Detecting Concern Interactions in Aspect-Oriented Designs

By
Pouria Shaker, B. Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Engineering

Memorial University of Newfoundland

Master of Engineering (2006) Memorial University of Newfoundland
(Electrical and Computer Engineering) St. John's, Newfoundland

Title: Detecting Concern Interactions in Aspect-Oriented Designs

Author: Pouria Shaker, B. Eng. (Memorial University of Newfoundland)

Supervisor: Dr. Dennis K. Peters

Number of pages: x, 113

Abstract

Aspect-Oriented Software Development (AOSD) is an emerging paradigm that addresses the limitation of Object-Oriented (OO) technology in localizing crosscutting concerns (e.g. logging, tracing, etc.) by introducing a new modularization mechanism: the aspect. Aspects localize the behaviour of crosscutting concerns (called advice) and specify points in the structure or execution trace of the core system (called join points) where advice applies. A weaving mechanism interleaves the execution of the aspects and the core. The behaviour of an Aspect-Oriented (AO) system is the woven behaviour of the aspects and the core; this woven behaviour may reveal conflicts in the goals of system concerns (core or crosscutting): such conflicts are called concern interactions. In this thesis, we present a process for detecting concern interactions in AO designs expressed in the UML and our weaving rule specification language (WRL). The process consists of two tasks: 1) a light-weight syntactic analysis of the AO model to reveal advice overlaps (e.g. instances where multiple advice applies to the same join point) as potential sources of interaction and 2) verification of desired

model properties before and after weaving to confirm/reject findings of task 1 and/or to reveal new interactions. At the heart of task 2 is a weaving process that maps an unwoven AO model to a behaviourally equivalent woven OO model.

Acknowledgements

I sincerely thank my supervisor, Dr. Dennis K. Peters for accepting me as a student, for his patience in and availability for discussing cumbersome details of my thesis, for his constant encouragements in my hours of despair, and for supporting my attendance in the *Aspects, Dependencies, and Interactions* (ADI) workshop at the 2006 edition of the *European Conference on Object-Oriented Programming* (ECOOP), as well as the 2005 and 2006 editions of the *Newfoundland Electrical and Computer Engineering Conference* (NECEC) to present our work.

The financial support received from the Natural Sciences and Engineering Research Council (NSERC), the Faculty of Engineering and Applied Science, and the School of Graduate Studies (SGS) of Memorial University of Newfoundland is gratefully acknowledged.

I am also grateful to Dr. Theodore S. Norvell for his insightful comments on the presentation of the weaving process, and Dr. Ramachadran Venkatesan, Associate Dean of Graduate Studies and Research, and Ms. Moya Crocker, Secretary to the

Associate Dean of Graduate Studies and Research, for their superb administrative support.

Last but not least, I wish to thank my family and friends for their love and support.

Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Description	Page (definition)
SOC	Separation of Concerns	1
OO	Object-Oriented	2
POP	Post-Object Programming	4
AOP	Aspect-Oriented Programming	4
AO	Aspect-Oriented	7
AOSD	Aspect-Oriented Software Development	10
CW	Call Waiting	11
CFB	Call Forward when Busy	11
CI	Concurrency Interceptor	12
PI	Priority Interceptor	12
UML	Unified Modeling Language	13
WRL	Weaving Rule Language	13
EAOP	Event-based AOP	16
FSM	Finite State Machine	19
BISL	Behavioural Interface Specification Language	25
JML	Java Modeling Language	26
LTS	Labeled Transition System	26
RTC	Run To Completion	34
WP1	Unoptimized Weaving Process	46
WP2	Optimized Weaving Process	60
WP2.1	WP2 without Concurrent Regions	67
FITEL	Feature Interactions in Telephony Systems case study	72
PRL	Price Reduction Limit	89
ECOMM	Electronic Commerce case study	89

Chapter 1

Introduction

1.1 The Aspect-Oriented Paradigm

1.1.1 Objects and Separation of Concerns

Over the years, computer hardware and software have evolved hand in hand. In the early days, due to hardware limitations, the problems solved by computers were simple and so was the software written to solve them. Demands for using computers to solve more complex problems led to advancements in hardware technology; software technology grew as a result to support the complex software solutions required for such problems. Traditional engineering disciplines manage the complexity of systems by *separation of concerns* (SOC); that is, identifying the system's *concerns* and dealing with each concern separately. With ideal SOC one can develop, test, and modify

system concerns in isolation and evolve systems to handle new concerns without changing their existing parts. In 1972, Paras called for the application of SOC to software development to cope with the increasing complexity of software systems, and suggested that ideal SOC can be approached through the technique of modularization [36]. Over the years, programming paradigms have emerged to help developers achieve better SOC by providing better modularization mechanisms. The Object-Oriented (OO) paradigm is currently the most popular; its primary unit of modularity, the class, improves SOC by grouping together data and behaviour related to a single concern; however not all concerns of a system can be simultaneously localized in classes. Consider the example (from [12]) of a typical OO model for a simplistic figure editing program shown in Figure 1.1. The concerns of representing the display screen and the figures, points, and lines on the screen (i.e., the *core* concerns) are localized by the concrete classes `Display`, `Figure`, `Point`, and `Line` respectively. Now consider the concern of updating the display screen each time points or lines move. This concern cannot be localized in a single module in this model. Its implementation cross-cuts the `Point` and `Line` modules as invocations of `Display.update()` in each of the modifier methods of `Point` and `Line`. In this model, display updating is a *crosscutting* concern. What if we try a different modularization that localizes the display updating concern? We will sadly discover that this will leave other concerns scattered across the new model. The crosscutting nature of concerns is an inherent property of many real

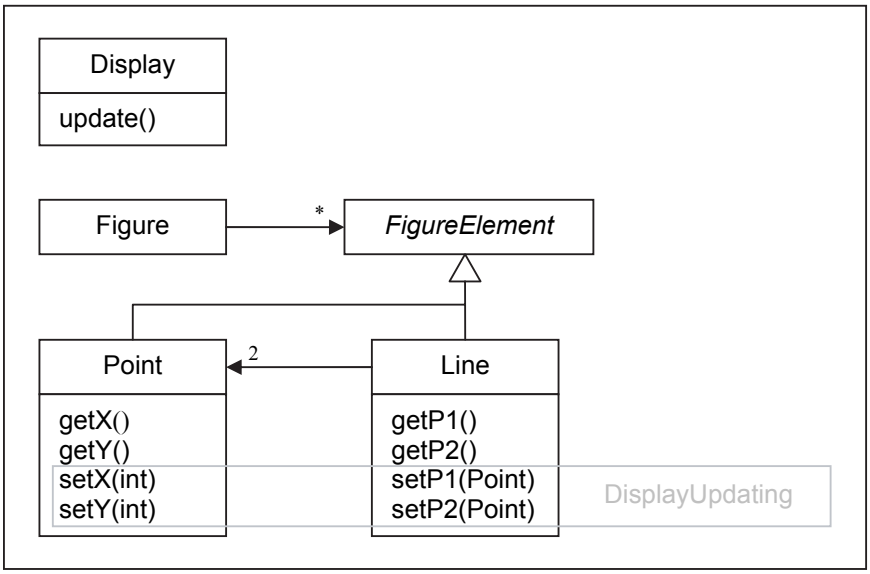

Figure 1.1: Crosscutting in OO models

problems and OO technology falls short in localizing all concerns in such problems.

As shown in Figure 1.2 (adopted from [24]), the concern space of many problems is *multi-dimensional*. An OO system is modularized across a single dimension. All concerns along this dimension are neatly localized in the OO model, while the remaining concerns crosscut the model. This is the result of mapping a multi-dimensional concern space onto a *single-dimensional* implementation space.

The inability of OO technology to simultaneously localize orthogonal concerns has its consequences: crosscutting concerns are implemented in several modules (*scattering*) and a single module implements more than one concern (*tangling*). These are signs of poor modularity: scattering leads to poor traceability from crosscutting concerns to their implementation, and tangling hinders ease of module implementation

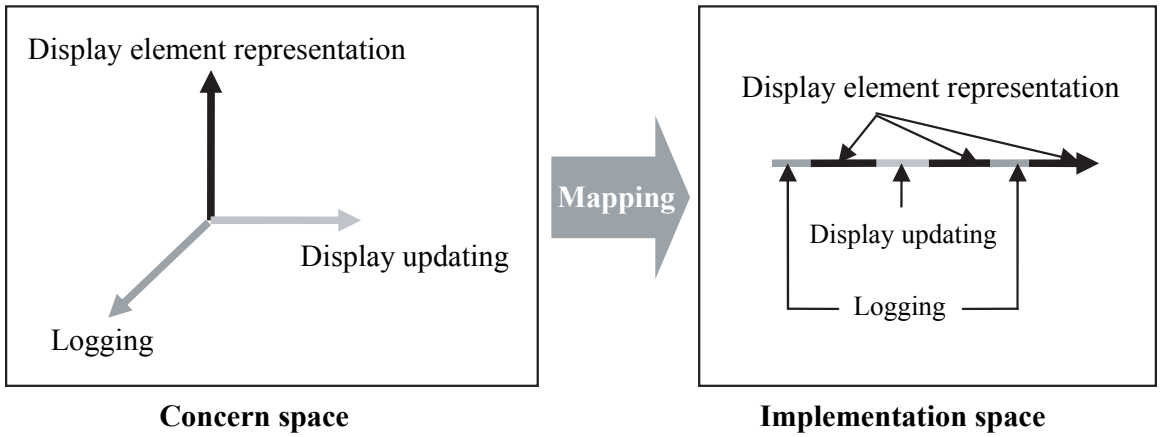

Figure 1.2: Mapping a multi-dimensional concern space onto a single-dimensional implementation space

(one has to focus on multiple concerns while implementing a module), comprehension, and reuse (the implementation of one concern comes with the baggage of other concerns). It also becomes hard to evolve the system since implementing an additional crosscutting concern involves modifying multiple modules.

1.1.2 Aspect-Oriented Programming

Several post-object programming (POP) technologies emerged to address the limitation of OO technology in achieving SOC across more than one dimension. These include adaptive methods [26], subject-oriented programming [44], composition filters [3], and aspect-oriented programming [21]. These related research paths converged under the title of *aspect-oriented programming* (AOP).

Despite ongoing and productive dialogue amongst the AOP community, a com-

mon consensus on what constitutes an AOP approach is yet to be reached (though significant efforts have been made including [29], [31], and [14]). Perhaps the most widely cited endeavour to characterize AOP is that of Filman [14]: that AOP is *quantification* and *obliviousness*. Quantification means that programs can include *quantified statements* (i.e. statements that apply to more than one place) of the form

> In programs P, whenever condition C arises, perform action A.

Obliviousness means that authors of a program P need not be aware of quantified statements that reference them. How do quantified statements help? Figure 1.3 illustrates how the display updating concern from the figure editing example of Section 1.1.1 can be localized in a quantified statement (another quantified statement could localize the logging concern). Notice how the authors of the `Point` and `Line` classes can be oblivious of the display updating concern (or other cross-cutting concerns such as logging) and focus on implementing the concerns of representing points and lines. In general given an N-dimensional concern space and an M-dimensional implementation space where M ¡ N, crosscutting concerns can be localized in quantified statements in an AOP system (this supports the notion that AOP does not replace existing technologies, rather it complements them); this improves modularity with the following implications:

- *Improved traceability*: Crosscutting concerns can be easily traced to quantified statements.

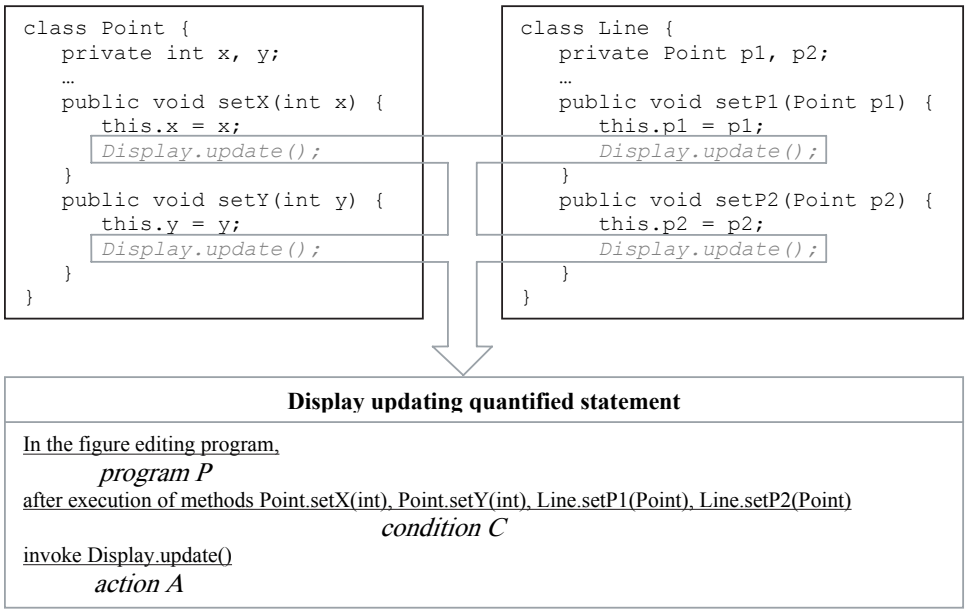

Figure 1.3: Use of quantified statements in the figure editing example

- *Ease of implementation and comprehension*: Authors/readers of modules can focus on implementing/understanding one concern and can be oblivious of crosscutting concerns.

- *Module reusability*: Modules implement a single concern and do not come with the baggage of other concern implementations.

- *Improved evolvability*: Adding a crosscutting concern is simply a matter of adding a quantified statement.

According to [14], to implement an AOP language (i.e., a language that allows quantified statements over oblivious programs) one must consider three issues:

- *Quantification*: What conditions can we use in quantified statements? In other

words, to what points in the execution of programs can actions be tied? Two broad types are:

  - Points that can be specified by elements of the static structure of programs (e.g. method calls which can be specified by method signatures)
  - Points that depend on run-time behaviour (e.g. size of the call stack)

- *Interface*: How do quantified statements communicate with programs and with other quantified statements?

- *Weaving*: What mechanism interleaves the execution of actions in quantified statements with the execution of affected programs?

AspectJ [19] is a general purpose aspect-oriented (AO) extension to Java developed by the AOP group at Xerox Palo Alto Research Center (PARC) and is perhaps the most popular existing AOP language. AspectJ allows writing quantified statements over conventional Java programs. Quantified statements are specified by class-like constructs called *aspects* (note that the term aspect-oriented programming was coined by Gregor Kiczales of Xerox PARC). Let us see how AspectJ addresses the three implementation issues listed above:

- *Quantification*: Conditions are specified by *pointcuts*, which are expressions that match a set of points in the execution of programs (i.e., *join points*). The kinds of join points supported (i.e., the *join point model*) include method or

constructor calls and executions, advice executions, static class initializations, object or aspect initializations, field read or write accesses, and exception handler executions.

- *Interface*: Aspect actions are specified by method like constructs called *advice*, which can be specified to execute before, after, or around join points. Aspects can gain contextual information from join points and use it in advice; this is done using parameterized pointcut expressions. Additionally, aspects can introduce fields and methods into types in the core through the *inter-type declaration* mechanism.

- *Weaving*: the AspectJ compiler (ajc), combines core and aspect source files and jar files into woven class files or jar files.

AspectJ terminology (e.g. aspect, join point, join point model, and advice) is widely used in AO literature, and will also be adopted in this document. Figure 1.4 shows an AspectJ aspect written for the display updating concern.

Other AOP languages such as the DJ library [26], Hyper/J [44], and composition filters [3] use different approaches to address the three implementation issues. The interested reader is referred to the cited sources for details.

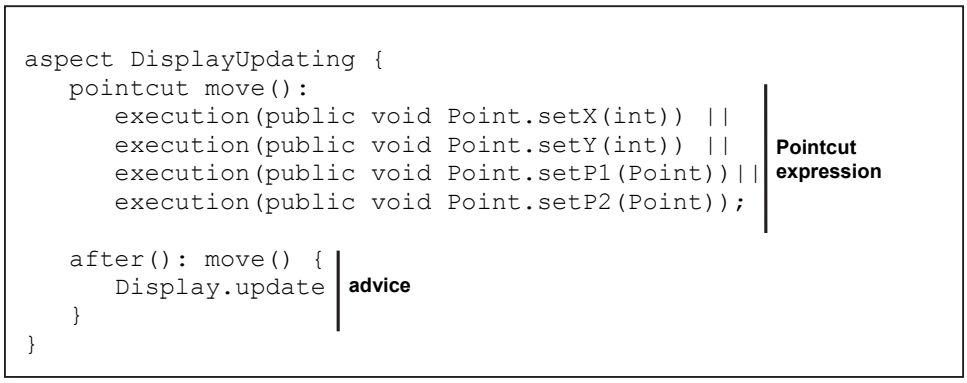

Figure 1.4: Display updating aspect

1.1.3 Aspect-Oriented Software Development

Software development has evolved from a programming activity to a full-blown engineering process. Modern software engineering constructs systems using processes that progressively refine higher-level abstractions of the system to lower-level abstractions starting from requirements and stopping at executable code. Preserving two important properties across this refinement process helps a great deal in producing high-quality software: modularity and traceability. Both modularity and traceability are crucial in managing change in systems. When the system changes at a given level of abstraction, modularity ensures that the change is localized, and traceability ensures that the change can be propagated naturally and easily to other levels of abstraction.

In Section 1.1.2, AOP was described as a technique that improves modularity at the code level. The benefit of applying the AO paradigm to earlier stages of the software development cycle is two-fold: first it ensures improved modularity at all stages

of the development process; secondly, preserving the notion of aspects throughout the development process ensures traceability. These ideas launched the field of *aspect-oriented software development* (AOSD) with an active research community. As stated in [13], the same way that AOP extends conventional programming technology, AOSD extends conventional software development practices. An excellent survey of research aimed at applying AO techniques to various stages of the development process including requirements engineering, specification, design, implementation, and evolution is given in [4].

1.2 Concern Interactions

By untangling cross-cutting behaviour from core behaviour, the AO paradigm makes it easier to reason about individual concern behaviour. Reasoning about overall system behaviour however, becomes a challenge as it requires examining the woven behaviour of the core and the aspects, which may or may not be explicitly available to the developer in a comprehensible form (this depends on the workings of the weaving mechanism). This situation can give rise to unanticipated anomalies in the behaviour of the woven system. The desired properties of the woven behaviour of two concerns (possibly compound, i.e. the result of weaving two or more primitive concerns) are (1) existing critical correctness properties of the behaviour of each individual concern and (2) new correctness properties of the woven system; if this set of properties is

inconsistent, we say that two or more of the concerns involved undesirably interact. In (1) we say *critical* correctness properties, to distinguish between desired and undesired interactions. The very purpose of weaving an additional concern may be to violate existing properties of constituent concerns in favor of achieving new properties for the woven system. In the remainder of this thesis the term *interaction* will be used to mean undesired interaction. A simple example of concern interactions from [10] is the interaction between logging and encryption aspects applied to some core system. The encryption aspect encrypts the content of messages passed within the core, while the logging aspect logs the messages for debugging purposes. If logging precedes encryption, encryption is compromised by a plain log file; and if encryption precedes logging, logging is compromised by an encrypted log file that is not very useful for debugging. More sophisticated instances of concern interactions have been identified in various domains including the following:

- *Telephony*: In modern telephony systems, users can subscribe to various *features* on top of their basic call service. Features subscribed to by one or more users may interact. As an example, suppose a user subscribed to *call waiting* (CW) and *call forward when busy* (CFB) is engaged in a call and receives a further call. If the call is forwarded due to CFB, CW is compromised and vice versa. Feature interactions in telephony systems has been an active research area for many years. A survey of the state of the art in this area is presented in [7].

- *Email*: Basic email can also be improved by various features (e.g. *spam filter*, *auto responder*, etc.) that may interact. Suppose an email service equipped with *decryption* and *forwarding* features receives an encrypted message. In *decryption* applies before *forwarding*, then a clear email message is sent over the internet compromising the *encryption/decryption* features. Such interactions have been studied in [16].

- *Middleware*: Middleware is software that connects software components, and supports the development and operation of these components by providing generic (e.g. security, messaging, etc.) and/or domain specific services. As explained in [27], these services may interact. For example, consider *interceptor* services in a J2EE compliant application server. Interceptors are arranged in a pipeline and process incoming requests from end components in order. Suppose the *concurrency interceptor* (CI), which allocates a thread from a limited pool to each request, comes before the *priority interceptor* (PI), which schedules requests based on their priority. If CI has no threads left for a new high priority request, PI is compromised.

- *Multimedia*: Interactions between Internet-based and multimedia/mobile services have been identified in [5]. For example, mechanisms for *power adaption* and *network bandwidth adaption* in a mobile device interact: if the device is running low on power, power adaptation instructs applications using network

bandwidth to stop; but as a result, network bandwidth adaptation instructs applications to make use of the bandwidth that has now become available.

1.3 Research Objective and Overview

The sooner an error is found in the software development process the easier it is to fix. In this document we present a process for detecting concern interactions at the design stage (see Figure 1.5). The process assumes AO designs expressed in the Unified Modeling Language (UML) [39] and our weaving rule specification language (WRL). Here, the data and behaviour of concerns are modeled separately using UML class and statechart diagrams, and rules for weaving concern behaviour are specified in WRL. WRL defines a join point model on UML statecharts and supports the following:

- Before and after advice (before advice can conditionally consume the advised join point)
- Assignment of aspect instances to core instances
- Aspects of aspects
- Aspect composition by a precedence operator on advice

The process consists of two tasks:

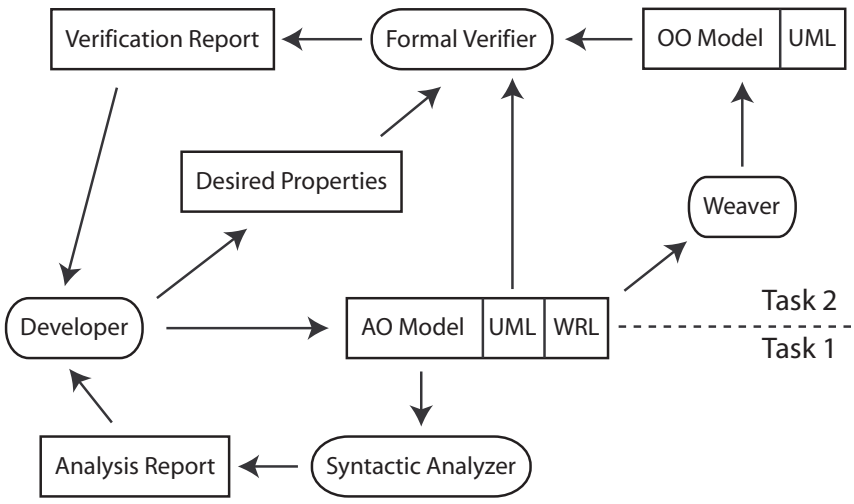

Figure 1.5: Design-level concern interaction detection process overview

- *Task 1:* The AO model is syntactically analyzed to reveal advice overlaps; e.g. instances where multiple advice is applied to the same join point. Such overlaps can be the source of interactions and can easily be overlooked by the developer. Examination of the analysis report by the developer may lead to revisions of the AO model.

