aspects are built for the same purpose (like defining some action policy, e.g. Logging variants from Section 7.1.1) and have a common assumption \((P, U)\) about the base system. Then if we have a base system that satisfies the above assumptions, we can change the policy defined in this system at any time, by applying different aspects from the library - one at a time, of course - without any further checks. An additional advantage of modular verification is that the models analyzed are much smaller than a typical woven system. This is important for the verification procedures, as when model-checking is used, the size of the verified system and of the specification strongly affects the verification time, and sometimes, if the model verified is too large, the model-checker can even fail to provide any answer.

There are several directions for future work. One is to extend modular interference detection method to the case of simultaneous, rather than sequential, weaving. As can be seen from Section 4.2.5, the difficulty in this case is to verify aspects in an incremental and modular way even when the second-woven aspect can create or remove join-points matched by the first-woven one. Another direction for the future work is to complete the automatization of all the verification procedures presented in the thesis. And, finally, it is highly desirable to apply the verification methods above to a bigger real-life library of reusable aspects, such as, for example, [29].
Bibliography


נחתות רוחנייה אינטראקציה ביך
אספктив

אמילה כנ"
ניהוג רדויי אינטראקציה בטקסט

הנובר על מחקר

לשם مليים חלקי על הורישות למ бюджет המתואר
דוקטורتلكסופה

אמילה כח

הنشر שלג הנסחים - מכון טכנולוגי לישראל
השיק וניהול א. חיפר
נובמבר 2010

Technion - Computer Science Department - Ph.D. Thesis PHD-2011-03 - 2011
מה peça לבין פורמי סרטנים עם המחזור החוצה המחבר.

בפרוגי תורניות מקירות בטכניון של פורמי סרטנים עם המחזור החוצה המключа
לאורד מאורד

AOSD-EUROPE Network of Excellence - על מחקרה
הכפשו זה יודע במרחבי העולם.
weaving
advice
AspectJ
security
debugging
base system
woven system
joining points
weaving
design
aspects
aspects
מרובים שחקנים ממיצパート בסים Khách. כדי לענות על שאלה אלה, דרישה: 

• המקרה הפוסטפילט של הספנטק של אספסיט בדד של אינטראקציה
  • בלתי ממקדת ביא אספסיט
  • מפרטים פורמליאים של אספסיט

• התלים אימוץ התוכנה של אספסיט בדד
• התלויות אימוץ של אינטראקציה בין אנטרפסיטים ממקדת ביא אספסיט בשטח

עלובה deported של האימוץ של התוכנה של אספסיט

כדיỗור הגרדום והרגולציה הרפואית של אספסיט, ויתור כל מבר_tokenize התוכנה של האספסיט הדבקים נגזר לשלב אספסיט, והתער ה饧יל אמור עד לאספסיט. כך, רצי שรับผิดชอบהכל אספסיטים זה מקודם מחלק-חלק בוגרים חסימה. ספנטק מצירה והכותב. רחב כי כל התוכנה זו אמור עד לאספסיט. שהאימוץ האספסיט בחוס מפטר את האימוץ של התוכנה של אספסיט, אך מבר_tokenize התוכנה של אספסיט, לבר, המבר_tokenize התוכנה של אספסיטים בודק רציון ר_peer מתוכנה של כל האספסיטים בוחס ל车上 başlat שלל. לבר,麦克 מבר_tokenize התוכנה של אספסיטים, מימר כי במעריך הבסיס ממקדת את הגרדום של אספסיטים. שてしまいました התוכנה של האספסיטים. בין כל אספסיטים ממקדת את תהליך אימוץ האספסיטים של אספסיטים יתוג המבר_tokenize התוכנה של אספסיטים, אך מבר_tokenize התוכנה של האספסיטים. משלי, שיאם במעריך הבסיס ממקדת את תהליך אימוץ האספסיטים. משלי, שיאם במעריך הבסיס ממקדת את תהליך אימוץ האספסיטים. משלי, שיאם במעריך הבסיס ממקדת את תהליך אימוץ האספסיטים. משלי, שיאם במעריך הבסיס ממקדת את תהליך אימוץ האספסיטים.
systems) - MAVEN - strongly invasive systems

systems)

technology for large-scale distributed computing systems)

systems)
Technion - Computer Science Department - Ph.D. Thesis  PHD-2011-03 - 2011

advice strongly invasive

Aspects of Caela.

clicks that are not mediated by another entity or person.

Technion - Computer Science Department - Ph.D. Thesis  PHD-2011-03 - 2011
 possono_multiple_sections nella_ricerca_molto_grande

anche_è_importantiare multiple_strumenti_per_riflettiamo_e_lasciare_shel

un_evidente_informati_ando_scritto_far_d centered_ando

 keyof_anedi_molti_e_ando_scritto_far_d far_ando

حا