- *Task 2:* A weaving process is applied to the AO model the output of which is a woven OO model expressed in the UML. Existing UML verification techniques (such as [41], [34], and [33]) are applied before (on the UML component of the AO model) and after weaving against desired properties specified by the developer to detect interactions as defined in Section 1.2. The verification report may reveal indirect interactions not exposed by task 1, and may be used to determine whether advice overlaps revealed by task 1 do indeed correspond to interactions.

It should be noted that formal verification of UML is still a research topic; however, we view it as an available technology and will reasonably assume that such tools will mature in the not too distant future. In our work, we used IFx [33] to formally verify UML models. IFx extends the IF toolset, a set of tools for model-checking and simulating models of communicating extended timed automata expressed in the *IF* language. In IF, temporal properties are expressed using *observer* automata. Observers monitor the execution of the model and react to state changes and events (e.g. signal receptions). An observer state can be designated as an *error* state, allowing the specification of *safety* properties. IFx defines a UML observer formalism, where an observer is modeled as a class (error states in the class statechart are labeled with stereotypes) and maps UML models and observers to IF.

The remainder of this document is organized as follows. Chapter 2 gives a survey of related work. Chapter 3 describes the details of the process: it describes our AO modeling language, the syntactic analysis of the AO model, and the process of weaving the AO model into a behaviourally equivalent OO model. Chapter 4 applies the process to two case studies. Chapter 5 evaluates the process empirically (based on the case studies) and analytically. Finally, Chapter 6 presents the conclusion and directions for future work.

Chapter 2

Related Work

In this chapter we present a survey of the state of the art in the area of concern interactions. Section 2.1 briefly describes research on a well-known and extensively studied instance of concern interactions: feature interactions in telephony systems, which we mentioned in Section 1.2. Research on concern interactions in *generic* AO systems is more scarce: in Section 2.2 through Section 2.5 we present (what we deem a reasonable coverage of) relevant published work in AO literature. Section 2.2 presents research on the classification of concern interactions. Section 2.3 describes *event-based AOP* (EAOP) a widely cited framework for AOP, with support for the detection and resolution of aspect interactions. Section 2.4 lists selected research on the application of formal specification and verification to AO systems. As we argued in Section 1.3, formal methods can be used to detect concern interactions.

Section 2.5 lists research on methods that enable *modular reasoning* of AO systems. Modular reasoning eliminates the need for analyzing the entire system to understand the effect of applying an aspect to the core. Such an understanding will aid developers in foreseeing and resolving concern interactions.

2.1 Feature Interactions

In [7] research on feature interactions in telephony systems has been categorized into three trends:

- *Software engineering:* It is argued that the creation of features is largely a software development task. Features are software entities that are complex, real-time, prone to frequent change, and must exhibit a high level of reliability. It is only natural to apply and *adapt* software engineering techniques used for development of software with similar attributes to the development of features. Such techniques include the use process models and methods for various phases of the development cycle (e.g. specification, design, testing, deployment, and maintenance). Software engineering can lessen the probability of unanticipated feature interactions *indirectly* by adding rigour, structure, and predictability to the feature creation process, and can *directly* aid feature interaction detection, resolution, and avoidance, with the design of process models with phases dedi-

cated to the application of methods, notations, or techniques used elsewhere in software engineering to the feature interaction problem.

- *Formal methods:* Formal description, modeling, and reasoning techniques (including process algebras, various flavours of automata, petri-nets, SDL, Promela, Z, and LOTOS) have been used to both detect unanticipated feature interactions and to validate expected ones at the *specification* level (i.e. independent of the implementation), though the latter use applies to most published work. The (widely known) benefits of formal methods include having an unambiguous documentation for features and the ability to perform automated analyses. The approach for using formal methods is either of the following:
  - Modeling features and the basic service with abstract *properties*, and defining interactions as inconsistencies in the properties
  - Modeling features and the basic service with behavioural models and properties on the models, and defining interactions as when individual models satisfy their properties, but the combined model fails to satisfy the conjoined properties

Our *definition* of interactions in Section 1.2 resembles the former approach, while task 2 of our process (defined in Section 1.3) applies the latter. The operation of combining models in the latter approach is tailored to a known set

of features and basic call service, while we define a weaving process for generic AO systems.

- *Online techniques:* Here, interactions are detected (and often resolved) at *run-time*. The benefits of using online techniques include operating on the real system and not its model, and support for detecting interactions between features developed by multiple vendors (where specification level information is not available) and features in legacy systems (where detailed documentation for features is not available). Online approaches require information about features that can be collected a-priori at design-time or during run-time. Also the control of monitoring feature communications can be localized in a *feature manager* or distributed amongst features. Our approach is clearly an *offline* technique.

Using AO technology to *detect* feature interactions has also been studied [30], where AspectJ [19] is used to encode the control software as a finite state machine (FSM) and features as aspects that change the FSM (or core). *Program slicing* [47] is used to identify the part (*slice*) of the core affected by each aspect. Overlaps in aspect slices are reported as interactions between features encoded by the aspects. A program slice, is an executable portion of the program that manipulates variables referenced by a set of program points called the *slicing criterion*. To extract the core slice affected by an AspectJ aspect, pointcut expressions are used as the slicing

criterion.

2.2 Classification of Concern Interactions

An approach for classifying and documenting aspect interactions is presented in [40]. It is argued that classification helps identify common patterns of interaction and their response type, and documentation makes interactions explicit providing useful knowledge that can used throughout the system life cycle. The classification defines four types of interactions:

- *Mutual exclusion:* An *undesired* interaction that occurs when two aspects implement mutually exclusive concerns such as alternative algorithms or policies. For this type of interaction, no mediation is possible and only one of the aspects can be used

- *Dependency:* An *undesired* interaction that occurs when the correct operation of aspect A depends on the presence and an expected mode of operation of aspect B (e.g. an authorization aspect that depends on an authentication aspect), and aspect B is not present or does not operate as expected

- *Reinforcement:* A *desired* interaction that occurs when one aspect positively influences the correct operation of another aspect (e.g. extends its functionality). Suppose that in an auction system we have authorization aspect. Adding an

aspect that monitors the location of users allows us to extend the authorization aspect to implement a more sophisticated authorization procedure that takes the user's location into account.

- *Conflict:* An *undesired* interaction that occurs when aspects *semantically* interfere; that is, when one aspect works correctly alone, but fails to do so when composed with other aspects. The logging vs. security example of Section 1.2 illustrates this kind of interaction. Conflict interactions can often be resolved by mediation.

Our definition of concern interactions is closest to the *conflict* category, though we also consider core/aspect interactions. Mutual exclusion and dependency categories as defined in [40] deal with dependent aspects. Such interactions can be captured at a higher level (e.g. the requirements phase). Nevertheless our pre/post weaving verification approach can potentially be used to capture such interactions as well.

An analysis of AO programs that classifies interactions between aspect advice and core methods is presented in [38] (interactions between aspects are not considered). Advice can interact with a method *directly* by augmenting, narrowing, or replacing its execution, or *indirectly* by using object fields also used by the method. Direct interactions can be found by task 1 of our process: before advice that does not consume, and after advice are augmenting advice; before advice that may consume is narrowing advice; and before advice that always consumes is replacement advice (see

Section 3.2 for a definition of consumes in before advice). Indirect interactions can be found by task 2.

2.3 Event-Based AOP

EAOP is perhaps the most explicit treatment of the problem of detecting and resolving aspect interactions in AO literature. EAOP defines a formal framework for AOP: the core is modeled by its *execution trace*, i.e. a sequence of join points emitted by the core in the course of its execution. Primitive aspects ($C \triangleright I$) are defined by a *crosscut* (C), which is a regular expression that matches a sequence of events in the execution trace, and an *insert* (I), which is the action to be performed when the crosscut is matched. Aspects can be composed by several operators:

- *Repetition ($\mu a.A$):* Repeats the behaviour of an aspect after it matches a join point sequence, where A is an aspect and a is the repetition variable.
- *Sequence ($A_1 - C \rightarrow A_2$):* Behaves like A_1 until C matches a join point sequence, at which point it behaves like A_2.
- *Choice ($A_1 \Box A_2$):* Behaves like the first aspect to match a join point sequence (e.g. if A_1 matches a join point first, the choice aspect behaves like A_1 and A_2 behaviour is dropped). Behaves like A_1 if both match a join point sequence.

Compound aspects are essentially *state machines* that evolve from one aspect to another in response to join points.

Weaving is implemented by a *monitor* that observes the execution trace and propagates join points one at a time to a *parallel* composition of aspects $(A_1||\ldots||A_n)$. In response to each join point, the parallel composition evolves into the parallel composition of each constituent aspect *evolved* in response to the join point. The parallel composition of two aspects can be *adapted* to, for instance:

- Propagate the join point to the first aspect and then to the second aspect
- Propagate the join point to the aspects in an arbitrary order
- Propagate the join point to the first aspect, and then only if the first aspect did (not) match a crosscut, to the second aspect

In EAOP, two aspects are said to interact when they match the same join point. A static analysis is introduced to detect such interactions. In Section 3.3 we use the term *advice overlap* for this definition, and explain that it may fail to capture important interactions in a system. We illustrate via case studies in Chapter 4 how task 2 of our process can detect such interactions. Composition operators, including (adapted) parallel composition, serve as linguistic support to resolve interactions (i.e. advice overlaps). In Chapter 5 we point out that our approach falls short of EAOP in linguistic support for interaction resolution, due to less aspect composition operators.

Other features of EAOP include:

- *Aspects of aspects:* Aspects themselves can contribute join points to the execution trace; that is, they can be advised by other aspects.
- *Aspect variables:* Variables in compound aspects allow information sharing amongst constituent aspects.
- *Requirement aspects:* These are special aspects that specify conditions that the core must satisfy in order for a normal aspect to be applicable.

2.4 Applying Formal Methods to AO systems

The following summarizes publications we came across that apply formal methods to AO systems, and in some instances explicitly state the applicability of their approach to the detection of interactions. Additional references on the subject can be found in [4].

- A process-algebraic foundation for AOP is presented in [2].
- A technique to verify properties of AspectJ [19] *aspects* is presented in [6]. It is argued that it is sufficient to analyze the aspect itself and the portion of the core program that it affects. Program slicing [47] is used to compute this portion (slice) as in [30]. Core slices are used to build useful models using tools such

as Bandera [9] (a tool that extracts FSMs from Java source code), that can be used to prove properties such as absence of aspect interactions at the code level. In a related work [45], model extraction and model-checking is applied to *woven* AspectJ.

- The AOP language SuperJ is introduced in [42]. In SuperJ crosscutting concerns are implemented in *superimpositions*, which are collections of *generic aspects* and new (singleton) classes. Generic aspects do not reference program units (e.g. method names) of a particular core. Instead, they reference *parameters* that are later *bound* to a particular core. The new singleton classes provide services that are used in methods and advice of generic aspects. A preprocessor applies a superimposition to a particular core by turning generic aspects to concrete AspectJ aspects. This is done by binding program units of the core to parameters of generic aspects. Additionally, the new singleton classes are added to the bound program. A superimposition includes a specification of its applicability conditions to core programs, and desired properties of the bound program. This specification allows proofs on the correctness of superimpositions and the legality of combining them (i.e. detecting interactions) independent of a particular core.

- Pipa, a *behavioural interface specification language* (BISL) for AspectJ, is introduced in [48]. A BISL specification describes how to use a module by detailing

a module's interface (i.e. static information such as method signatures) and its behaviour from a client's point of view. BISLs are language dependent. Pipa is an extension to the Java modeling language (JML) [25], a BISL for Java, and enables the formal specification of AspectJ modules. A process for transforming an AspectJ program with its Pipa specification into a corresponding Java program with its JML specification is described in [48]. This allows the use of JML-based tools to formally verify properties of AspectJ programs.

While [6], [42], and [48] apply to AO systems at the source code level, the following research (like ours) targets AO systems at the design and specification level.

- In [32] AO systems are modeled using the *role modeling* approach [37], where a system concern is modeled by a set of *roles* that collaborate to address the concern. Each role represents an object (that can be involved in one or more concerns) and describes those properties of the object that are relevant to the concern. Weaving two concerns involves merging their role models by identifying roles in one concern with roles in the other and merging them into *fat* roles. Remaining roles are carried over to the woven model. Role models are expressed in Alloy [18], a formal object modeling language, and formal analyses are performed on the Alloy models.

- In [35] the core and aspects are modeled using *labeled transition systems* (LTS). The core LTS is stored in a flexible data structure, allowing aspect LTS to be

woven in at run-time. A run-time manager is responsible for the dynamic aspect integration, and also applies run-time model-checking to detect aspect/aspect and core/aspect interactions. The run-time model checking only checks for interactions in the current system execution and therefore does not suffer from the state-space explosion problem. Once an interaction is detected a combination of resolution strategies that use a-priori knowledge of interactions as well as generic resolution strategies are applied. The resolution strategies themselves are adaptive in that they can change based on the result of the detection and resolution.

- In [23] aspect advice and the core program are modeled as FSMs. A process is presented that takes the core FSM, pointcut designators (i.e. points where advice apply), and desired behavioural properties of the core that are to be satisfied before and after weaving, and automatically generates *interfaces* that advice applied to the core can be verified against. The interfaces describe the model-checker state, at states of the core that lead to and return from advice. Advice (authored possibly at a different time or place) can be verified against these interfaces in isolation from the core; the authors term this capability *modular advice verification*. This allows for the detection of core/aspect interactions without the need for the computationally expensive verification of the entire woven system.

Our work differs from these efforts in two respects: First, it uses a practitioner-friendly AO modeling language made up of a main-stream design language (the UML) and a simple (and intuitive) domain specific language (WRL). Second, the computationally expensive formal verification is preceded by a light-weight syntactic analysis. Our work particularly differs from [35] in that it is an *offline* process: all tasks (e.g. syntactic analysis, weaving, and formal verification) are performed at *design time*. In contrast, [35] presents an *online* process, where aspects can be woven and interactions detected and resolved at *run time*. Also, it differs from [23] in that it targets aspect/aspect interactions as well as core/aspect interactions.

2.5 Modular Reasoning on AO systems

Modular reasoning, as defined in [20], is the ability to reason about a module by examining its interface, implementation, and the interfaces of other modules referenced in its implementation. AOP enables modular reasoning on cross-cutting concerns implemented in aspect modules. It is argued, however, that due to the *obliviousness* criterion (see Section 1.1.2), AOP hinders modular reasoning on core modules advised by aspects, since fully understanding a core module requires examining all aspects that advise it, references to which are not present in the core's implementation (i.e. it requires a *global* system analysis). The following research aims at enabling modular reasoning in AO systems:

- In [20] it is argued that regardless of whether AOP is used or not, reasoning about cross-cutting concerns does indeed require a global analysis of the system due to the *scattering* and *tangling* phenomena (see Section 1.1.2). With AOP, once the *deployment configuration* is known, a single pass of global analysis is sufficient to construct *aspect-aware interfaces* for modules. In an AspectJ implementation, the aspect-aware interface of an aspect module includes signatures of advised join points, and that of a core module includes the core interface augmented with signatures of advice applied to the core. Once aspect-aware interfaces are constructed, modular reasoning becomes possible in the AO system.

- *Open modules* is introduced in [1] as a module system where a module's interface is made up of methods and *advisable* pointcuts. Details of when a pointcut is matched is hidden in the module's implementation. Here, aspects can only advise external calls to methods of a module's interface and pointcuts in a module's interface. It is argued that in AspectJ like languages, an aspect is tightly coupled with implementation details of a core module (such as method names and method implementation details). Open modules ensures that the dependency is restricted to a well-defined interface as in conventional module systems.

- In [8] it is proposed to divide aspects into two groups: *observers* and *assistants*.

Observers do not change the specification of core modules they advise and as such they preserve modular reasoning, without the need for core modules to explicitly reference them. Assistants on the other hand *can* change the specification of core modules, and therefore, modular reasoning only becomes possible when core modules explicitly reference assistants. A module is said to *accept assistance* when it lists assistants that advise *it* or modules that it uses.

- In [43], it is argued that the lack of constraints on the core program implied by obliviousness results in high coupling between aspects and the core. For example the change of a method name in the core can break many aspects that advise it (due to pointcuts). *Design rules* on AO programs is presented as an alternative to obliviousness. Design rules impose restrictions on 1) the kinds of exposed join points, 2) join point naming schemes, and 3) behaviour across join points (e.g. pre and post conditions for advice execution at join points). These restrictions imply an interface that aspects must adhere to. The first two restrictions imply that pointcut expressions, once written in compliance with the design rules, need not be changed, and the third restriction implies absence of core/aspect interactions.

In our approach we perform analyses to detect concern interactions, which is very different from the modular reasoning approach. For this reason no direct connection (other than the fact that both approaches are means to the same end) is described.

Chapter 3

Process

3.1 A Restricted UML Formalism

In this section we semi-formally describe a subset of the UML that is of interest in our process. The syntax of the UML subset of interest is expressed using sets, relations, and functions while the semantics of UML statecharts is expressed informally in English (presenting a formal semantics of UML statecharts is well beyond the scope of this thesis). In reading this section, the reader is encouraged to refer to Section 4.1.1 and Section 4.2.1 for examples. This formalism is largely based on [46]. An OO UML model is a set of classes. A class has a name, data, and behaviour. Class data is a set of variables called attributes. A variable is a tuple $(name, type) \in \mathbf{Var}$.

$$\mathbf{Var} = \mathbf{Id} \times \{ \texttt{int}, \texttt{bool}, \texttt{class} \}$$

Class behaviour is a statechart. A statechart S is a tuple:

$$(State_{and},\ State_{or}, \searrow,\ \text{ini},\ Signal,\ Call,\ Trans,\ \text{label}) \in \textbf{Statechart}$$

A description of elements of this tuple follows. The set of states $State = State_{and} \cup State_{or}$ and the superstate relation $\searrow \subseteq State \times State$ form a state tree rooted in $\texttt{root} \in State_{or}$ with two restrictions:

- Along any path from the root to a leaf, state types alternate between *and* (i.e. $st \in State_{and}$) and *or* (i.e. $st \in State_{or}$).
- Leafs are necessarily *and* states.

For every *or* state $st \in State_{or}$, one child, $\text{ini}(st) \in State_{and}$, is designated as its initial state. When an *and* state is entered, the initial state of its children are automatically entered. $Event = Signal \cup Call$ is the set of events that statechart S can receive. *Signal* events are for asynchronous (non-blocking) communication while *call* events are for synchronous (blocking - i.e., sender blocks until event processing completes) communication. An event is a tuple $(name,\ Args) \in \textbf{Ev}$: It has a name, and a sequence of variables called arguments. $Trans$ is the set of transitions of statechart S. A transition is a tuple $(src,\ e,\ g,\ act,\ dst) \in \textbf{Trans}$ with the equivalent graphical notation

$$src \xrightarrow{e[g]/act} dst$$

where src/dst, e, g, and act are the transition's source/destination state, trigger, guard, and action respectively.

$$\textbf{Trans} = State_{and} \times (Event \ \cup \ \{ \ (*, Args = \emptyset) \ \}) \times \textbf{Guard} \times \textbf{Action} \times State_{and}$$

A transition with trigger $*$ is called a *null transition*. **Guard** is the set of logical formulae over variables in the *scope* of S (i.e. data of the class with behaviour S) and **Action** is generated by the grammar:

$$\begin{aligned}
Action \quad & ::= Assign \mid Invoke \mid \texttt{skip} \\
Assign \quad & ::= Id \ \texttt{:=} \ Expr \\
Invoke \quad & ::= (Id \mid \texttt{self}) \ (\texttt{.} \mid \texttt{!}) \ InvokeExpr \\
InvokeExpr \quad & ::= Id \ \texttt{(} \ [Expr \ (\texttt{,} \ Expr)^*] \ \texttt{)}
\end{aligned}$$

The action language includes *assignment* actions (e.g. $a := 5$) and *event invocation* actions (e.g. $obj!ev()$ for sending a signal event and $obj.ev()$ for sending a call event). Here we assume only simple actions without loss of expressiveness: compound actions such as conditionals and sequences can be realized by combining simple actions with statechart mechanisms such as guarded and null transitions: a transition with a compound sequence action $src \xrightarrow{e[g]/act_1;act_2} dst$ can be expanded to the transitions $src \xrightarrow{e[g]/act_1} st_i \xrightarrow{*[true]/act_2} dst$; and a transition with a conditional action $src \xrightarrow{e[g]/if(cond) \ act_1 \ else \ act_2} dst$ can be expanded to the transitions $src \xrightarrow{e[g \wedge cond]/act_1} dst$ and $src \xrightarrow{e[g \wedge !cond]/act_2} dst$. Actions can have labels. The label of an action act is given by label(act) $\in$ **Id** if one exists.

The active *configuration* σ of statechart S is the set of states in which it resides (i.e. its active states). The following rules apply:

- `root` is always active.
- If an *and* state is active, then so are all of its children.
- If an *or* state is active then so is exactly one of its children.
- If a state is active, then so are all of its ancestors.

The execution state of S is a tuple $\langle \sigma, \nu, q \rangle$ where σ is the active configuration, ν is a map from variables in the scope of S to their values, and q is the queue for events received by S. The initial execution state is $\langle \sigma_0, \nu_0, q_0 \rangle$ where σ_0 is given by the function ini and the state `root` (defined above) and the above rules on configurations, ν_0 is given by the model's *initial instantiation* described later in this section, and $q_0 = \emptyset$. An execution state is *stable* if no state in the active configuration is the source of a null transition and is *transient* otherwise. Events in q are processed one-by-one in FIFO order in a *run to completion* (RTC) step so long as q is not empty. The RTC processing of an event e takes s from one stable execution state to the next, $\langle \sigma_i, \nu_i, q_i \rangle \xrightarrow{e} \langle \sigma_{i+1}, \nu_{i+1}, q_{i+1} \rangle$, by the following process (adopted from [17]):

1. *All enabled transitions are identified:* A transition is enabled if its source state is in σ_i, it is triggered by e, and its guard is true with respect to values of variables in the scope of S described by ν.

2. *Enabled transitions are fired:* Firing a transition $src \xrightarrow{e[g]/act} dst$ causes s to leave src (i.e. src and its descendants that were in the active configuration are removed from the active configuration), execute act updating ν_i (due to assignment actions) and q_i (due to event invocation actions to self), and enter dst (i.e. dst and a subset of its descendants determined by ini and the rules governing active configurations described above are added to the active configuration) updating σ_i. Two enabled transitions are in *conflict* if their source states have an ancestry relation, i.e. one is an ancestor of the other (multiple enabled transitions with the same source state are disallowed). Between conflicting transitions, only the transition whose source state is lowest in the state tree fires. The order of firing of enabled transitions is non-deterministic.

3. *Null transitions are handled:* If Step 2 lands S in a transient execution state, $*$ is dispatched causing all enabled null transitions to fire as per Step 2. This loop continues until a stable execution state is reached. Intermediate steps that occur within an RTC step are called *microsteps*.

Based on the above definitions, a class c can be defined as a tuple $(name, Attr,\ s) \in$ **Class**

$$\mathbf{Class} = \mathbf{Id} \times \mathcal{P}(\mathbf{Var}) \times \mathbf{Statechart}$$

We will augment the definition of an OO model with a specification of its *initial instantiation*: i.e. the set of objects, and initial values of their attributes, in the model's initial execution state. An object is a tuple $(name, c, \text{iniInst}) \in \mathbf{Object}$.

$$\mathbf{Object} = \mathbf{Id} \times \mathbf{Class} \times (\mathbf{Var} \rightarrow \mathbf{Value})$$

Finally we define an OO model as a set of classes and their initial instantiation: i.e. a tuple $(C, O) \in \mathbf{OOM}$ where $\forall o \in O,\ o.c \in C$ (i.e. all objects in the models are instances of classes in the model).

$$\mathbf{OOM} = \mathcal{P}(\mathbf{Class}) \times \mathcal{P}(\mathbf{Object})$$

3.2 The Aspect-Oriented Model

This section gives a semi-formal definition of AO models which is an extension/modification of the AO modeling approach of [28] (our contributions will be discussed at the end of this section). In reading this section, the reader is encouraged to refer to Section 4.1.1 and Section 4.2.1 for examples. An AO model has two parts: a UML part (i.e. an OO model), and a WRL part. The UML models data and behaviour for each concern (core or aspect) with classes. The WRL specifies how concerns cross-cut one another. Hence, an AO model is a tuple in the set:

$$\mathbf{AOM} = \{\ (oom, wrl) \mid oom \in \mathbf{OOM} \wedge wrl \in \mathbf{WRL}(oom)\ \}$$

The set $\mathbf{WRL}(oom)$ is defined in the remainder of this section. The WRL part of an AO model maps join points in the behaviour of an instance of one class $c = (\mathtt{C}, Attr, s)$ (the core) to advice of instances of other classes (aspects). The WRL join point model follows:

- *Event join point:* Is a tuple $(\sigma, e) \in \mathbf{JP_{ev}}(c)$ and corresponds to the RTC processing of event e by the core statechart, when it is in a configuration σ_i just before the event is processed, where $\sigma \subseteq \sigma_i$.

  $$\mathbf{JP_{ev}}(c) = \mathcal{P}(c.s.State) \times c.s.Event$$

  For σ to be valid, it must be the subset of *some* configuration of the core:

  $$\forall s_1, s_2 \in \sigma,\ \mathrm{lca}(s_1, s_2) \in c.s.State_{or} \implies s_1 \in \mathrm{ancest}(s_2) \vee s_2 \in \mathrm{ancest}(s_1)$$

  where lca and ancest stand for *least common ancestor* and *ancestor* respectively.

- *Action join point:* Is a label $l \in \mathbf{JP_{act}}(c)$ and corresponds to the execution of the action labeled l in the core statechart.

  $$\mathbf{JP_{act}}(c) = \{l \mid l \in \mathrm{Range}(c.s.\mathrm{label})\}$$

  The WRL join point model can be extended to include join points for specific actions (e.g. event invocation).

A join point can expose contextual data from the core that may be used in advice. The context of event join point (σ, e) is $e.Args$. By default, action join points have

no context; however, context can be defined for extensions (e.g. an event invocation join point can expose parameters of the invocation).

WRL advice is the aspect statechart's evolution in response to a join point. Several possible evolutions are described as a *tree* of evolution steps (or *advice nodes*), with each path from the root to a leaf corresponding to one possible evolution. Advice can be specified to apply *before* or *after* the join point. For an aspect (which is a class) $a = (\texttt{A}, Attr, s)$, advice nodes can take one of two forms:

- *Action node:* Is a tuple $(\sigma, act) \in \mathbf{Node_{act}}(a)$

$$\mathbf{Node_{act}}(a) = \mathcal{P}(a.s.State) \times (InvokeExpr \ \cup \ \{ \ Skip \ \})$$

  where $act = e(params)$ or `skip`. If $act = e(params)$ the node's action is a single evolution step of the aspect statechart by the execution of the event invocation action $e(params)$, which leads to the RTC processing of event $e \in a.s.Event$ with arguments *params* (expressions over the the advised join point's context) by the aspect statechart. If $act =$ `skip`, the node's action is to do nothing. The node's action is performed only if the node is *enabled*: i.e., the aspect statechart is initially in a configuration σ_i, where $\sigma \subseteq \sigma_i$. For σ to be valid, it must be the subset of *some* configuration of the aspect.

- *Consume node:* A special node $con \in \mathbf{Node_{con}}(a)$

$$\mathbf{Node_{con}}(a) = \{ \ con(a)_i \mid i \in \mathbb{N} \ \}$$

that does not evolve the aspect statechart; rather it halts advice execution and consumes the advised join point. Here, the label *con* symbolizes a consume node and the subscripts i are used to differentiate one consume node in the tree from another.

We define an advice of aspect a as a tuple $(root(a),\ N_{act},\ N_{con}, \searrow) \in \mathbf{Advice}(a)$.

$$\mathbf{Advice}(a) = \{\ root(a)\ \} \times \mathbf{Node_{act}}(a) \times \mathbf{Node_{con}}(a) \times (\mathbf{Node}(a) \times \mathbf{Node}(a)$$

where $\mathbf{Node}(a) = \{\ root(a)\ \} \cup\ setNode_{act}(a) \cup\ setNode_{con}(a)$. The set of nodes $N = \{\ root(a)\ \} \cup\ N_{act} \cup\ N_{con}$ and the parent relation $\searrow$ form a tree rooted in *root* with the following restrictions:

- Consume nodes must be leaves, cannot have siblings, and can only appear in *before* advice

- Sibling action nodes cannot be concurrently enabled for any configuration

$$\forall n_1, n_2 \in N_{act}, n_1 \in \text{siblings}(n_2),\ \exists s_1 \in n_1.\sigma, s_2 \in n_2.\sigma,$$
$$\text{lca}(s_1, s_2) \in a.s.State_{or} \wedge s_1 \notin \text{ancest}(s_2) \wedge s_2 \notin \text{ancest}(s_1)$$

  where siblings(n) means the siblings of node n in the advice tree.

Upon occurrence of the advised join point, the aspect statechart evolves through a sequence of steps described by action nodes of the advice tree along a path that is traced as follows: starting from the root and until a leaf is reached or the path is

blocked, the path is extended by the enabled child of its tail and the action described by the enabled child is performed (note that consume nodes are always enabled). If the tail has no enabled children, the path is blocked.

Based on the definitions above, we define a WRL specification as a tuple in the set

$$\mathbf{WRL}(oom) = \{\ (\mathrm{am}, \mathrm{om}) \mid\ \mathrm{am} \in \mathbf{AdvMap}(oom) \wedge \mathrm{om} \in \mathbf{ObjMap}(oom, \mathrm{am})\ \}$$

The first part of the WRL is an *advice map*, which is a member of the set:

$$\mathbf{AdvMap}(oom) : \mathbf{CoreJP}(oom) \rightarrow \mathcal{P}(\mathbf{AspectAdv}(oom))$$

$$\mathbf{CoreJP}(oom) = \{\ (c, jp) \mid c \in oom.c \wedge jp \in \mathbf{JP}(c)\ \}$$

$$\mathbf{AspectAdv}(oom) = \{\ (a, adv) \mid a \in oom.c \wedge adv \in \mathbf{Advice}(a)\ \}$$

where $\mathbf{JP}(c) = \mathbf{JP_{ev}}(c)\ \cup\ \mathbf{JP_{act}}(c)$. An element $[(c, jp) \mapsto AS] \in \mathbf{AdvMap}(oom)$, specifies that for OO model *oom*, the occurrence of join point jp in an instance of the core class c triggers a set of advice AS that is partitioned into a set of before advice AS_{bef} and a set of after advice AS_{aft}. Each partition of AS is *totally ordered*: the total order (of advice *precedence*) on AS serves as a *composition operator* on aspects. An element $(a, adv) \in AS$ specifies advice adv of an instance of the aspect class a. The above formulation allows a class to be both a core and and an aspect; hence, the possibility of *aspects of aspects*. We impose the following restrictions on advice maps: 1) aspect classes cannot receive *signal* events and 2) two classes may not mutually

(transitively) advise one another. The reason for these restrictions will be explained in Section 3.4.

The second part of the WRL is an *object map*, which is a member of the set:

$$\begin{aligned}\mathbf{ObjMap}(oom, \mathrm{am}) = \{\ (oc, a) \mapsto oa \mid\ & oc \in \mathbf{CoreObj}(oom, \mathrm{am}) \wedge \\ & a \in \mathbf{Aspect}(oom, \mathrm{am}, oc.c) \wedge \\ & oa \in oom.O\ \wedge oa.c = a\ \}\end{aligned}$$

$$\mathbf{CoreObj_{ev/act}}(oom, \mathrm{am}) = \{\ o \in oom.O \mid o.c \in \mathbf{Core_{ev/act}}(oom, \mathrm{am})\ \}$$

$$\mathbf{CoreObj}(oom, \mathrm{am}) = \mathbf{CoreObj_{ev}}(oom, \mathrm{am})\ \cup\ \mathbf{CoreObj_{act}}(oom, \mathrm{am})$$

$$\begin{aligned}\mathbf{Core_{ev/act}}(oom, \mathrm{am}) = \{\ c \in oom.C \mid\ & \exists jp \in \mathbf{JP_{ev/act}}(c), a, adv, \\ & (c, jp) \mapsto (a, adv) \in \mathrm{am}\ \}\end{aligned}$$

$$\begin{aligned}\mathbf{Aspect_{ev/act}}(oom, \mathrm{am}, c) = \{\ a \in oom.C \mid\ & \exists jp \in \mathbf{JP_{ev/act}}(c), adv, \\ & (c, jp) \mapsto (a, adv) \in \mathrm{am}\ \}\end{aligned}$$

$$\mathbf{Aspect}(oom, \mathrm{am}, c) = \mathbf{Aspect_{ev}}(oom, \mathrm{am}, c)\ \cup\ \mathbf{Aspect_{act}}(oom, \mathrm{am}, c)$$

An element $(oc, a) \mapsto oa \in \mathbf{ObjMap}(oom, \mathrm{am})$ specifies that for OO model *oom* with advice map am, the core instance *oc* is advised by the instance *oa* of aspect a. We require that every instance of a core be assigned exactly *one* instance of every aspect the core is advised by (this restriction can be can be checked for by a straightforward analysis of the WRL specification).

WRL	::= $Aspect+$ $Precedence*$
$Aspect$	::= `className` $Core+$
$Core$	::= `className` ((`advLabel:`)? ($BeforeAdvice$ | $AfterAdvice$)) + $ObjectM$
$BeforeAdvice$	::= `before` $JoinPoint$ (`consume` | $BeforeAdviceNode+$)
$AfterAdvice$	::= `after` $JoinPoint$ $AfterAdviceNode+$
$BeforeAdviceNode$	::= $AdviceAction$ (`consume` | $BeforeAdviceNode*$)
$AfterAdviceNode$	::= $AdviceAction$ $AfterAdviceNode*$
$AdviceAction$	::= ($StateExp$, `aspectEvent(`$Args$`))` | `null`
$JoinPoint$	::= ($StateExp$, `coreEvent)` | `actionLabel`
$ObjectMap$	::= (`coreObject -> aspectObject`)+
$Precedence$	::= `coreName` $JoinPoint$`: adviceLabel(> adviceLabel`)+

Figure 3.1: Alternative WRL syntax

We have presented a mathematical syntax for WRL; however, alternative practitioner friendly syntax such as that shown in Figure 3.1 (which resembles conventional programming languages) can be developed ($StateExp$ is a regular expression that matches a set of states). See sections Section 4.1.1 and Section 4.2.1 for a correspondence between the mathematical syntax for WRL and that of Figure 3.1.

At this point it is appropriate to highlight the extensions/modifications of our AO modeling approach with respect to that presented in [28]. The AO modeling approach of [28] consists of modeling each concern structurally and behaviourally using UML class and statechart diagrams. The weaving of aspects with the core is based on the notion of *event interception and reinterpretation*: events targeted at the core are intercepted and reinterpreted to an event of the aspect. Reinterpretation can occur before/after the core handles the event and may also consume the event. This frame-

work for specifying the behaviour of AO systems using statecharts is implemented in Java as event interception and reinterpretation cannot be expressed directly in the UML. Our approach extends these concepts by introducing an extensible join point model for UML statecharts that not only captures event interception/reinterpretation via *event* join points but also includes *action* join points which allow aspects to extend/modify core behaviour in more interesting ways. In addition, the presentation of advice as a tree as well as advice precedence are new to our approach. But perhaps more importantly, we separate weaving rule specification from the concern models (which can be produced with the user's favourite UML CASE tool) and present a weaving algorithm in Section 3.4 to produce a woven UML model from the concern models and the weaving rules. This woven model can be formally verified using UML verification tools, the process of which is much less expensive than the verification of Java models produced via the AO statechart framework of [28].

3.3 Syntactic Analysis of the AO model

The syntactic analysis of the AO model reveals the following to the developer:

- When multiple advice applies to the same join point. While this information is explicit in the WRL syntax of Section 3.2 (as the mapping of join points to sets of advice), it may be hidden in an alternative WRL syntax such as that

of Figure 3.1. In the syntax of Figure 3.1, a WRL specification is made up of a number of aspect declarations (each designates a class as an aspect), each of which can have any number of core declarations (each designates classes that the aspect advises). Each core declaration is made up of one or more pieces of advice that apply to given join points of the core. Detecting instances where more than one piece of advice applies to the same join point is the straightforward process of maintaining a list of advised join points per aspect (which can easily be done during parsing) and computing the intersection of these lists.

- When one advice consumes a join point preventing other advice from executing. This could happen in two cases:
  - A *before* advice consumes a join point preventing all *before* advice (of lower precedence) and all *after* advice that apply to the same join point from executing.
  - An advice consumes an *event* join point preventing all advice on *action* join points that may occur *within* the event join point from executing. For action join point $jp_{act} = l$ and event join point $jp_{ev} = (\sigma, e)$ of core c, define tr as:

    $$tr \in c.s.Trans,\ tr.act = c.s.\text{label}(l)$$

    Let $\text{trans}(e, \sigma_i, \sigma_f)$ be the set of transitions that fire in the RTC step

$\langle \sigma_i, \nu_i, q_i \rangle \xrightarrow{e} \langle \sigma_f, \nu_f, q_f \rangle$ for some ν_i, q_i, ν_f, q_f, and define Tr as:

$$
\begin{aligned}
Tr &= \bigcup_{\sigma_i \in \Sigma_i, \sigma_f \in \Sigma_f(e,\sigma_i)} \text{trans}(e, \sigma_i, \sigma_f) \\
\Sigma_i &= \{\ \sigma_i \in \text{valid configurations of } c.s \mid \sigma \subseteq \sigma_i\ \} \\
\Sigma_f(e, \sigma_i) &= \{\ \sigma_f \mid \exists \nu_i, q_i, \nu_f, q_f,\ \langle \sigma_i, \nu_i, q_i \rangle \xrightarrow{e} \langle \sigma_f, \nu_f, q_f \rangle\ \}
\end{aligned}
$$

We say that jp_{act} may occur within jp_{ev} if an only if $tr \in Tr$. The computation of T could be expensive, in particular, due to the computation of the Σ_f sets, which requires an (potentially exhaustive) exploration of the state-space. To make the computation feasible, one could specify a maximum depth on the exploration, as the goal is to simply alert the user of *possible* interactions. An exhaustive exploration of the state-space is deferred to task 2 of the process as explained in Section 1.3.

As stated in Section 1.3, such *advice overlaps* are *potential* sources of *aspect* interaction. However, not all aspect interactions are due to advice overlaps and not all advice overlaps cause aspect interactions. Additionally, advice overlaps do not point to aspect/core interactions.

3.4 The Weaving Process

In this section, we present a functional description of the transformation of an AO model to a (by definition) behaviorally equivalent OO model. The transformation

serves two purposes:

- It gives an operational semantics for WRL in UML.
- it describes an unoptimized weaving process (WP1).

The highest level description of the transformation is given by the function weave, which is defined below:
weave :**AOM** $\rightarrow$ **OOM**. Maps an AO model, *aom*, to an OO model that is a modification of the AO model's UML as prescribed by its WRL.

$$\text{weave}(aom) \stackrel{\text{df}}{=}$$

$$(\text{modClasses}(aom),\ \text{modObjects}(aom))$$

modClasses :**AOM** $\rightarrow \mathcal{P}$(**Class**). Maps AO model, *aom*, to a set of classes that is a modification of the AO model's classes as prescribed by its WRL. This involves:

- Adding one proxy class per core class whose event join points are advised
- Modifying core classes whose event join points are advised to enable synchronous communication with their proxy
- Transferring references to core classes whose event join points are advised to their proxy counterparts
- Modifying core classes whose action join points are advised

$\text{modClasses}(aom) \stackrel{\text{df}}{=}$

$\text{modCoreClients}(Core_{ev},\ aom.oom.C \setminus Core\ \cup\ \mathbf{Proxy}\ \cup\ \mathbf{Core}')$

$\mathbf{Proxy} = \{\ \text{proxy}(aom, c) \mid c \in Core_{ev}\ \}$

$\mathbf{Core}' = \{\ \text{modCore}(aom, c) \mid c \in Core\ \}$

$Core_{ev/act} = \mathbf{Core_{ev/act}}(aom.oom, aom.wrl.\text{am})$

$Core = Core_{ev}\ \cup\ Core_{act}$

Complexity. The number of classes added to aom is $|Core_{ev}|$.

proxy :$\mathbf{AOM} \times \mathbf{Class} \to \mathbf{Class}$. Maps a core class c in AO model aom, whose event join points are advised, to a *proxy* class for the core. The purpose of the proxy is to implement advice on event join points of the core. We describe the proxy class in parts (*partial* classes) and obtain a full description by *merging* the parts: *base* describes the proxy name $\texttt{P}_\texttt{c}$ (all names for classes, attributes, and states introduced by the weaving process are arbitrary and are assumed to be unique in their scope); its data, which are associations with the core ($\texttt{attr}_\texttt{c}$) and all aspects a that advise event join points on the core ($\texttt{attr}_a$); and its basic behaviour, which is to forward incoming events that do not correspond to advised event join points, to the core. *advSetElems* is merger of a set of partial classes that each describe class elements that implement a set of advice on an advised event join point.

$$\text{proxy}(aom, c) \stackrel{\text{df}}{=}$$

$$p_c = \text{merge}(\mathtt{P_c}, \{\ base, advSetElems\ \})$$

`// References to core and aspects, and basic behaviour`

$$base = (\mathtt{P_c}, Attr, s)$$

$$Attr = \{\ \mathtt{attr_c} : c\ \} \cup \{\ \mathtt{attr_a} : a \mid a \in Aspect_{ev}\ \}$$

$$Aspect_{ev} = \mathbf{Aspect_{ev}}(aom.oom, aom.wrl.\text{am}, c)$$

$$s = (\ State_{and} = \{\ \mathtt{idle}\ \},\ State_{or} = \{\ \mathtt{root}\ \},\ \searrow = \{\ (\mathtt{root}, \mathtt{idle})\ \},$$

$$\text{ini} = \{\ \mathtt{root} \mapsto \mathtt{idle}\ \},\ Signal = c.s.Signal,\ Call = c.s.Call,$$

$$Trans = Tr,\ \text{label} = \emptyset)$$

$$Tr = \{\ \mathtt{idle} \xrightarrow{e[\text{g}(e)]/\text{act}(e)} \mathtt{idle} \mid e \in c.s.Event\ \}$$

$$\text{g}(e) = \begin{cases} \mathtt{true} & \nexists jp \in \mathbf{JP_{ev}},\ jp.e = e \\ \forall \mathtt{jp} \in \mathbf{JP_{ev}},\ \mathtt{jp}.\sigma \not\subseteq \mathtt{attr_c}.\sigma & else \end{cases}$$

$$\text{act}(e) = \begin{cases} \mathtt{attr_c}!\mathtt{e}(ergs) & e \in c.s.Call \\ \mathtt{attr_c}.\mathtt{e}(e.Args) & e \in c.s.Signal \end{cases}$$

`// Elements implementing advice on event join points`

$$advSetElems = \text{merge}(\mathtt{P_c}, \{\ \ \text{advSetElems}(\mathtt{P_c}, aom, c, jp, \mathtt{idle}, \mathtt{idle}) \mid jp \in \mathbf{JP_{ev}}\ \})$$

$$\mathbf{JP_{ev}} = \{\ jp \mid\ \ jp \in \mathbf{JP_{ev}}(c) \wedge (c, jp) \in \text{Domain}(aom.wrl.\text{am}) \wedge aom.wrl.\text{am}(c, jp) \neq \emptyset\ \}$$

Complexity. The number of attributes, states, and transitions added to p_c is $1 + |Aspect_{ev}|$, 1, and $|c.s.Event|$, respectively.

modCore :**AOM** $\times$ **Class** $\rightarrow$ **Class**. Maps a core class c in AO model *aom* to a modified version of the core. There are two modification tasks:

- Core data and behaviour are modified to wrap advised *action* join points in before and after advice (if present): this involves adding to core data, associations to aspects that advise action join points of the core ($Attr$), and replacing core transitions whose action execution corresponds to an advised action join point, with states and transitions that describe the join point *and* its set of advice ($advSetElems$).

- If any *signal* receptions of the core trigger an advised *event* join point, they are made into *call* receptions. This modification, together with the restriction from Section 3.2 that aspects can only have *call* event receptions, are necessary to impose order on the evolution of core and aspect statecharts, since call events correspond to *synchronous* (blocking) communication. However, for the same reason, mutual advice between two classes leads to a woven model that *deadlocks*; hence the restriction from Section 3.2 that disallows mutual advice.

$$\text{modCore}(aom, c) \stackrel{\text{df}}{=}$$

$$\text{merge}(c.name, \{\ (c.name, c.Attr\ \cup\ Attr, s),\ advSetElems\ \})$$

`// References to aspects`

$Attr = \{\ \texttt{attr}_{\texttt{a}} : a \mid a \in Aspect_{act}\ \}$

$Aspect_{act} = \mathbf{Aspect_{act}}(aom.oom, aom.wrl.\mathrm{am}, c)$

`// Signal to call transformation and transition removal`

$s = (\ c.State_{and}, \ldots,$

$\quad Signal = c.Signal \setminus \mathbf{Sig}, Call = c.Call\ \cup\ \mathbf{Sig},$

$\quad Trans = c.Trans \setminus \{\ \mathrm{tr}(jp) \mid jp \in \mathbf{JP_{act}}\ \}, c.\mathrm{label})$

$\mathbf{Sig} = \{\ sig \in c.s.Signal \mid \exists jp \in \mathbf{JP_{ev}},\ sig = jp.e\ \}$

$\mathrm{tr}(jp) = t \in c.s,\ t.act = c.s.\mathrm{label}(jp)$

`// Elements implementing advice on action join points`

$advSetElems = \mathrm{merge}(\{\ \ \mathrm{advSetElems}(\ c.name, aom, c, jp,$

$\qquad\qquad \mathrm{tr}(jp).src, \mathrm{tr}(jp).dst) \mid$

$\qquad jp \in \mathbf{JP_{act}}\ \})$

$\mathbf{JP_{ev/act}} = \{\ jp \mid\ \ jp \in \mathbf{JP_{ev/act}}(c) \wedge (c, jp) \in \mathrm{Domain}(aom.wrl.\mathrm{am}) \wedge$

$\qquad aom.wrl.\mathrm{am}(c, jp) \neq \emptyset\ \}$

Complexity. The number of attributes added to c is $|Aspect_{act}|$.

advSetElems :$\mathbf{Id} \times \mathbf{AOM} \times \mathbf{Class} \times \mathbf{JP}(c) \times \mathbf{Id} \times \mathbf{Id} \rightarrow \mathbf{Class}$. Maps a core join point (c, jp) of AO model *aom* to a partial class that describes elements of the class *id*, that implement the *set* of advice *as* on the core join point. The partial class itself

is described in parts: $attrs$ describes attributes $Attr_{e\_arg}$ that are place-holders for arguments of event e (the event that starts *event* join point jp or triggers the transition whose action execution is *action* join point jp), and an attribute $attr_{jp\_con}$ that flags the consumption of the join point by some *before* advice; $advTrees$ is the merger of a set of partial classes that each describe a *cluster* of states and transitions that implement a single advice tree in as (a cluster for advice adv has an initial state $\texttt{st}_{\texttt{adv\_i}}$ and a final state $\texttt{st}_{\texttt{adv\_f}}$); $advTreeLinks_{bef/aft}$ describes transitions that link the clusters for advice trees in $as_{bef/aft}$ in order of advice *precedence* (head/tail($as_{bef/aft}$) is the highest/lowest precedence advice in $as_{bef/aft}$); $advSetLinks$ describes transitions that link st_i (the state id resides in before the join point), lists of before/after advice tree clusters, and st_f (the state id resides in after the join point) so as to implement the behaviour of wrapping the join point with *before* and *after* advice (if present) and suppressing the join point if some *before* advice consumes it.

$$\text{advSetElems}(id, aom, c, jp, st_i, st_f) \stackrel{\text{df}}{=}$$

$$\text{merge}(id, \{\ attrs, advTrees, advTreeLinks_{bef}, advTreeLinks_{aft},$$

$$advSetLinks\ \})$$

```
// Argument place-holder and consume flag attributes
```

$$attrs = (id, Attr = Attr_{e\_arg}\ \cup\ Attr_{jp\_con}, s_{\emptyset})$$

$$Attr_{e\_arg} = \{\ \texttt{attr}_{\texttt{e\_arg}} : arg.type \mid arg \in e.Args\ \}$$

$$Attr_{jp\_con} = \begin{cases} \{\ \texttt{attr}_{\texttt{jp\_con}} : \texttt{bool}\ \} & as_{bef} \neq \emptyset \\ \emptyset & as_{bef} = \emptyset \end{cases}$$

$s_\emptyset = (\emptyset, \ldots, \emptyset)$

`// Clusters of states and transitions each implementing`

`// an advice tree`

$$advTrees = \text{merge}(id, \{\ \ \text{advTreeElems}(id, c, jp, a, adv, \text{parent}(st_i)) \mid (a, adv) \in as\ \})$$

`// Transitions linking clusters that implement advice trees`

$$advTreeLinks_{bef/aft} = \text{advTreeLinks}_{\text{bef/aft}}(id, aom, c, jp, as, st_i)$$

`// Transitions linking the idle state, and cluster sets`

`// for before and after advice`

$$advSetLinks = (\ id, Attr = \emptyset, s = (\emptyset, \ldots, Trans = Tr_s\ \cup\ Tr_m\ \cup\ Tr_f, \emptyset))$$

$$Tr_s = \{\ st_i \xrightarrow{e[g]/Attr_{e\_arg} := e.Args; stm} \texttt{st}_{adv\_\texttt{i}}\ \}$$

$$stm, adv = \begin{cases} \texttt{attr}_{\texttt{jp\_con}} := \texttt{false}, \text{head}(as_{bef}) & as_{bef} \neq \emptyset \\ act, \text{head}(as_{aft}) & as_{bef} = \emptyset \end{cases}$$

$$Tr_m = \begin{cases} \emptyset & as_{bef} = \emptyset \vee as_{aft} = \emptyset \\ \{\ \texttt{st}_{\text{tail}(as_{bef})\_\texttt{f}} \xrightarrow{[!\texttt{attr}_{\texttt{jp\_con}}]/act_{e.Args \leftarrow Attr_{e\_args}}} \texttt{st}_{\text{head}(as_{aft})\_\texttt{i}}\ \} & else \end{cases}$$

$$Tr_f = \{\ \texttt{st}_{adv\_\texttt{f}} \xrightarrow{l} st_f\ \}$$

$$l, adv = \begin{cases} \epsilon, \text{tail}(as_{aft}) & as_{aft} \neq \emptyset \\ [!\texttt{attr}_{\texttt{jp\_con}}]/act_{e.Args \leftarrow Attr_{e\_args}}, \text{tail}(as_{bef}) & as_{aft} = \emptyset \end{cases}$$

$$e, g, act = \begin{cases} jp.e,\ jp.\sigma \subseteq \texttt{attr}_\texttt{c}.\sigma,\ \texttt{attr}_\texttt{c}.\texttt{e}(\texttt{e.Args}) & jp \in \mathbf{JP_{ev}}(c) \\ tr.e,\ tr.g,\ tr.act & jp \in \mathbf{JP_{act}}(c) \end{cases}$$

$tr = t \in c.s,\ t.act = c.s.\text{label}(jp)$

$as = aom.wrl.\text{am}(c, jp)$

Complexity. The number of attributes and transitions added to id is at most $1 + |e.Args|$ and at most 3, respectively.

$\text{advTreeLinks}_{\text{bef/aft}}$:$\mathbf{Id} \times \mathbf{AOM} \times \mathbf{Class} \times \mathbf{JP}(c) \times \mathcal{P}(\mathbf{AspectAdvice})(aom.oom)$ $\times \mathbf{Id} \rightarrow \mathbf{Class}$. Maps a set of *before/after* advice as on core join point (c, p) of AO model aom to a partial class that describes transitions in the statechart of proxy id that link *clusters* of statechart elements for each advice in as in order of precedence. For *before* advice, the final state of each advice cluster is additionally linked to the *idle* state conditional upon the consume flag, $attr_{jp\_con}$, to prevent execution of further advice upon consumption of the join point.

$$\text{advTreeLinks}_{\text{bef/aft}}(id, aom, c, jp, as, st_i) \stackrel{\text{df}}{=}$$

$$\begin{cases} \text{merge}(id, \{\ links, \text{advTreeLinks}_{\text{bef/aft}}(\text{pop}(as))\ \}) & |as| \geq 1 \\ (id, Attr = \emptyset, s_\emptyset) & else \end{cases}$$

$links = (id, Attr = \emptyset, s = (\emptyset, \ldots, Trans = Tr_{bef/aft}, \emptyset)))$

$$Tr_{bef} = \{\ \mathtt{st}_{\mathrm{top}(as_{bef})\_\mathtt{f}} \xrightarrow{[\mathtt{attr}_{\mathtt{jp\_con}}]} st_i\ \} \cup \begin{cases} \{\ \mathtt{st}_{\mathrm{top}(as_{bef})\_\mathtt{f}} \xrightarrow{[!\mathtt{attr}_{\mathtt{jp\_con}}]} \mathtt{st}_{\mathrm{top}(\mathrm{pop}(as_{bef}))\_\mathtt{i}}\ \} & |as| > 1 \\ \emptyset & else \end{cases}$$

$$Tr_{aft} = \begin{cases} \{\ \mathtt{st}_{\mathrm{top}(as_{aft})\_\mathtt{f}} \rightarrow \mathtt{st}_{\mathrm{top}(\mathrm{pop}(as_{aft}))\_\mathtt{i}}\ \} & |as| > 1 \\ \emptyset & else \end{cases}$$

Complexity. The number of transitions added to id is at most $2|as| - 1$

advTreeElems :$\mathbf{Id} \times \mathbf{Class} \times \mathbf{JP_{ev}}(c) \times \mathbf{Class} \times \mathbf{Advice}(a) \times \mathbf{Id} \rightarrow \mathbf{Class}$. Maps aspect advice (a, adv) applied to the event core join point (c, jp), to a partial class that describes a *cluster* of statechart elements of class id (under state st_p), that implement the aspect advice. The partial class is described in two parts: *terminalStates* describes the initial and final states of the cluster ($\mathtt{st}_{\mathtt{adv\_i}}$ and $\mathtt{st}_{\mathtt{adv\_f}}$) and *body* describes the internals of the cluster.

$\mathrm{advTreeElems}(id, c, jp, a, adv, st_p) \stackrel{\mathrm{df}}{=}$

$merge(id, terminalStates \cup \{\ body\ \})$

```
// Start and end states of cluster
```

$terminalStates = \{\ (id, Attr = \emptyset, s = sc))\ \}$

$sc = (\ State_{and} = \{\ \mathtt{st_{adv\_i}}, \mathtt{st_{adv\_f}}\ \},\ State_{or} = \emptyset,$

$\qquad \searrow = \{\ (st_p, \mathtt{st_{adv\_i}}), (st_p, \mathtt{st_{adv\_f}})\ \}, \emptyset, \ldots, \emptyset)$

`// Cluster body`

$body = advNodeElems(id, c, jp, a, adv, adv.root, st_p)$

Complexity. (together with advNodeElems). The number of states and transitions added to id is $1 + |adv.N|$ and at most $2|adv.N|$, respectively.

advNodeElems :$\mathbf{Id} \times \mathbf{Class} \times \mathbf{JP_{ev}}(c) \times \mathbf{Class} \times \mathbf{Advice}(a) \times a.Node \times \mathbf{Id} \rightarrow \mathbf{Class}$. Maps aspect advice (a, adv) applied to the event core join point (c, jp), to a partial class describing statechart elements of the class id (under state st_p) that implement the part of advice tree adv that is rooted in node n (i.e. part of the internals of the cluster for adv). The partial class has two parts: $nodeElems$: for a non-leaf node n, it describes states for children of n and transitions Tr_{child} leading to them from the state st corresponding to node n (these correspond to choices in selecting the next evolution step of the aspect statechart), as well as a possible transition Tr_{exit} from st to the final state of the cluster (this corresponds to exiting advice execution when the execution path is blocked); for a leaf node n, it simply describes a transition Tr_{leaf} from st to the final state of the cluster to mark the end of advice execution. $childElems$ is the merger of partial classes each describing a part of the cluster for adv due to subtrees of adv rooted in a child of n.

$\text{advNodeElems}(id, c, jp, a, adv, n, st_p) \stackrel{\text{df}}{=}$

$\text{merge}(id, nodeElems \ \cup \ \{ \ childElems \ \})$

`// Child node states, and transitions leaving the state`

`// corresponding to n`

$nodeElems = (id, Attr = \emptyset, s = sc)$

$sc = (\ State_{and} = \{ \ \mathtt{st}_{\mathtt{adv\_}ch} \mid ch \in \text{children}(n) \ \},\ State_{or} = \emptyset,$

$\qquad \searrow\!= \{ \ (st_p, \mathtt{st}_{\mathtt{adv\_}ch}) \mid ch \in \text{children}(n) \ \},\ \emptyset, \ldots,\ \emptyset,$

$\qquad Trans = Tr_{child} \ \cup \ Tr_{leaf} \ \cup \ Tr_{exist},\ \emptyset)$

$Tr_{child} = \{ \ st \xrightarrow{\text{l}(ch)} \mathtt{st}_{\mathtt{adv\_}ch} \mid ch \in \text{children}(n) \ \}$

$$\text{l}(ch) = \begin{cases} [ch.\sigma \subseteq \mathtt{attr}_{\mathtt{a}}.\sigma]/ch.act_{jp.e.Args \leftarrow Attr_{e\_args}} & ch \in adv.N_{act} \\ /\mathtt{attr}_{\mathtt{jp\_con}} := \mathtt{true} & ch \in adv.N_{con} \end{cases}$$

$$Tr_{leaf} = \begin{cases} \{ \ st \rightarrow \mathtt{st}_{\mathtt{adv\_f}} \ \} & children(n) = \emptyset \\ \emptyset & else \end{cases}$$

$$Tr_{exit} = \begin{cases} \emptyset & children(n) \ \cap \ adv.N_{con} \neq \emptyset \\ \{ \ st \xrightarrow{[\forall ch \in children(n),\ ch.\sigma \not\subseteq \mathtt{attr}_{\mathtt{a}}.\sigma]} \mathtt{st}_{\mathtt{adv\_f}} \ \} & else \end{cases}$$

$$st = \begin{cases} \mathtt{st}_{\mathtt{adv\_i}} & n = adv.root \\ \mathtt{st}_{\mathtt{adv}\_n} & else \end{cases}$$

`// Cluster parts rooted in children of n`

$childElems = merge(id, \{ \quad advNodeElems(id, c, jp, adv, ch) \mid$

$ch \in \text{children}(n) \ \})$

modCoreClients :$\mathcal{P}(\textbf{Class}) \times \mathcal{P}(\textbf{Class}) \rightarrow \mathcal{P}(\textbf{Class})$. Maps the set of classes $Class$ to a modified version of the set, where all references to core classes whose event joint points have been advised (i.e. members of $Core_{ev}$) are changed to references to proxies of those core classes.

$\text{modCoreClients}(Core_{ev}, Class) \stackrel{\text{df}}{=}$

$\{ \ (c.name, c.Attr \setminus Attr_{core} \ \cup \ Attr_{proxy}, c.s) \mid c \in Class \ \}$

$Attr_{proxy} = \{ \ (name, p_c) \mid (name, c) \in Attr_{core} \ \}$

$Attr_{core} = \{ \ attr \in c.Attr \mid attr.type \in Core_{ev} \ \}$

modObjects :$\textbf{AOM} \rightarrow \mathcal{P}(\textbf{Object})$. Maps AO model, aom, to a set of objects that is a modification of the AO model's objects as prescribed by its WRL. This involves:

- Adding a proxy instance per instance of core classes whose event join points are advised, and transferring initial references to such core instances to their proxy counterparts.

- Setting initial aspect references of proxy instances and instances of core classes whose action join points are advised, to the appropriate aspect instance as prescribed by the *object map* of the AO model's WRL.

$\text{modObjects}(aom) \stackrel{\text{df}}{=}$

$\{\ (o.name,\ o.c,\ o.\text{ini} \setminus \text{ini}_{core}\ \cup\ \text{ini}_{proxy}) \mid o \in \mathbf{Obj'}\ \}$

$\text{ini}_{proxy} = \{\ attr \mapsto op_c \mid attr \mapsto oc \in \text{ini}_{core}\ \}$

$\text{ini}_{core} = \{\ attr \mapsto oc \in o.\text{ini} \mid attr.type \in Core_{ev}\ \}$

$\mathbf{Obj'} = aom.oom.O \setminus CoreObj\ \cup\ \mathbf{CoreObj'}\ \cup\ \mathbf{ProxyObj}$

$\mathbf{CoreObj'} = \{\ (oc.name,\ oc.c,\ oc.ini\ \cup\ \text{ini}'(oc)) \mid oc \in CoreObj_{act}\ \}$

$\text{ini}'(oc) = \{\ \texttt{attr}_\texttt{a} \mapsto oa \mid a \in Aspect_{act}(oc.c) \wedge wrl.om(oc, a) = oa\ \}$

$\mathbf{ProxyObj} = \{\ (\texttt{op}_\texttt{c},\ p_c,\ \text{p\_ini}(oc)) \mid (\texttt{oc},\ c,\ \text{ini}) \in CoreObj_{ev}\ \}$

$$\begin{aligned}\text{p\_ini}(oc) = &\ \{\ \texttt{attr}_\texttt{c} \mapsto oc\ \}\ \cup \\ &\{\ \texttt{attr}_\texttt{a} \mapsto oa \mid a \in Aspect_{ev}(oc.c) \wedge wrl.om(oc, a) = oa\ \}\end{aligned}$$

$$\begin{aligned}CoreObj_{ev/act} &= \mathbf{CoreObj_{ev/act}}(aom.oom, aom.wrl.\text{am}) \\ Core_{ev/act} &= \mathbf{Core_{ev/act}}(aom.oom, aom.wrl.\text{am}) \\ Aspect_{ev/act}(c) &= \mathbf{Aspect_{ev/act}}(aom.oom, aom.wrl.\text{am}, c)\end{aligned}$$

Complexity. The number of objects added to aom is $|CoreObj_{ev}|$.

merge :$\mathbf{Id} \times \mathcal{P}(\mathbf{Class}) \to \mathbf{Class}$. Maps a set of partial descriptions of class id to a merger of those descriptions.

$\text{merge}(id, C) \stackrel{\text{df}}{=}$

$(id,\ \bigcup_{c \in C} c.Attr,\ (\bigcup_{c \in C} c.State_{and}, \ldots, \bigcup_{c \in C} c.label))$

If we define the size of an OO model as the sum of the number of classes, objects, attributes, states, and transitions in the model, we conclude from the above definitions that the size increase of the woven OO model with respect to the OO component of the unwoven AO model is in $O(|C_{ev}|+|Cobj_{ev}|+st_{ev}+st_{act}+attr_{ev}+attr_{act}+tr_{ev}+tr_{act})$ where

$$
\begin{aligned}
st_{ev/act} &= |C_{ev/act}| \times (jp_{ev/act} \times adv \times n) \\
attr_{ev/act} &= |C_{ev/act}| \times (a_{ev/act} + jp_{ev/act} \times args) \\
tr_{ev} &= |C_{ev}| \times (ev + jp_{ev} \times adv^2 \times n) \\
tr_{act} &= |C_{act}| \times (jp_{act} \times adv^2 \times n)
\end{aligned}
$$

with definitions

- $C_{ev/act}$: Core classes whose event/action join points are advised
- $CObj_{ev}$: Objects of classes in C_{ev}
- $a_{ev/act}$: Number of aspects advising each core in $C_{ev/act}$
- $jp_{ev/act}$: Number of advised event/action join points per core in $C_{ev/act}$
- $args$: Number of arguments of events of each advised join point
- adv : Number of advice trees per advised join point
- n : Number of nodes per advice tree
- ev : Number of event receptions per class

3.5 An Optimized Weaving Process

In this section, we present an optimized weaving process (WP2), whose outcome is a woven OO model better suited to formal verification. The key to the optimization is to move class elements that implement advice on event join points of a core from the core proxy to the core itself, removing the need for proxies. In the absence of proxies, the size of such elements (i.e. $st_{ev} + attr_{ev} + tr_{ev}$ defined in Section 3.4) has a lesser impact on the size of a flat finite state automata that simulates the woven OO model. This implies lower verification complexity. Unfortunately, the benefits of the optimization come with a cost: loss of support for *after* advice on event join points, since *after* advice applies after the completion of a join point, and while it is possible to observe when an advised event join point of a core completes from its proxy (this is when the call action that triggers the join point completes), it is not possible to do so from within the core in the presence of concurrency.

We augment the UML statechart action language of Section 3.1 with *condition* and *sequence* compound actions. We will replace states and transitions that implement an advice set on a core event join point (as introduced by WP1) with an equivalent compound action. This is necessary, since our optimized weaving process relies on the atomic execution of advice on event join points with respect to actions of the core. In the interest of reusing some of the definitions of Section 3.4, we will assume the unwoven model makes use only of simple actions.

$$
\begin{aligned}
Action &::= Assign \mid Invoke \mid \texttt{skip} \mid If \mid Seq \\
If &::= \texttt{if(}BoolExpr\texttt{)}Action \\
Seq &::= Action\texttt{;}Action
\end{aligned}
$$

In the following, we describe WP2 by its differences (function redefinitions and new functions) with WP1.

modClasses :$\mathbf{AOM} \to \mathcal{P}(\mathbf{Class})$. Maps AO model, *aom*, to a set of classes that is a modification of the AO model's classes as prescribed by its WRL. The modifications apply only to core classes.

$\text{modClasses}(aom) \stackrel{\text{df}}{=}$

$aom.oom.C \setminus Core \ \cup \ \mathbf{Core}'$

$\mathbf{Core}' = \{ \ \text{modCore}(aom, c) \mid c \in Core \ \}$

$Core_{ev/act} = \mathbf{Core_{ev/act}}(aom.oom, aom.wrl.\text{am})$

$Core = Core_{ev} \ \cup \ Core_{act}$

modCore :$\mathbf{AOM} \times \mathbf{Class} \to \mathbf{Class}$. Maps a core class c in AO model *aom* to a modified version of the core. Here, core data and behaviour are modified to prepend advised *event* join points with *before* advice, and to wrap *action* join points in before and after advice.

$\text{modCore}(aom, c) \stackrel{\text{df}}{=}$

$\text{merge}(c.name, \{ \ (c.name, c.Attr \ \cup \ Attr, s), \ advSetElems \ \})$

`// Modifications for advice on event join points`

$c' = \text{modCore}_{\text{ev}}(aom, c)$

`// Modifications for advice on action join points`

$Attr = \{\ \texttt{attr}_{\texttt{a}} : a \mid a \in Aspect_{act}\ \}$

$Aspect_{act} = \mathbf{Aspect_{act}}(aom.oom, aom.wrl.\text{am}, c')$

$s = (\ c'.State_{and}, \ldots, c'.Trans \setminus \{\ \text{tr}(jp) \mid jp \in \mathbf{JP_{act}}\ \}, c'.\text{label})$

$$advSetElems = \text{merge}(\{\ \ \text{advSetElems}(\ c'.name, aom, c', jp, \text{tr}(jp).src, \text{tr}(jp).dst) \mid jp \in \mathbf{JP_{act}}\ \})$$

$$\text{tr}(jp) = t \in c'.s,\ t.act = c'.s.\text{label}(jp)$$

$$\mathbf{JP_{act}} = \{\ jp \mid\ jp \in \mathbf{JP_{act}}(c) \wedge (c, jp) \in \text{Domain}(aom.wrl.\text{am}) \wedge aom.wrl.\text{am}(c, jp) \neq \emptyset\ \}$$

$\text{modCore}_{\text{ev}}$:**AOM** $\times$ **Class** $\rightarrow$ **Class**. Maps a core class c in AO model *aom* to a modified version of the core. Core data and behaviour are modified to prepend advised *event* join points with *before* advice. The data modification is the addition of references to aspects that advise event join points of the core, consume flags for the consumption of advised event join points by some *before* advice, and place-holders for arguments of events that trigger an advised event join point. The behavioural modification is the splitting of the core statechart into two concurrent regions:

- The *core* region rooted in $\texttt{st}_{\texttt{core}}$ that contains a modified version of the original

core statechart, where each transition $tr \in \mathbf{Tr}(jp)$ that may be triggered upon occurrence of an advised event join point jp is replaced with a set of transitions $\mathbf{Tr}'(jp, tr)$ that *delay* jp by a microstep.

- The *advice* region rooted in $\texttt{st}_{\texttt{adv}}$ that contains a single state $\texttt{st}_{\texttt{idle}}$ with self transitions $\mathrm{tr}_{\mathrm{adv}}(jp)$, triggered by advised event join points jp, whose actions implement *before* advice on the join points. The resulting behaviour is that upon occurrence of an advised join point, the corresponding self-transition in the *advice* region is taken (causing the corresponding *before* advice to execute) concurrently with the *delay* microstep of transitions $\mathbf{Tr}'$ in the *core* region. In the next microstep, the join point proceeds if it has not been consumed by some *before* advice.

$\mathrm{modCore}_{\mathrm{ev}}(aom, c) \stackrel{\mathrm{df}}{=}$

$(c.name, c.Attr \ \cup \ Attr, s)$

```
// References to aspects, join point consume flag,
// and event argument place-holders
```

$$Attr = \ \{\ \texttt{attr}_{\texttt{a}} : a \mid a \in Aspect_{ev}\ \} \ \cup$$

$$\{\ \texttt{attr}_{\texttt{jp\_con}} : \texttt{bool} \mid jp \in \mathbf{JP}_{\mathbf{ev}}\ \} \ \cup$$

$$\{\ attr \in Attr_{e\_arg} \mid e = jp.e \wedge jp \in \mathbf{JP}_{\mathbf{ev}}\ \}$$

$$Attr_{e\_arg} = \{\ \texttt{attr}_{\texttt{e\_arg}} : arg.type \mid arg \in e.Args\ \}$$

// Concurrent regions

$$s = (\ c.s.State_{and} \cup \{\ \mathtt{st}_{\mathtt{top}},\ \mathtt{st}_{\mathtt{idle}}\ \} \cup$$

$$\{\ \mathtt{st}_{\mathtt{tr}} \mid tr \in \mathbf{Tr}(jp) \wedge jp \in \mathbf{JP_{ev}}\ \},$$

$$c.s.State_{or} \cup \{\ \mathtt{st}_{\mathtt{core}},\ \mathtt{st}_{\mathtt{adv}}\ \},$$

$$\searrow',\ \mathrm{ini}',\ c.s.Signal,\ c.s.Call,\ Trans',\ c.s.label)$$

$$\searrow' = \ c.s. \searrow \setminus \{\ (\mathtt{root}, st) \in c.s. \searrow\ \} \cup$$

$$\{\ (\mathtt{st}_{\mathtt{core}}, st) \mid (\mathtt{root}, st) \in c.s. \searrow\ \} \cup$$

$$\{\ (\mathtt{root}, \mathtt{st}_{\mathtt{top}}),\ (\mathtt{st}_{\mathtt{top}}, \mathtt{st}_{\mathtt{core}}),\ (\mathtt{st}_{\mathtt{top}}, \mathtt{st}_{\mathtt{adv}}),\ (\mathtt{st}_{\mathtt{adv}}, \mathtt{st}_{\mathtt{idle}})\ \} \cup$$

$$\{\ (\mathrm{parent}(tr), \mathtt{st}_{\mathtt{tr}}) \mid tr \in \mathbf{Tr}(jp) \wedge jp \in \mathbf{JP_{ev}}\ \}$$

$$\mathrm{ini}' = \ c.s.\mathrm{ini} \setminus \{\ \mathtt{root} \mapsto st \in c.s.\mathrm{ini}\ \} \cup$$

$$\{\ \mathtt{st}_{\mathtt{core}} \mapsto st \mid \mathtt{root} \mapsto st \in c.s.\mathrm{ini}\ \} \cup$$

$$\{\ \mathtt{root} \mapsto \mathtt{st}_{\mathtt{top}},\ \mathtt{st}_{\mathtt{adv}} \mapsto \mathtt{st}_{\mathtt{idle}}\ \}$$

// Transition replacement/addition

$$Trans' = \ c.s.Trans \setminus \{\ tr \in \mathbf{Tr}(jp) \mid jp \in \mathbf{JP_{ev}}\ \} \cup$$

$$\{\ tr' \in \mathbf{Tr}'(jp, tr) \mid jp \in \mathbf{JP_{ev}} \wedge tr \in \mathbf{Tr}(jp)\ \} \cup$$

$$\{\ \mathrm{tr}_{\mathrm{adv}}(jp) \mid jp \in \mathbf{JP_{ev}}\ \}$$

// To be replaced

$$\mathbf{Tr}(jp) = \{\ tr \in c.s.Trans \mid \exists \sigma \in \Sigma_i,\ tr.src \in \sigma \wedge tr.e = jp.e\ \}$$

$$\Sigma_i(jp) = \{\ \sigma_i \in \text{valid configurations of } c.s \mid\ jp.\sigma \subseteq \sigma_i\ \}$$

// Replacement

$\mathbf{Tr}'(jp, tr) =$

$\{ \quad tr.src \xrightarrow{tr.e} \mathtt{st}_{\mathtt{tr}},$

$\quad \mathtt{st}_{\mathtt{tr}} \xrightarrow{[tr.g_{jp.e.Args \leftarrow Attr_{e\_args}} \ \wedge \ !\mathtt{attr}_{\mathtt{jp\_con}}]/tr.act_{jp.e.Args \leftarrow Attr_{e\_args}}} tr.dst,$

$\quad \mathtt{st}_{\mathtt{tr}} \xrightarrow{[!tr.g_{jp.e.Args \leftarrow Attr_{e\_args}} \ \vee \ \mathtt{attr}_{\mathtt{jp\_con}}]} tr.src \ \}$

// New advice transition

$\mathrm{tr}_{\mathrm{adv}}(jp) = \mathtt{st}_{\mathtt{idle}} \xrightarrow{jp.e[jp.\sigma \subseteq \mathtt{this}.\sigma]/act} \mathtt{st}_{\mathtt{idle}}$

$act = \quad \mathtt{attr}_{\mathtt{jp\_con}} := \mathtt{false};\ Attr_{e\_args} := jp.e.Args;$

$\qquad \mathrm{advAction}(aom, c, jp)$

$Aspect_{ev} = \mathbf{Aspect_{ev}}(aom.oom, aom.wrl.\mathrm{am}, c')$

$\mathbf{JP_{ev}} = \{ \ jp \mid \quad jp \in \mathbf{JP_{ev}}(c) \wedge (c, jp) \in \mathrm{Domain}(aom.wrl.\mathrm{am}) \ \wedge$

$\qquad\qquad aom.wrl.\mathrm{am}(c, jp) \neq \emptyset \ \}$

advAction :$\mathbf{AOM} \times \mathbf{Class} \times \mathbf{JP_{ev}}(c) \rightarrow Action$. Maps the set of *before* advice on the event core join point (c, jp) of AO model aom to a compound action that implements the advice set. Note that seq(Act) is the sequential composition of actions in Act. In Act is *ordered*, the order is preserved in the sequential composition, and if it is unordered, the order of composition is arbitrary.

$\mathrm{advAction}(aom, c, jp) \stackrel{\mathrm{df}}{=}$

$\mathrm{seq}(\{ \ treeAction(a, adv) \mid (a, adv) \in as_{bef} \ \})$

$\mathrm{treeAction}(a, adv) \stackrel{\mathrm{df}}{=}$

$\texttt{if}(!\texttt{attr}_{\texttt{jp\_con}})$ {

seq({ advAction(a, adv, ch) $\mid ch \in children(adv.root)$ })

}

nodeAction$(a, adv, n) \stackrel{\text{df}}{=}$

$$\begin{cases} \texttt{if}(n.\sigma \subseteq \texttt{attr}_{\texttt{a}}.\sigma)\ \{ & \\ \quad \texttt{attr}_{\texttt{a}}.ch.act;\ \text{seq}(\{\ \text{advAction}(a, adv, ch)\ \mid ch \in children(n)\ \}) & \\ \} & n \in adv.N_{act} \\ \texttt{attr}_{\texttt{jp\_con}} := \texttt{true} & n \in adv.N_{con} \end{cases}$$

$as = aom.wrl.\text{am}(c, jp)$

modObjects :**AOM** $\rightarrow \mathcal{P}$(**Object**). Maps AO model, *aom*, to a set of objects that is a modification of the AO model's objects as prescribed by its WRL. This involves setting initial aspect references of instances of core classes whose action join points are advised, to the appropriate aspect instance as prescribed by the *object map* of the AO model's WRL.

modObjects$(aom) \stackrel{\text{df}}{=}$

$aom.oom.O \setminus CoreObj \ \cup\ \mathbf{CoreObj'}$

$\mathbf{CoreObj'} = \{\ (oc.name,\ oc.c,\ oc.ini\ \cup\ \text{ini}'(oc)) \mid oc \in CoreObj\ \}$

$\text{ini}'(oc) = \{\ \texttt{attr}_{\texttt{a}} \mapsto oa \mid a \in Aspect(oc.c) \wedge wrl.om(oc, a) = oa\ \}$

$$CoreObj = \mathbf{CoreObj}(aom.oom, aom.wrl.\text{am})$$

$$Core = \mathbf{Core}(aom.oom, aom.wrl.\text{am})$$

$$Aspect(c) = \mathbf{Aspect}(aom.oom, aom.wrl.\text{am}, c)$$

Based on the above definitions, for the optimized weaving approach, the size increase of the woven OO model with respect to the OO component of the unwoven AO model is in $O(st_{ev} + st_{act} + attr_{ev} + attr_{act} + tr_{ev} + tr_{act})$ with redefinitions

$$st_{ev} = |C_{ev}| \times (jp_{ev} \times adv \times n + jp_{ev} \times trans)$$

$$tr_{ev} = |C_{ev}| \times (jp_{ev} \times adv^2 \times n + jp_{ev} \times trans)$$

where $trans$ is the number of transitions triggered by an event corresponding to an advised event join point.

3.5.1 Optimized Weaving without Concurrent Regions

To accommodate UML verification tools such as [33] that do not support concurrent regions in UML statecharts (i.e. *and* states that have more than one child), we present a variation on WP2 (WP2.1) described above that does not make use of concurrent regions. Once again, the compromise comes at a cost: we require that statecharts in the UML part of the AO model:

- Do not have concurrent regions.
- Do not have *conflicting* transitions (see Section 3.1).

The *FITEL* AO model and many other useful AO models (we believe) satisfy these conditions. We describe WP2.1 by its difference with the WP2; that is, the $\text{modCore}_{\text{ev}}$ function.

$\text{modCore}_{\text{ev}}$:**AOM** $\times$ **Class** $\rightarrow$ **Class**. Maps a core class c in AO model *aom* to a modified version of the core. Core data and behaviour are modified to prepend advised *event* join points with *before* advice. The data modification is the same as in WP2, and the behavioural modification is different only in that advice actions are embedded directly in the one microstep delay of transitions $\mathbf{Tr}'$ rather than in a separate concurrent region. Restriction 1 above, ensures that exactly *one* member of a possible *set* of conflicting transitions can be triggered upon occurrence of an advised event join point, and restriction 2 ensures that this set has only one member. So it is possible to precisely determine the transition in the core statechart that advice actions should be embedded in.

$\text{modCore}_{\text{ev}}(aom, c) \stackrel{\text{df}}{=}$

$(c.name, c.Attr \ \cup \ Attr, s)$

```
// References to aspects, join point consume flag,
// and event argument place-holders
```

$$Attr = \{\ \texttt{attr}_{\texttt{a}} : a \mid a \in Aspect_{ev}\ \} \cup$$
$$\{\ \texttt{attr}_{\texttt{jp\_con}} : \texttt{bool} \mid jp \in \mathbf{JP_{ev}}\ \} \cup$$
$$\{\ attr \in Attr_{e\_arg} \mid e = jp.e \wedge jp \in \mathbf{JP_{ev}}\ \}$$
$$Attr_{e\_arg} = \{\ \texttt{attr}_{\texttt{e\_arg}} : arg.type \mid arg \in e.Args\ \}$$

// Behaviour modifications

$$s = (\ c.s.State_{and} \cup \{\ \texttt{st}_{\texttt{tr}} \mid tr \in \mathbf{Tr}(jp) \wedge jp \in \mathbf{JP_{ev}}\ \}, c.s.State_{or},$$
$$\searrow' \cup \{\ (\text{parent}(tr), \texttt{st}_{\texttt{tr}}) \mid tr \in \mathbf{Tr}(jp) \wedge jp \in \mathbf{JP_{ev}}\ \},$$
$$\ldots,\ Trans',\ c.s.label)$$

// Transition replacement

$$Trans' = c.s.Trans \setminus \{\ tr \in \mathbf{Tr}(jp) \mid jp \in \mathbf{JP_{ev}}\ \} \cup$$
$$\{\ tr' \in \mathbf{Tr'}(jp, tr) \mid jp \in \mathbf{JP_{ev}} \wedge tr \in \mathbf{Tr}(jp)\ \} \cup$$
$$\{\ \text{tr}_{\text{adv}}(jp) \mid jp \in \mathbf{JP_{ev}}\ \}$$

// To be replaced

$$\mathbf{Tr}(jp) = \{\ tr \in c.s.Trans \mid \exists \sigma \in \Sigma_i,\ tr.src \in \sigma \wedge tr.e = jp.e\ \}$$

$$\Sigma_i(jp) = \{\ \sigma_i \in \text{valid configurations of } c.s \mid\ jp.\sigma \subseteq \sigma_i\ \}$$

// Replacement

$$\mathbf{Tr'}(jp, tr) =$$
$$\{\ tr.src \xrightarrow{tr.e/\text{act}(tr,jp)} \texttt{st}_{\texttt{tr}},$$
$$\texttt{st}_{\texttt{tr}} \xrightarrow{[tr.g_{jp.e.Args \leftarrow Attr_{e\_args}} \wedge\ !\texttt{attr}_{\texttt{jp\_con}}]/tr.act_{jp.e.Args \leftarrow Attr_{e\_args}}} tr.dst,$$
$$\texttt{st}_{\texttt{tr}} \xrightarrow{[!tr.g_{jp.e.Args \leftarrow Attr_{e\_args}} \vee\ \texttt{attr}_{\texttt{jp\_con}}]} tr.src\ \}$$

$$\begin{aligned}\mathrm{act}(tr, jp) = \;& \mathtt{attr}_{\mathtt{jp\_con}} := \mathtt{false};\ Attr_{e\_args} := jp.e.Args; \\ & \mathrm{advAction}(aom, c, jp)\end{aligned}$$

$$Aspect_{ev} = \mathbf{Aspect_{ev}}(aom.oom, aom.wrl.\mathrm{am}, c')$$

$$\begin{aligned}\mathbf{JP_{ev}} = \{\ jp \mid \;& jp \in \mathbf{JP_{ev}}(c) \wedge (c, jp) \in \mathrm{Domain}(aom.wrl.\mathrm{am}) \wedge \\ & aom.wrl.\mathrm{am}(c, jp) \neq \emptyset\ \}\end{aligned}$$

Based on the definition above, for WP2.1, the size increase of the woven OO model with respect to the OO component of the unwoven AO model is in $O(st_{ev} + st_{act} + attr_{ev} + attr_{act} + tr_{ev} + tr_{act})$ with redefinitions:

$$\begin{aligned}st_{ev} &= |C_{ev}| \times (jp_{ev} \times trans \times adv \times n + jp_{ev} \times trans) \\ tr_{ev} &= |C_{ev}| \times (jp_{ev} \times trans \times adv^2 \times n + jp_{ev} \times trans)\end{aligned}$$

Chapter 4

Case Studies

4.1 Feature Interactions in Telephony Systems

Our first case study is a well-known example from the domain of feature interactions in telephony systems adopted from [22]. Here, the telephony system is comprised of a set of users (telephone receivers), a network switch, and a set of control software modules (one per user). All communication between users and control software modules goes through the switch. In its basic form, a control software module manages a simple connection between its user and another party by communicating with its user and the other party's control software module. In modern telephony systems, users can enhance their control software module by subscribing to various features such as *call forwarding* (CF), which forwards incoming calls to a third party, and *originating call*

screening (OCS), which prevents outgoing calls to users on a screening list. In some instances, features fail to co-exist: i.e. features interfere with one another's operation or in other words, they interact. As an example, imagine that user 1 has subscribed to OCS, with a screen on user 3, and that user 2 has subscribed to CF, with all calls forwarded to 3. If 1 calls 2, and the call is forwarded to 3 due to 2's CF, then 1's OCS is compromised, and if the call is not forwarded due to 1's OCS, 2's CF is compromised. Hence the two features interact. We will show our process can be used to detect this interaction. The case study will be referred to as *FITEL* for *feature interactions in telephony systems*. Note that in this case study and the next (Section 4.2) the weaving was performed by hand.

4.1.1 AO model

UML

The UML part of *FITEL*'s AO model is shown in the following figures:

- Class names and data are shown graphically in Figure 4.1.
- Class statecharts are shown graphically in Figure 4.2 and Figure 4.3 (note that $src \xrightarrow{stm_1;\ldots;stm_n} dst$ is an abbreviation for $src \xrightarrow{stm_1} i_1 \ldots i_{n-1} \xrightarrow{stm_n} dst$), and their event receptions are shown in abbreviated notation in Figure 4.4.
- The initial instantiation is shown in Figure 4.5.

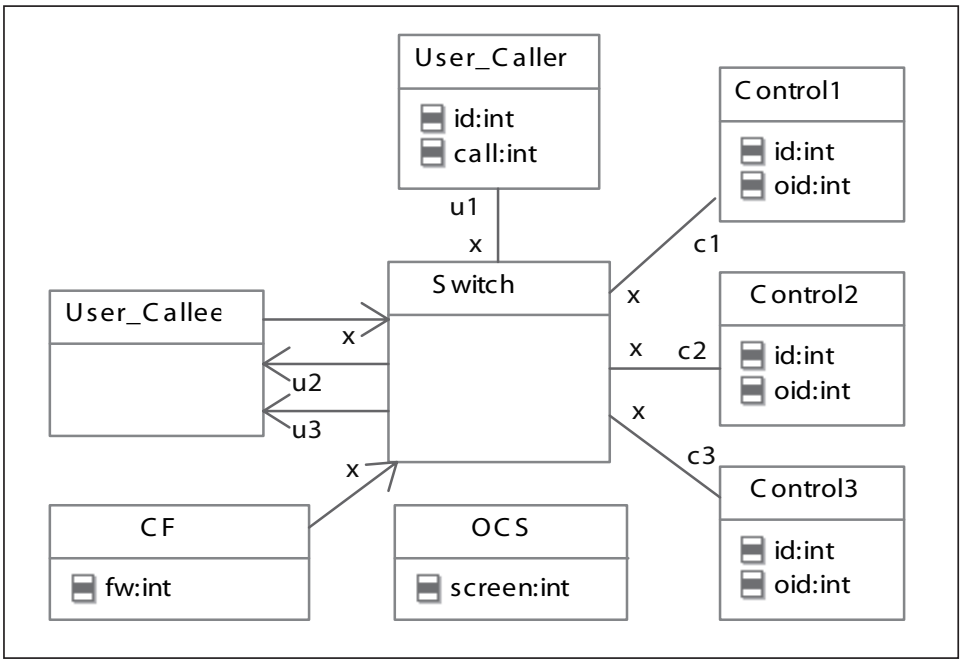

Figure 4.1: *FITEL* OO model class names and data

To see the mapping between the set-based notation for classes presented in Section 3.1 and the alternative notation used in the figures, compare the set-based representation of class `CF` below, with that of Figure 4.1, Figure 4.2, and Figure 4.4.

$$cf = (CF, Attr, s)$$

$$Attr = \{\ (fw,\ \texttt{int}),\ (x,\ \texttt{Switch})\ \}$$

$$s = \ (\{\ \texttt{idle}\ \},\ \{\ \texttt{root}\ \},\ \{\ (\texttt{root},\ \texttt{idle})\ \},\ [\texttt{root} \mapsto \texttt{idle}],$$

$$\{\ (\ \texttt{iring},\ \{\ (\texttt{oid},\ \texttt{int})\ \}\ )\ \},\ \emptyset,$$

$$\{\ \texttt{idle} \xrightarrow{\texttt{iring/x!oring(oid,fw)}} \texttt{idle}\ \},\ \emptyset)$$

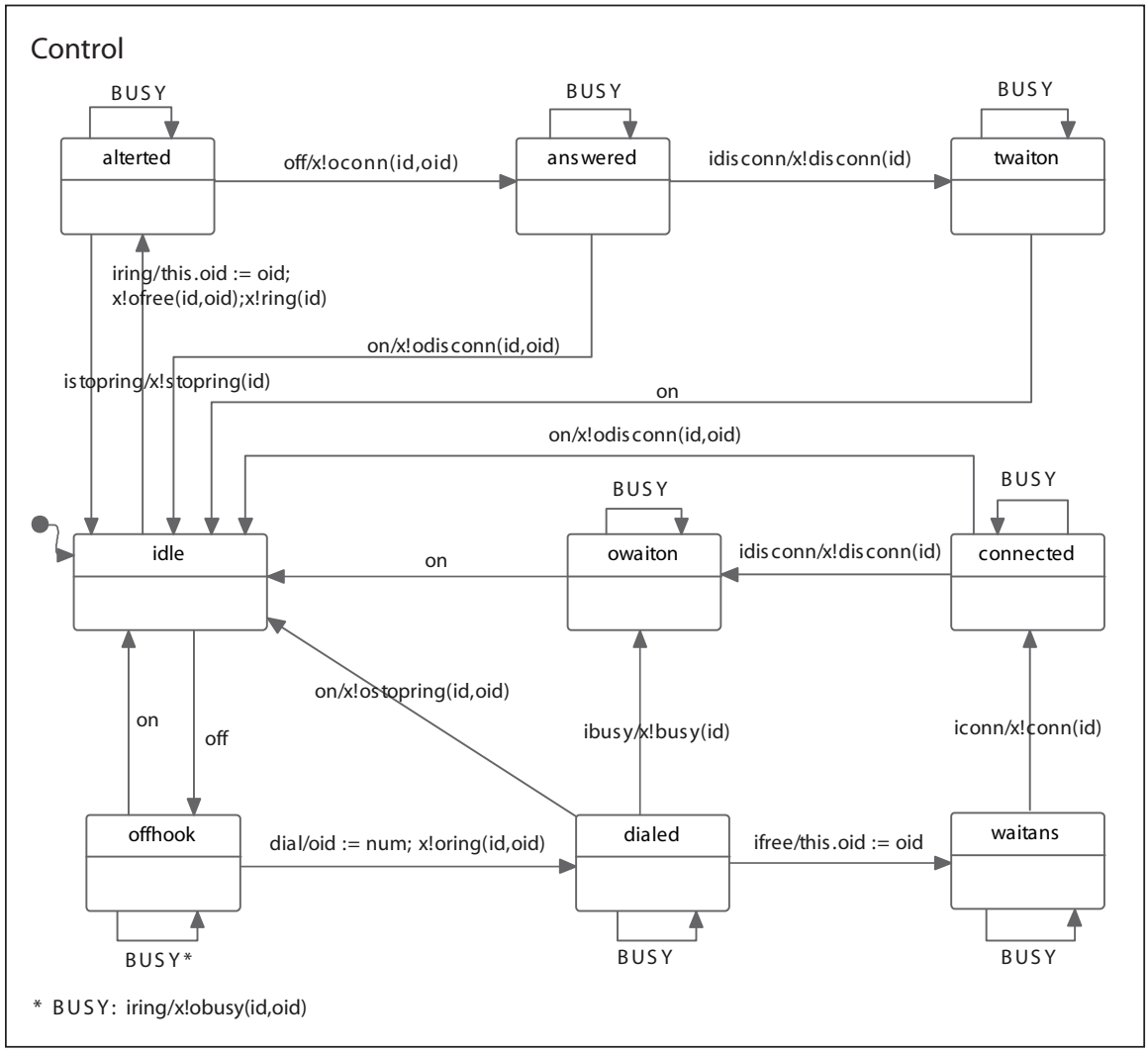

Figure 4.2: *FITEL* OO model statecharts for Control1, Control2, and Control3

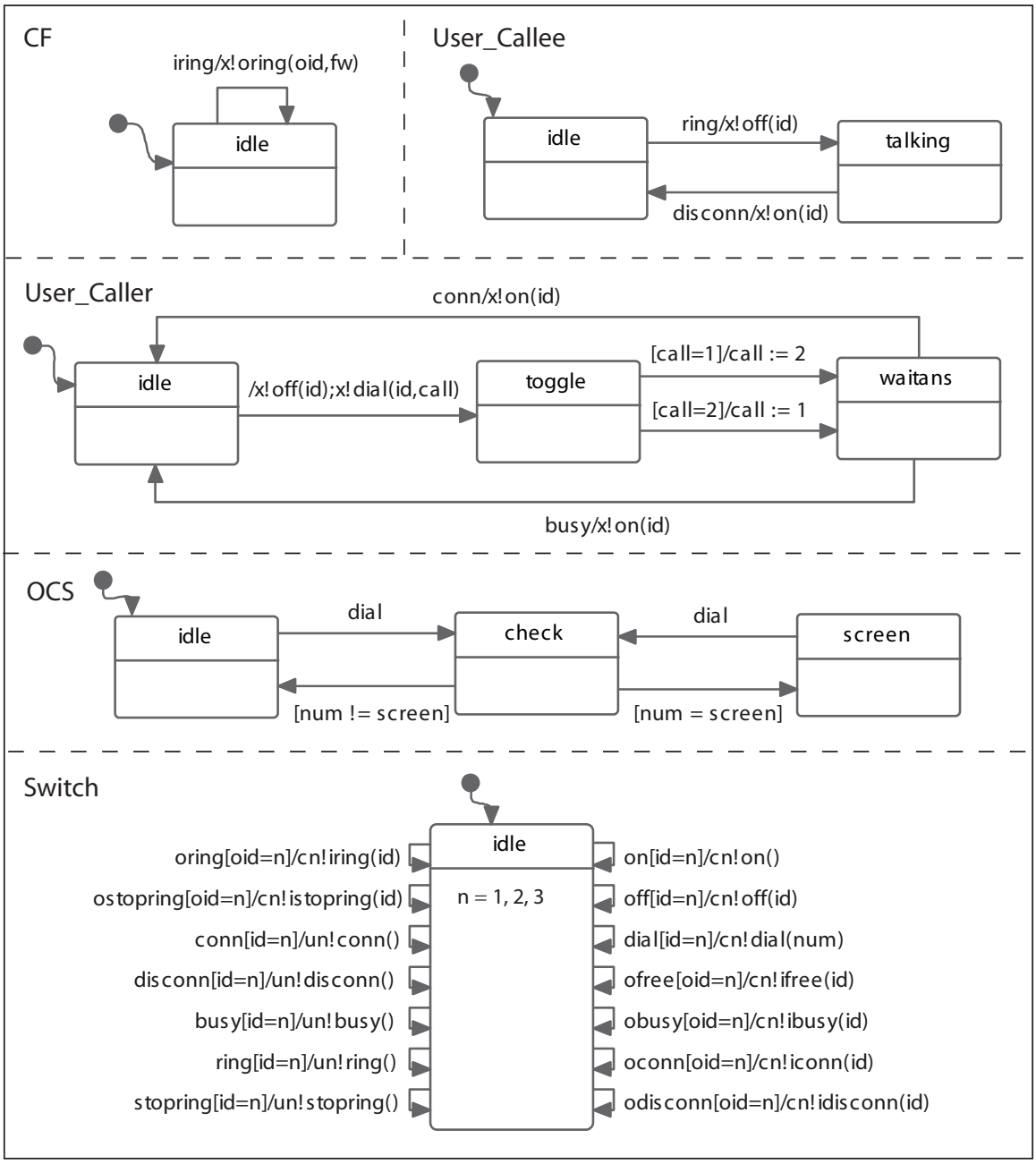

Figure 4.3: *FITEL* OO model statecharts for CF, User_Caller, User_Callee, OCS, and Switch

`Control1 − 3`
$Signal = \{$ $on(), off(), dial(num : int)$ // from user $ifree(oid : int), iconn(oid : int), idisconn(oid : int), ibusy(oid : int),$ $iring(oid : int), istopring(oid : int)$ // from control $\}$ $Call = \emptyset$

`User_Caller − Callee`
$Signal = \{$ $conn(), disconn(), busy(), ring(), stopring()$ $\}$ $Call = \emptyset$

`CF`
$Signal = \emptyset,\ Call = \{$ $ring(oid : int)$ $\}$

`OCS`
$Signal = \emptyset,\ Call = \{$ $dial(num : int)$ $\}$

`Switch`
$Signal =$ $\{$ $on(id : int), off(id : int), dial(id : int, num : int)$ // from user $ofree(id : int, oid : int), oconn(id : int, oid : int), odisconn(id : int, oid : int),$ $obusy(id : int, oid : int), oring(id : int, oid : int)$ $ostopring(id : int, oid : int)$ // from control (to control) $conn(id : int), disconn(id : int), busy(id : int),$ $ring(id : int), stopring(id : int)$ // from control (to user) $\}$ $Call = \emptyset$

Figure 4.4: *FITEL* OO model events

$$
\begin{array}{l}
(\texttt{os1}, \texttt{Switch}, [\ c1 \mapsto oc1, c2 \mapsto oc2, c3 \mapsto oc3, \\
\qquad\qquad\qquad u1 \mapsto ou1, u2 \mapsto ou2, u3 \mapsto ou3]), \\
(\texttt{oc1}, \texttt{Control1}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 1]), \\
(\texttt{oc2}, \texttt{Control2}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 2]), \\
(\texttt{oc3}, \texttt{Control3}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 3]), \\
(\texttt{ou1}, \texttt{User\_Caller}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 1, \texttt{call} \mapsto 2]), \\
(\texttt{ou2}, \texttt{User\_Callee}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 2]), \\
(\texttt{ou3}, \texttt{User\_Callee}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 3]), \\
(\texttt{ocf}, \texttt{CF}, [\texttt{x} \mapsto \texttt{os1}, \texttt{fw} \mapsto 3]), \\
(\texttt{oocs}, \texttt{OCS}, [\texttt{x} \mapsto \texttt{os1}, \texttt{screen} \mapsto 3])\}
\end{array}
$$

Figure 4.5: *FITEL* OO model initial instantiation

WRL

The WRL part of *FITEL*'s AO model is shown in Figure 4.6, and is explained below:

- *Weaving rule 1:* Before core instance `oc1 : Control1` can process event `dial(num : int)` when it is in state `offhook`, aspect instance `oocs : OCS` processes the event. If as a result, `oocs`'s statechart lands in state `screen`, the event is consumed (preventing the core from seeing it). Otherwise, `oc1` processes the event as usual.

- *Weaving rule 2:* Before core instance `oc2 : Control2` can process event `iring(oid : int)` regardless of its current state, aspect instance `ocf : CF` processes the event. Then, unconditionally, the event is consumed.

Figure 4.7 shows *FITEL*'s WRL in the alternative syntax of Figure 3.1.

Advice mapping:
$(c1, jp1) \mapsto as1$
$c1 =$ `control1`
$jp1 = (\{$ `offhook` $\}, \texttt{dial}(\texttt{num} : \texttt{int}))$
$as1 = as1_{bef} = \{$ (`OCS`, `adv1`) $\}$
$adv1 =$

```
root
  ({idle},dial(num))
    ({idle}, skip)
    ({screen}, skip)
      consume
  ({screen},dial(num))
    ({idle}, skip)
    ({screen}, skip)
      consume
```

Object assignment:
$\{$ (`oc1`, `OCS`) $\mapsto$ `oocs` $\}$

Weaving Rule 1

Advice mapping:
$(c2, jp2) \mapsto as2$
$c2 =$ `control2`
$jp2 = (\{$ `root` $\}, \texttt{iring}(\texttt{oid} : \texttt{int}))$
$as2 = as2_{bef} = \{$ (`CF`, `adv2`) $\}$
$adv2 =$

```
root
  ({idle},iring(oid))
    consume
```

Object assignment:
$\{$ (`oc2`, `CF`) $\mapsto$ `ocf` $\}$

Weaving Rule 2

Figure 4.6: *FITEL* WRL

```
Aspect OCS
  Core Control1
    before ({offhook},dial)
      ({idle},dial(num))
        ({idle}, skip)
        ({screen}, skip)
          consume
      ({screen},dial(num))
        ({idle}, skip)
        ({screen}, skip)
          consume
    ObjectMap
      oc1 -> oocs
```

Weaving Rule 1

```
Aspect CF
  Core Control2
    before ({root},iring)
      ({idle},iring(oid))
        consume
    ObjectMap
      oc2 -> ocf
```

Weaving Rule 2

Figure 4.7: *FITEL* WRL in alternative syntax

4.1.2 Woven OO model

WP1

FITEL's woven OO model (using WP1) is shown in the figures below:

- Class names and data are shown in Figure 4.8.
- Statecharts of proxies for core classes `Control1` and `Control2` are shown in Figure 4.9 and Figure 4.10 respectively (statecharts of other classes are the same as in the unwoven AO model), and event receptions of all statecharts are shown in Figure 4.11. Note that some statechart transitions added by the weaving process may be *unreachable* in the sense that they are not enabled for any configuration of the woven model. The woven model can be simplified by removing such transitions. In *FITEL*'s woven model (using WP1), the unreachable transitions are: transitions from `st_adv1_i`, `st_n1`, and `st_n2` to `st_adv1_f` in `PControl1`; and transition from `st_adv2_i` to `st_adv2_f` and the transition from `st_adv2_f` to `idle` guarded by `!jp2_con` in `PControl2`.
- The initial instantiation is shown in Figure 4.12.

WP2

FITEL's woven OO model (using WP2) is shown in the figures below:

- Class names and data are shown in Figure 4.13.

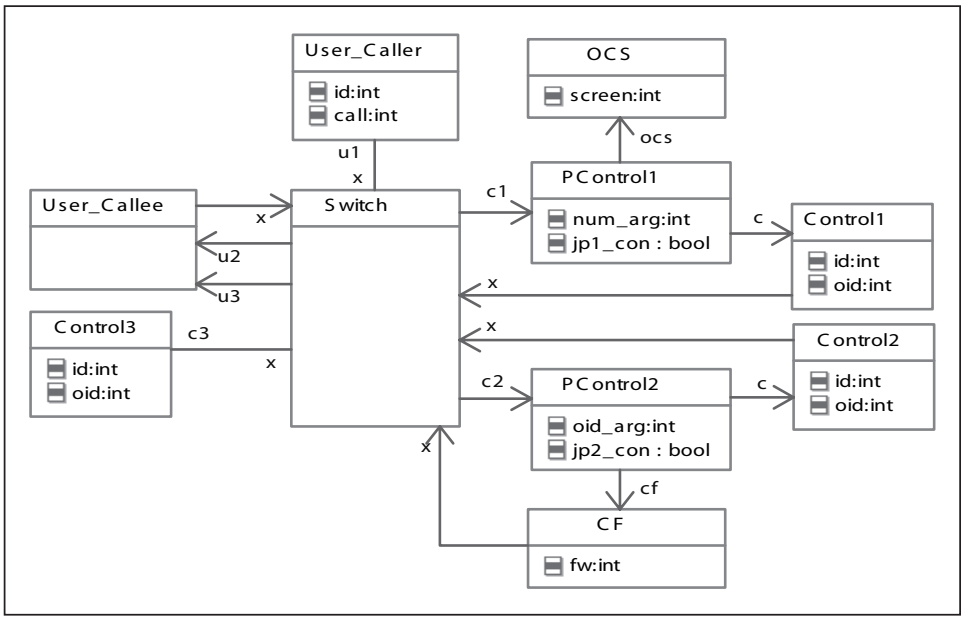

Figure 4.8: *FITEL* woven OO model (using WP1) data

- The modified portion of statecharts of core classes `Control1` and `Control2` are shown in Figure 4.14 and Figure 4.15 respectively (statecharts of other classes, as well as event receptions of all classes are the same as in the unwoven AO model).

- The initial instantiation is shown in Figure 4.16.

WP2.1

FITEL's woven OO model (using WP2.1) is shown in the figures below:

- Class names and data and the initial instantiation are the same as in WP2.

- The modified portion of statecharts of core classes `Control1` and `Control2` are

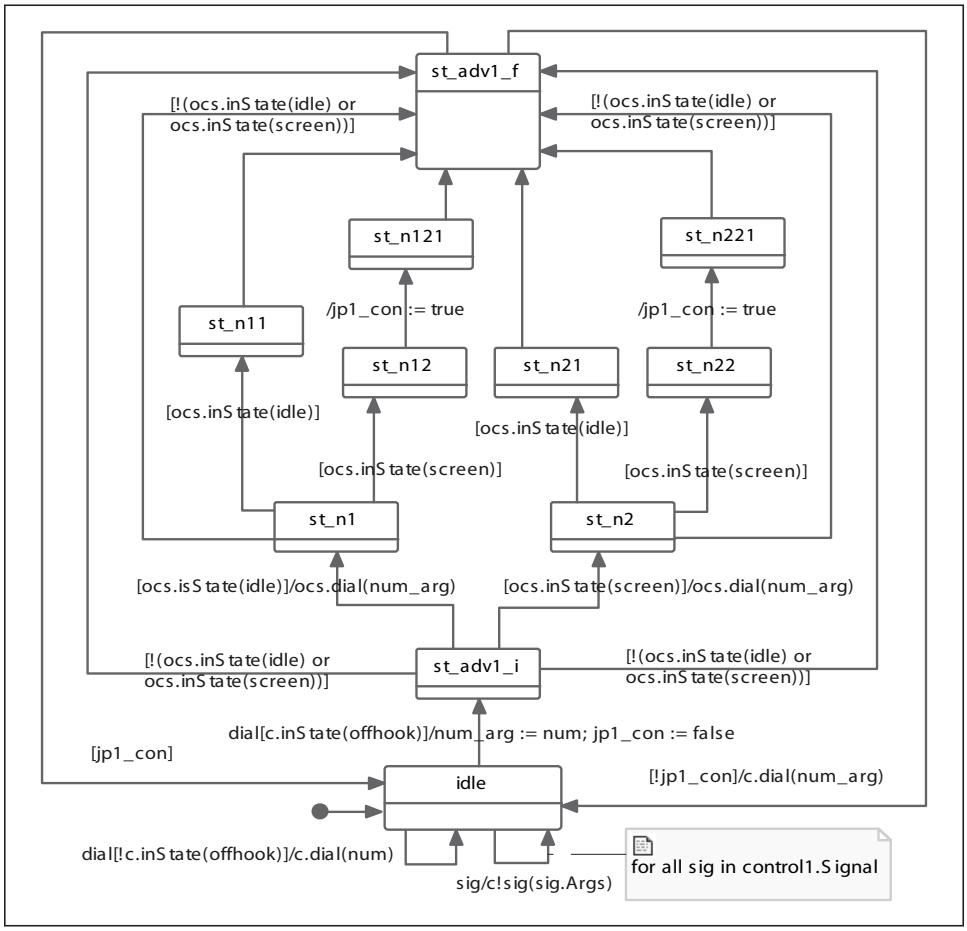

Figure 4.9: *FITEL* woven OO model (using WP1) statechart for `PControl1` (proxy of `Control1`)

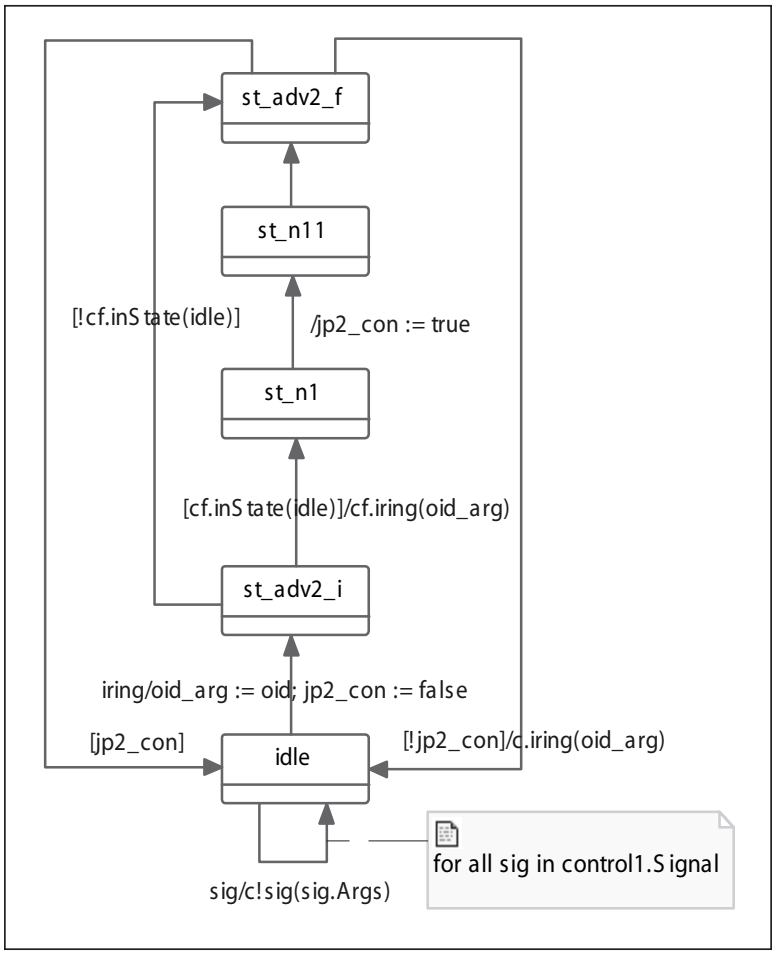

Figure 4.10: *FITEL* woven OO model (using WP1) statechart for `PControl2` (proxy of `Control2`)

`Control1`
$Signal = \{\ on(), off()$ // from user $ifree(oid : int), iconn(oid : int), idisconn(oid : int), ibusy(oid : int),$ $iring(oid : int), istopring(oid : int)$ // from control $\}$ $Call = \{\ dial(num : int)\ \}$

`Control2`
$Signal = \{\ on(), off(), dial(num : int)$ // from user $ifree(oid : int), iconn(oid : int), idisconn(oid : int), ibusy(oid : int),$ $istopring(oid : int)$ // from control $\}$ $Call = \{\ iring(oid : int)\ \}$

`PControl1` $-$ `2`
same as `Control1` $-$ `2` *of Figure 4.4*

`Control3`, `User_Caller` $-$ `Callee`, `Switch`
same as Figure 4.4

Figure 4.11: *FITEL* woven OO model (using WP1) events

$(\texttt{os1}, \texttt{Switch}, [\ c1 \mapsto opc1, c2 \mapsto opc2, c3 \mapsto oc3,$
$u1 \mapsto ou1, u2 \mapsto ou2, u3 \mapsto ou3]),$
$(\texttt{opc1}, \texttt{PControl1}, [\texttt{c} \mapsto \texttt{oc1}, \texttt{ocs} \mapsto \texttt{oocs}, \texttt{num}_\texttt{a}\texttt{rg} \mapsto \texttt{0}, \texttt{jp1\_con} \mapsto \texttt{false}]),$
$(\texttt{opc2}, \texttt{PControl2}, [\texttt{c} \mapsto \texttt{oc2}, \texttt{cf} \mapsto \texttt{ocf}, \texttt{oid}_\texttt{a}\texttt{rg} \mapsto \texttt{0}, \texttt{jp2\_con} \mapsto \texttt{false}]),$
$\ldots$ (other initial instantiations in *FITEL*'s OO model, unchanged)$\}$

Figure 4.12: *FITEL* woven OO model (using WP1) initial instantiation

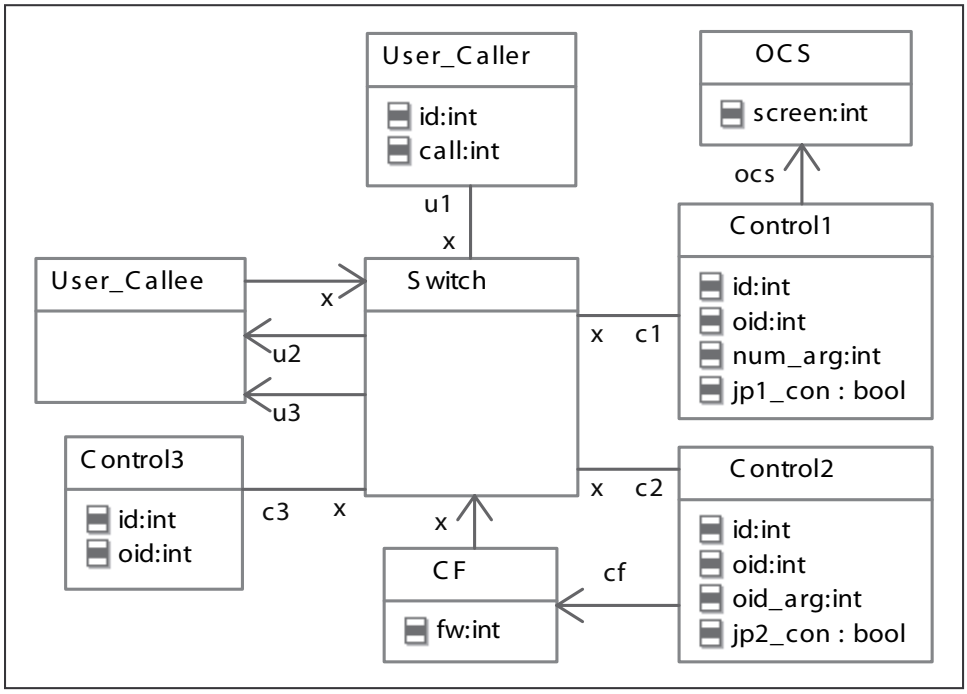

Figure 4.13: *FITEL* woven OO model (using WP2) data

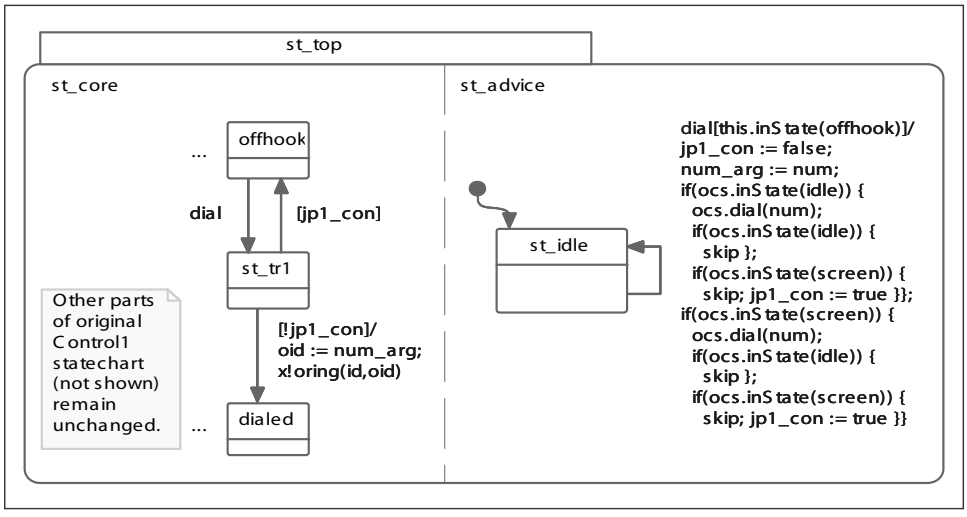

Figure 4.14: *FITEL* woven OO model (using WP2) statechart for `Control`

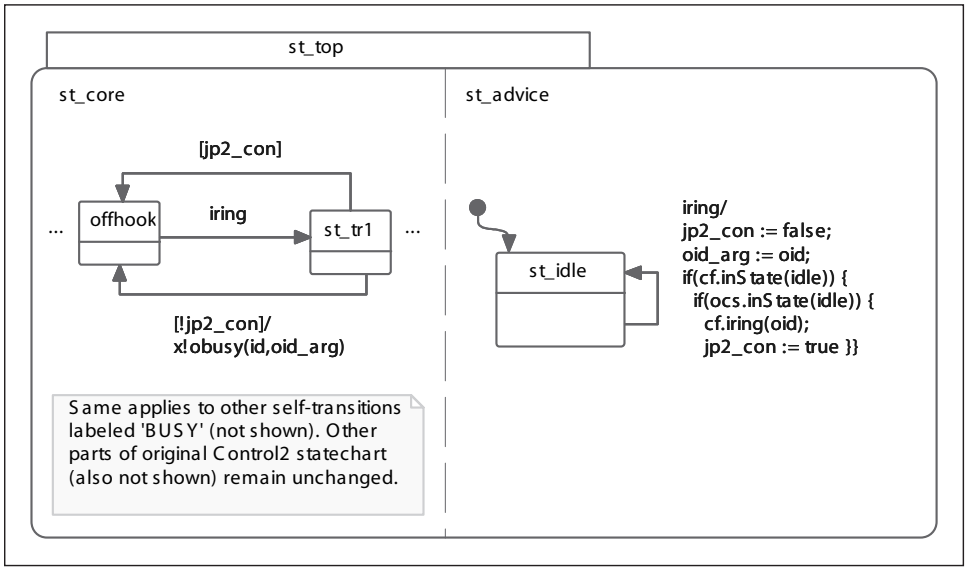

Figure 4.15: *FITEL* woven OO model (using WP2) statechart for `Control2`

$(\texttt{oc1}, \texttt{Control1}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 1, \texttt{ocs} \mapsto \texttt{oocs}, \texttt{num}_{\texttt{a}}\texttt{rg} \mapsto 0, \texttt{jp1\_con} \mapsto \texttt{false}]),$
$(\texttt{oc2}, \texttt{Control2}, [\texttt{x} \mapsto \texttt{os1}, \texttt{id} \mapsto 2, \texttt{cf} \mapsto \texttt{ocf}, \texttt{oid}_{\texttt{a}}\texttt{rg} \mapsto 0, \texttt{jp2\_con} \mapsto \texttt{false}]),$
. . . (other initial instantiations in *FITEL*'s OO model are unchanged)}

Figure 4.16: *FITEL* woven OO model (using WP2) initial instantiation

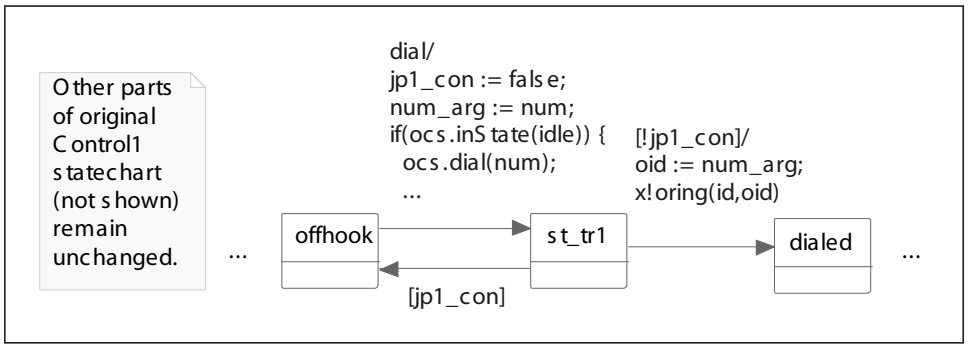

Figure 4.17: *FITEL* woven OO model (using WP2.1) statechart for *Control1*

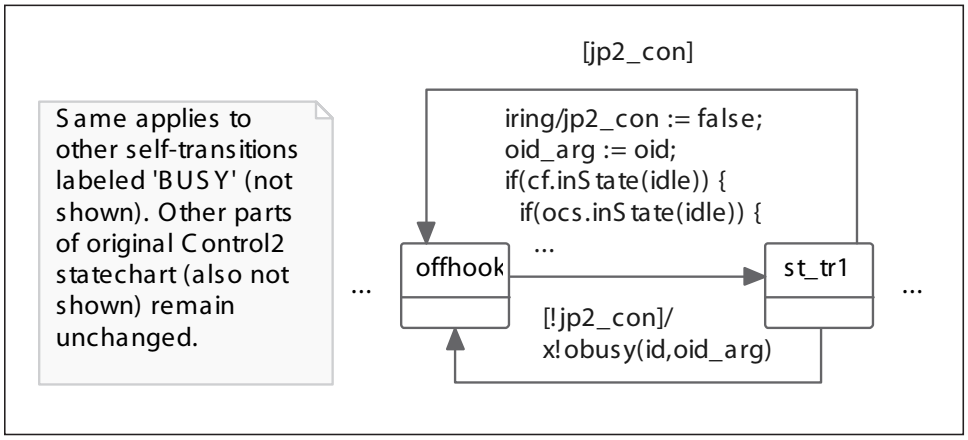

Figure 4.18: *FITEL* woven OO model (using WP2.1) statechart for *Control2*

shown in Figure 4.17 and Figure 4.18 respectively (statecharts of other classes, as well as event receptions of all classes are the same as in the unwoven AO model).

4.1.3 Reports

Analysis Report

The syntactic analysis of the *FITEL* AO model reveals nothing, while as we know, interactions do exist in this model. These are revealed in the verification report as we shall see in the next section.

Verification Report

We verified the following correctness property of *FITEL*'s woven OO model using IFx [33] as discussed in Section 1.3:

$$Prop_{ocs} = \text{`No connection from user 1 to user 3 is possible'}$$

$Prop_{ocs}$ is required by the OCS subscription of user 1. Its observer class behaviour is shown in Figure 4.19 (the class has no data). $Prop_{ocs}$ was verified once with only *Weaving Rule 1* (only weave OCS), and once with both *Weaving Rule 1 & 2* (weave both OCS and CF) of Figure 4.6 on a machine with 2GB of memory. Table 4.1 tabulates the results using weaving processes WP1 and WP2.1 (IFx does not yet support concurrent regions). The column *States* is the size of the state space of an IF model equivalent to *FITEL*'s woven model and is a measure of the woven model's verification complexity (using WP1 and with both OCS and CF woven, the state-space is too large to fit even in 2GB of memory). Note that $Prop_{ocs}$ is satisfied with

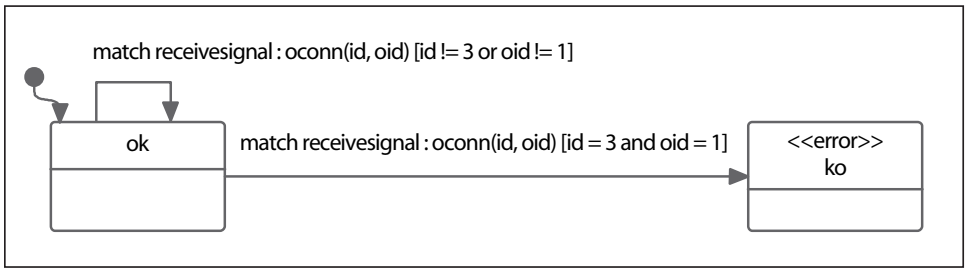

Figure 4.19: $Prop_{ocs}$ observer class behaviour

only OCS woven, but fails to satisfy with both OCS and CF woven: this indicates an interaction between these two features.

Table 4.1: *FITEL* verification results using IFx

	WP1		WP2.1	
	States	$Prop_{ocs}$	*States*	$Prop_{ocs}$
OCS	≈ 160000	✓	≈ 57000	✓
$OCS + CF$	550000+	–	≈ 120000	×

4.2 Interactions in an Electronic Commerce Shop

Our second case study is an adaption of the *e-commerce shop* example used to illustrate EAOP in [11]. Here we consider a trivial electronic commerce shop where a customer makes requests for purchasing products and is billed accordingly. The shop has two promotions in place: the regular *discount* promotion is that any purchase that exceeds some threshold t is awarded a pd percent price reduction from their purchase; the *bingo* promotion is that the n^{th} customer is awarded a pb percent price reduction from their purchase (and the promotion is repeated for the next n

customers). The shop has a *profiling* mechanism that allows the manager to monitor the number of discounts awarded to customers so far. Suppose the shop has a *price reduction limit* (PRL) policy that the total price reduction awarded to a customer for a given purchase shall not exceed some percentage pt. If $pd + pb > pt$ and the n^{th} makes a purchase that exceeds t, PRL is violated (i.e. the promotions interact). We will illustrate how our process can be used to detect this violation. This case study will be referred to as *ECOMM*.

4.2.1 AO model

UML

The UML part of *FITEL*'s AO model is shown in the following figures:

- Class names and data are shown graphically in Figure 4.20.
- Class statecharts (with action labels) are shown graphically in Figure 4.21, and their event receptions are shown in abbreviated notation in Figure 4.22. Note that the customer continually makes purchases of \$40.
- The initial instantiation is shown in Figure 4.23. Note that here, t = \$30, $n = 100$, $pd = \%10$, $pb = \%50$, and $pt = \%50$.

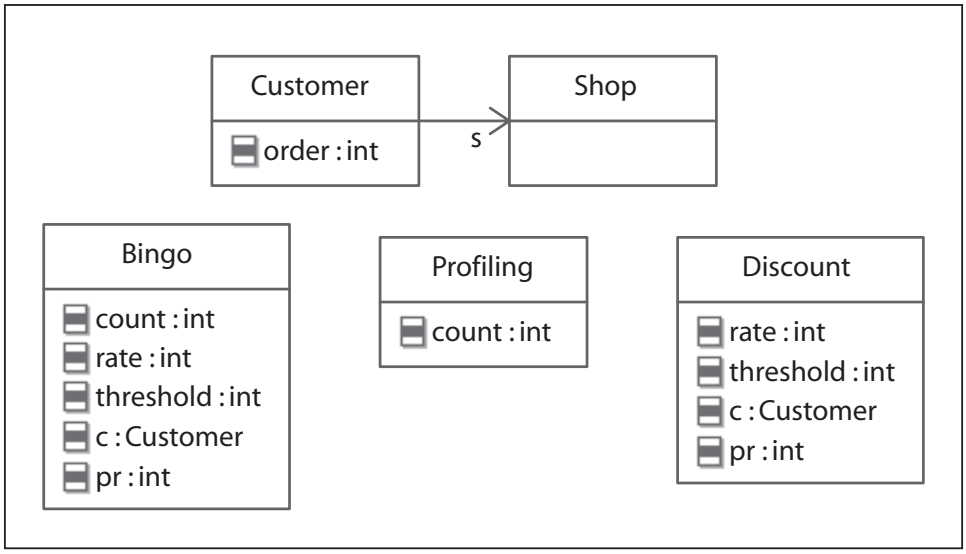

Figure 4.20: *ECOMM* OO model class names and data

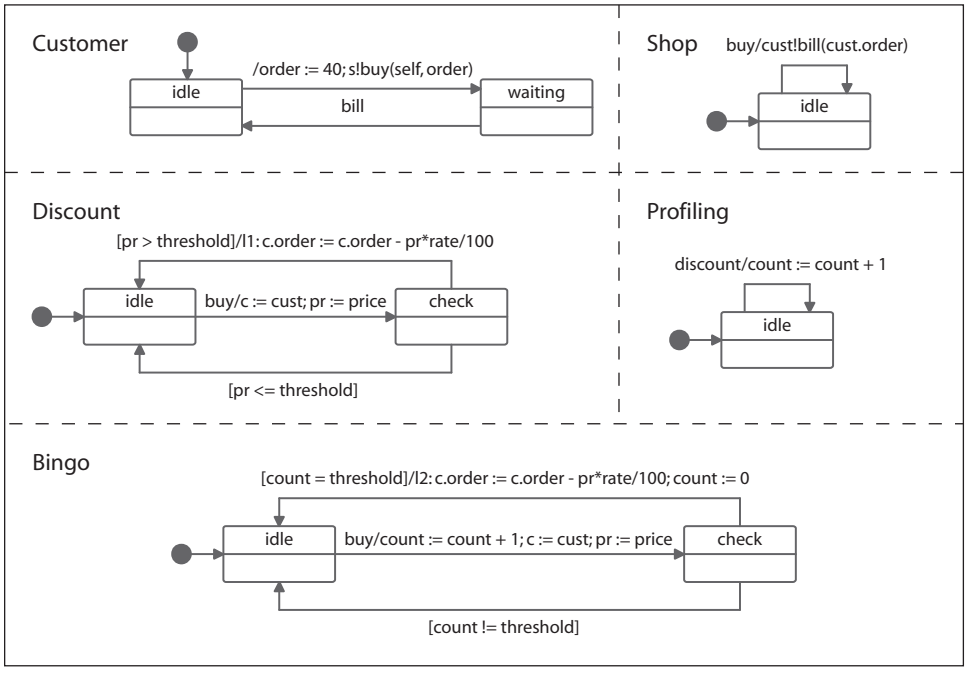

Figure 4.21: *FITEL* OO model statecharts (with action labels)

`Customer`
$Signal = \{\ bill(price : int)\ \},\ Call = \emptyset$

`Shop`
$Signal = \{\ buy(cust : Customer, price : int)\ \},\ Call = \emptyset$

`Discount`
$Signal = \emptyset,\ Call = \{\ buy(cust : Customer, price : int)\ \}$

`Bingo`
$Signal = \emptyset,\ Call = \{\ buy(cust : Customer, price : int)\ \}$

`Profiling`
$Signal = \emptyset,\ Call = \{\ buy(cust : Customer, price : int)\ \}$

Figure 4.22: *ECOMM* OO model events

$(\texttt{oc}, \texttt{Customer}, [\texttt{s} \mapsto \texttt{os}, \texttt{order} \mapsto 0]$
$(\texttt{os}, \texttt{Shop}, \emptyset),$
$(\texttt{od}, \texttt{Discount}, [\texttt{rate} \mapsto 10, \texttt{threshold} \mapsto 40, \texttt{c} \mapsto \texttt{null}, \texttt{pr} \mapsto 0]),$
$(\texttt{ob}, \texttt{Bingo}, [\texttt{count} \mapsto 0, \texttt{rate} \mapsto 50, \texttt{threshold} \mapsto 100, \texttt{c} \mapsto \texttt{null}, \texttt{pr} \mapsto 0]),$
$(\texttt{op}, \texttt{Profiling}, [\texttt{count} \mapsto 0])\}$

Figure 4.23: *ECOMM* OO model initial instantiation

WRL

The WRL part of *ECOMM*'s AO model is shown in Figure 4.24. Here, `Shop` is advised by `Discount` and `Bingo`, which themselves are advised by `Profiling` (this illustrates an *aspect of aspect* scenario). The weaving rules are explained below:

- *Weaving rule 1:* Before core instance `os : Shop` can process event `buy(cust : Customer, price : int)` when it is in state `idle`, first aspect instance `od : Discount` and then `ob : Bingo` process the event. Then unconditionally, `os` processes the event.

- *Weaving rule 2:* After the execution of action `l1` in core instance `od : Discount`, event `discount()` is dispatched to aspect instance `op : Profiling` if its statechart is in state `idle`.

- *Weaving rule 3:* After the execution of action `l2` in core instance `ob : Bingo`, event `discount()` is dispatched to aspect instance `op : Profiling` if its statechart is in state `idle`.

Figure 4.25 shows *FITEL*'s WRL in the alternative syntax of Figure 3.1.

4.2.2 Woven OO model

WP1

ECOMM's woven OO model (using WP1) is shown in the figures below:

Advice mapping:
$(c1, jp1) \mapsto as1$
$c1 =$ `Shop`
$jp1 = (\{$ `idle` $\}, \texttt{buy}(\texttt{cust} : \texttt{Customer}, \texttt{price} : \texttt{int}))$
$as1 = as1_{bef} =$ $\{$ $(\texttt{Discount}, \texttt{adv1}), (\texttt{Bingo}, \texttt{adv2})$ $\}$ and
$(\texttt{Discount}, \texttt{adv1}) > (\texttt{Bingo}, \texttt{adv2})$
$adv1 = adv2 =$

```
root
  ({idle}, buy(cust, price))
```

Object assignment:
$\{$ $(\texttt{os}, \texttt{Discount}) \mapsto \texttt{od}, (\texttt{os}, \texttt{Bingo}) \mapsto \texttt{ob}$ $\}$

Weaving Rule 1

Advice mapping:
$(c2, jp2) \mapsto as2$
$c2 =$ `Discount`
$jp2 = l1$
$as2 = as2_{aft} = \{$ $(\texttt{Profiling}, \texttt{adv3})$ $\}$
$adv3 =$

```
root
  ({idle},discount())
```

Object assignment:
$\{$ $(\texttt{od}, \texttt{Profiling}) \mapsto \texttt{op}$ $\}$

Weaving Rule 2

Advice mapping:
$(c3, jp3) \mapsto as3$
$c3 =$ `Bingo`
$jp3 = l2$
$as3 = as3_{aft} = \{$ $(\texttt{Bingo}, \texttt{adv4})$ $\}$
$adv4 =$

```
root
  ({idle},discount())
```

Object assignment:
$\{$ $(\texttt{ob}, \texttt{Profiling}) \mapsto \texttt{op}$ $\}$

Weaving Rule 3

Figure 4.24: *ECOMM* WRL

```
Aspect Discount
  Core Shop
    before ({idle},buy)
      ({idle},buy(cust,price))
    ObjectMap
      os -> od
```

Weaving Rule 1 (part 1)

```
Aspect Bingo
  Core Shop
    before ({idle},buy)
      ({idle},buy(cust,price))
    ObjectMap
      os -> ob
```

Weaving Rule 1 (part 2)

```
Aspect Profiling
  Core Discount
    after l1
      ({idle},discount())
    ObjectMap
      od -> op
  Core Bingo
    after l2
      ({idle},discount())
    ObjectMap
      ob -> op
```

Weaving Rules 2 and 3

Figure 4.25: *ECOMM* WRL in alternative syntax

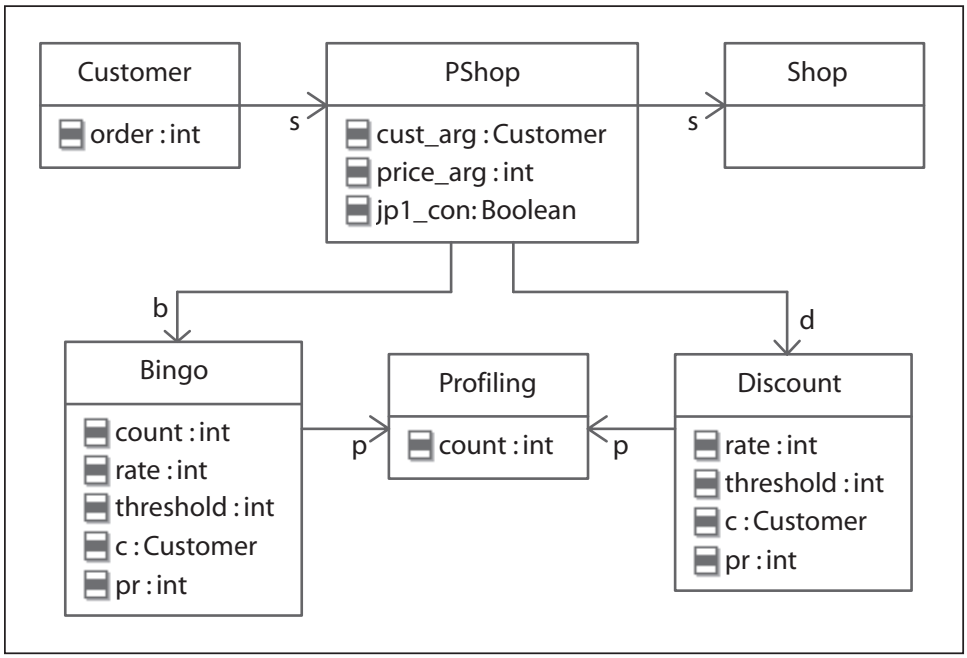

Figure 4.26: *ECOMM* woven OO model (using WP1) data

- Class names and data are shown in Figure 4.26.

- Statecharts of the proxy for core class `Shop` and modified core classes `Discount` and `Bingo` are shown in Figure 4.27 (statecharts of other classes are the same as in the unwoven AO model), and event receptions of all statecharts are shown in Figure 4.28. Note that the $\texttt{jp1}_\texttt{c}\texttt{on}$ attribute of `PShop` is not necessary due to the absence of *consume* nodes in advice. All transitions guarded by $!\texttt{jp1}_\texttt{c}\texttt{on}$, therefore are *unreachable* and can be removed. The same is true of all transitions guarded by `!s.inState(idle)`, `!d.inState(idle)`, or `!b.inState(idle)`.

- The initial instantiation is shown in Figure 4.29.

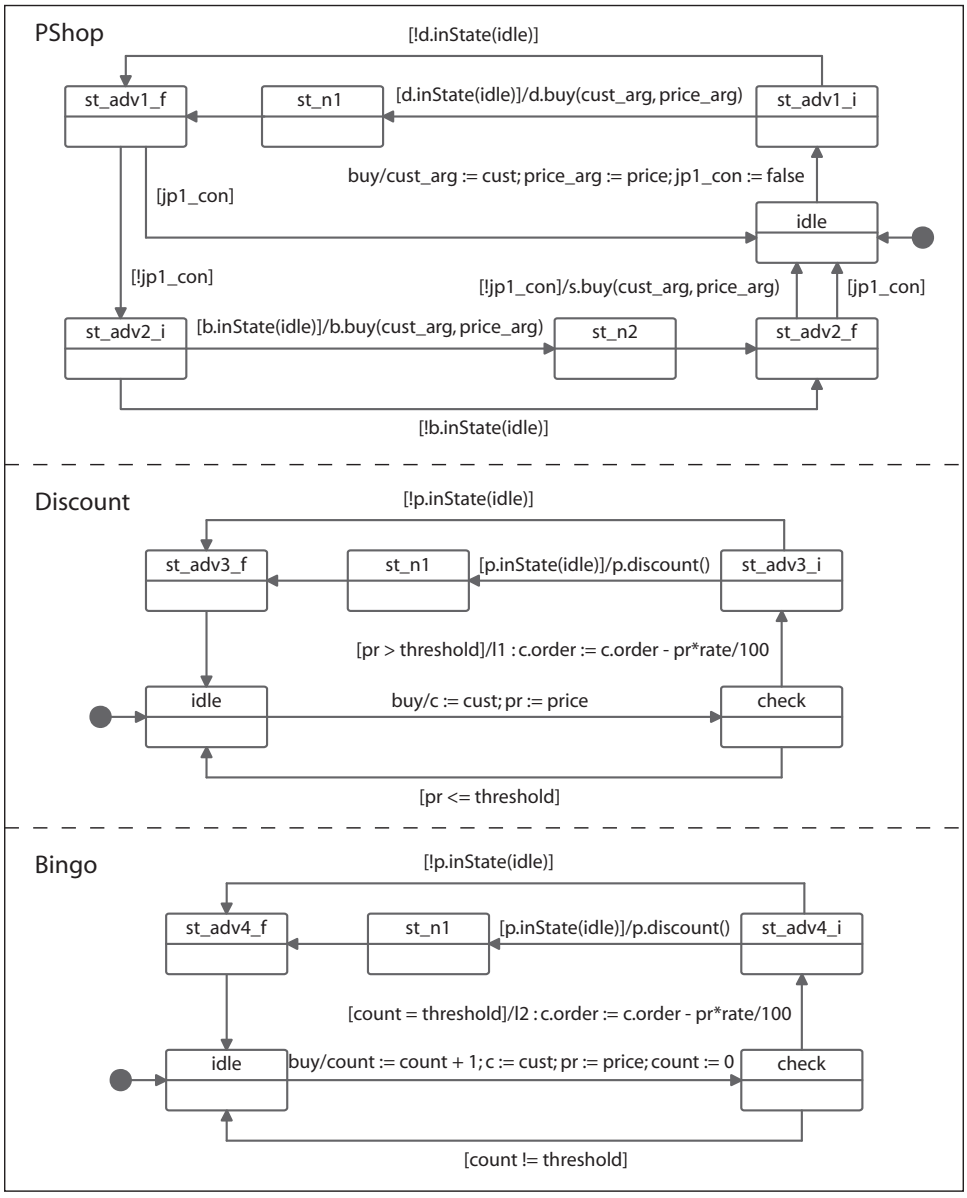

Figure 4.27: *ECOMM* woven OO model (using WP1) statecharts for `PShop` (proxy of `Shop`), `Discount`, and `Bingo`

`Shop`
$Signal = \emptyset,\ Call = \{\ buy(cust : Customer, price : int)\ \}$

Figure 4.28: *ECOMM* woven OO model (using WP1) events

$(\texttt{oc}, \texttt{Customer}, [\texttt{s} \mapsto \texttt{ops}, \texttt{order} \mapsto \texttt{0}]$
$(\texttt{ops}, \texttt{PShop}, [\ s \mapsto os, b \mapsto ob, d \mapsto od,$
$\quad cust_{arg} \mapsto null, price_{arg} \mapsto 0, jp1_{con} \mapsto false]),$
$(\texttt{od}, \texttt{Discount}, [\texttt{p} \mapsto \texttt{op}, \texttt{rate} \mapsto \texttt{10}, \texttt{threshold} \mapsto \texttt{40}, \texttt{c} \mapsto \texttt{null}, \texttt{pr} \mapsto \texttt{0}]),$
$(\texttt{ob}, \texttt{Bingo}, [\ p \mapsto op, count \mapsto 0, rate \mapsto 50, threshold \mapsto 100,$
$\quad c \mapsto null, pr \mapsto 0]),$
... (other initial instantiations in *ECOMM*'s OO model are unchanged)

Figure 4.29: *ECOMM* woven OO model (using WP1) initial instantiation

WP2

ECOMM's woven OO model (using WP2) is shown in the figures below:

- Class names and data are shown in Figure 4.30.
- The statechart of core class `Shop` is shown in Figure 4.31 (statecharts of other classes as the same as in WP1, and the event receptions of all classes are the same as in the unwoven AO model).
- The initial instantiation is shown in Figure 4.32.

WP2.1

ECOMM's woven OO model (using WP2.1) is shown in the figures below:

- Class names and data and the initial instantiation are the same as in WP2.
- The statechart of core class `Shop` is shown in Figure 4.33 (statecharts of other classes, as well as event receptions of all classes are the same as in WP2).

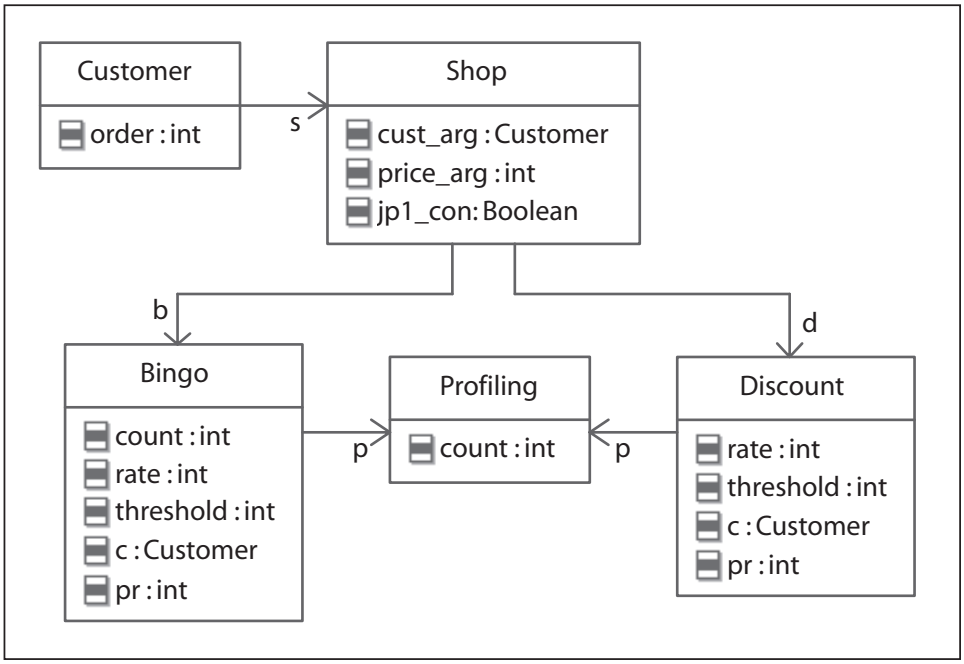

Figure 4.30: *ECOMM* woven OO model (using WP2) data

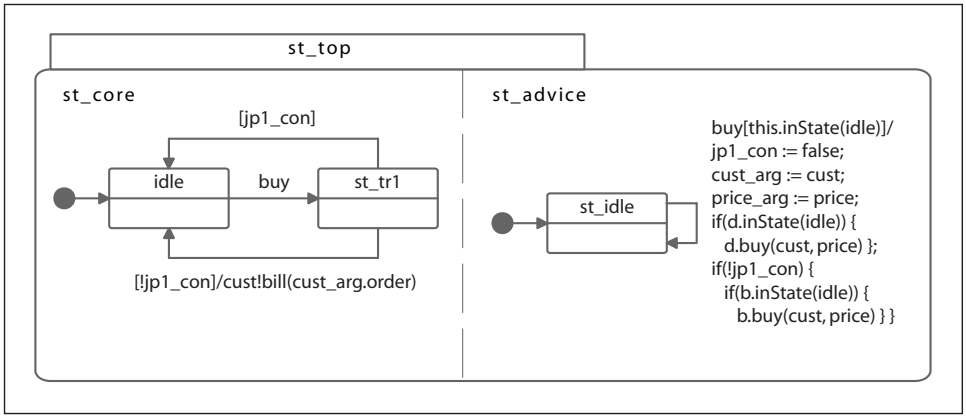

Figure 4.31: *ECOMM* woven OO model (using WP2) statechart for `Shop`

$(\texttt{oc}, \texttt{Customer}, [\texttt{s} \mapsto \texttt{os}, \texttt{order} \mapsto 0]$
$(\texttt{os}, \texttt{Shop}, [\ b \mapsto ob, d \mapsto od,$
$\quad cust \mapsto null, price_{arg} \mapsto 0, jp1_{con} \mapsto false]),$
$\ldots$ (the initial instantiation of other classes is the same as in WP1)$\}$

Figure 4.32: *ECOMM* woven OO model (using WP2) initial instantiation

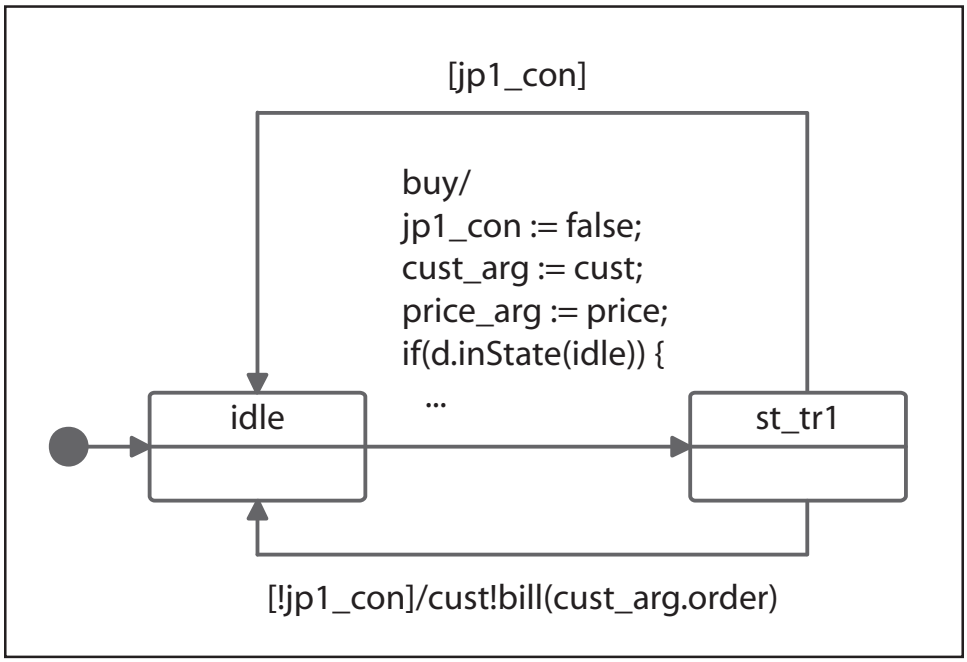

Figure 4.33: *ECOMM* woven OO model (using WP2.1) statechart for *Shop*

4.2.3 Reports

Analysis Report

The syntactic analysis of the *ECOMM* AO model reveals an overlap between advice (`Discount`, $adv1$) and (`Bingo`, $adv2$) of Figure 4.24, as they both apply to the same join point ($jp1$) and (`Discount`, $adv1$) > (`Bingo`, $adv2$) (note that while the overlap is explicit in the state-based WRL syntax of Figure 4.24, it is not so in the alternative syntax of Figure 4.25). This overlap indicates a *possible* interaction between concerns `Discount` and `Bingo`.

Verification Report

We verified PRL (whose observer class data and behaviour are shown in Figure 4.34) for the UML part of *ECOMM*'s unwoven AO model, and for its woven OO model (using WP1 and WP2.1) with all aspect combinations (for combinations involving `Profile`, the unbounded incrementation of the `count` attribute of `Profile` was excluded from its behaviour to reduce verification complexity - this behaviour change does not affect PRL). The results are tabulated in Table 4.2. For the unwoven model, the state-space size is 37 states and PRL is satisfied. Note that PRL only fails to satisfy with combinations $Discount+Bingo$ and $Discount+Bingo+Profiling$. This confirms that the advice overlap detected by the syntactic analysis (Section 4.2.3) task does indeed correspond to an interaction between concerns *Discount* and *Bingo*.

Table 4.2: *ECOMM* verification results using IFx

	WP1		WP2.1	
	States	PRL	*States*	PRL
No advice	≈ 160000	✓	≈ 57000	✓
Discount	550000+	✓	≈ 120000	✓
$Discount + Bingo$	550000+	×	≈ 120000	×
$Discount + Profiling$	550000+	✓	≈ 120000	✓
$Discount + Bingo + Profiling$	550000+	×	≈ 120000	×
Bingo	550000+	✓	≈ 120000	✓
$Bingo + Profiling$	550000+	✓	≈ 120000	✓

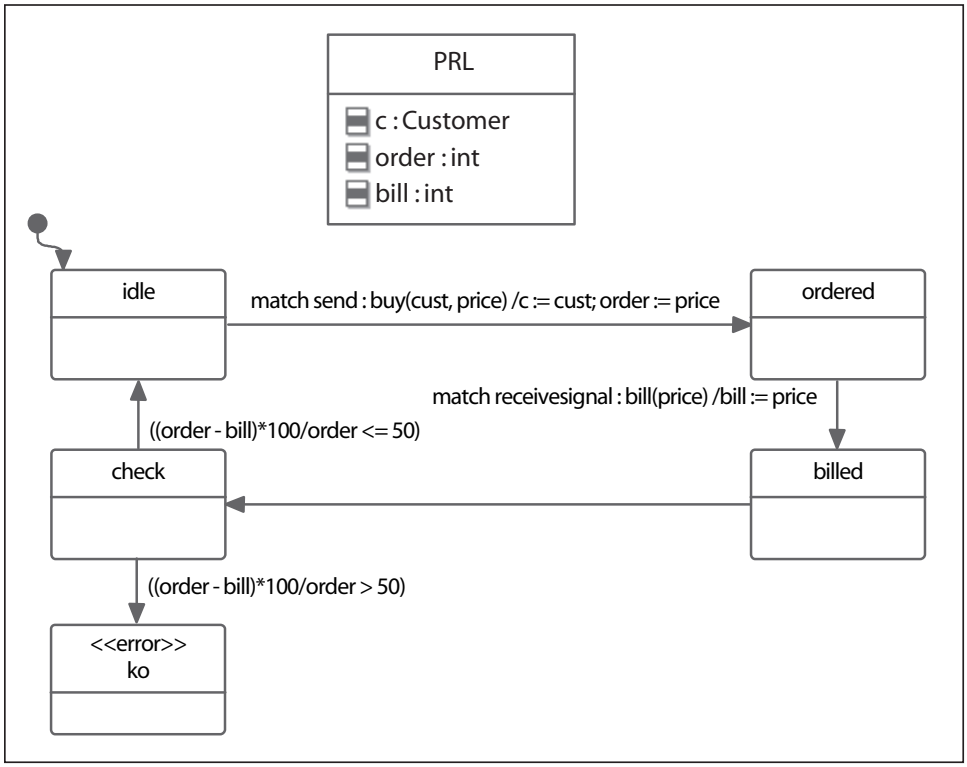

Figure 4.34: PRL observer class data and behaviour

Chapter 5

Discussion

This chapter evaluates the proposed process by informal analytical arguments concerning the verification complexity of the woven model generated by the weaving processes, the expressiveness of the AO modeling language, and the traceability of results from the static analysis and verification tasks to the AO model; and by empirical results obtained from applying the process to *FITEL* and *ECOMM*.

Verification complexity. The three weaving processes (WP1, WP2, and WP2.1) introduced in this paper differ both in the *number* of class elements they introduce to implement advice on *event* join points, and in *where* they allocate such elements in the woven model. The increased verification complexity of the woven model compared to the UML part of the unwoven model depends on both the *number* and *allocation* of advice elements (for event join points). The number of advice elements depends

on the AO model (as explained in previous sections), while for a given number of advice elements, the impact of the allocation of these elements on the verification complexity of the woven model decreases in order, from WP1 to WP2 to WP2.1. The decrease in verification complexity from one weaving process to another comes at the expense of support for AO model features: From WP1 to WP2 we lose support for *after* advice, and from WP2 to WP2.1 we additionally lose support for concurrent regions and conflicting transitions.

Expressiveness of AO model. We will assess the expressiveness of the AO modeling language described in Section 3.2 by comparing it to the (in our opinion, expressive) aspect definition language of EAOP. As described in Section 2.3, in EAOP, an aspect in its most basic form, is a rule that maps a join point in the *execution trace* of the core to advice. Aspects can be composed by recursion, choice, sequential, and (adapted) parallel composition operators. Compound aspects are state machines that evolve from one constituent aspect to another (based on the composition operators) in response to join points. Aspects themselves can contribute join points to the execution trace; that is, they can be advised by other aspects. In our modeling approach, a basic aspect is a state machine that reacts to join points by executing one or more advice trees (before or after the join point), where each advice tree prescribes one or more evolution steps based on the join point context (and optionally, the consumption of the join point in the case of before advice). Aspects can only

be composed sequentially, but at the granularity of advice; that is, order is imposed on advice (within a before/after category) and not on aspects. As explained in Section 3.2 aspects of aspects are also supported in our approach. We believe, there is no fundamental difficulty is changing the weaving processes to support the rich composition operators of EAOP; however, the affect of such support on the verification complexity of the woven model is an important consideration.

On another note, with static weaving processes such as ours, *per instance* advice (i.e. advice on a particular instance of a core class rather than on all instances of the core) cannot be supported. *FITEL* (Section 4.1) is an example of where per instance advice is needed: ideally, we would model the basic control software with a single class (say `Control`) and assign features to specific instances of this class. Per instance advice can be (perhaps not attractively) simulated statically by duplicating the core class for each advised core instance. This method has been applied to *FITEL* in Section 3.1 by duplicating `Control` per user ($\texttt{Control}1 - 3$).

Traceability Static analysis is performed on the unwoven model, and as such, its results are readily traceable to elements of the AO model. Formal verification however, is performed on the woven model. Assuming the UML verifier tool presents error scenarios in UML (rather than in a language that the tool translates UML to, e.g. Promela [41] or IF [33] - and both [41] and [33] do so), traceability of error scenarios to the unwoven AO model deteriorates from WP1 to WP2 to WP2.1, as class elements

that implement advice on *event* join points become less localized (advice elements for *action* join points have the same allocation for all approaches). Regardless of the weaving process used, the direct mapping from advice elements to advice trees allows reasonable traceability.

Empirical Results It appears that with current UML verification technology (based on a sample of one verification tool), WP1 is not feasible for moderately complex models (such as *FITEL* - Section 4.1), and should be used only if the model requires *after* advice on *event* join points and/or is relatively simple (such as *ECOMM* - Section 4.2). WP2.1 and we speculate WP2 (though without empirical evidence due to the limitation of the UML verification tool at our disposal) on the other hand, do appear feasible, and we believe a large set of useful AO models satisfy their restrictions.

Chapter 6

Conclusion

We have presented a process for detecting concern interactions in AO designs expressed in UML (for modeling concern data and behaviour) and WRL (a domain specific language for specifying how concerns crosscut). The process consists of two tasks: 1) A syntactic analysis of the unwoven AO model to alert the developer of potential sources of interaction. 2) Verifying properties of the model before and after the weaving of concerns to confirm/reject findings of task 1 and/or to reveal new interactions. At the heart of task 2 is a weaving process that maps an unwoven AO model to a behaviourally equivalent woven OO model. We present three weaving processes: WP1 (supports all features of WRL; yields a woven model of high verification complexity), WP2 (does not support after advice; verification complexity of woven model is generally lower than WP1), and WP2.1 (does not support after advice, con-

current regions, and conflicting transitions; verification complexity of woven model is generally lower than WP2), the choice of which is driven by required WRL features and the complexity of the AO model. For the (moderately complex) *FITEL* case study, we observed that using IFx [33] for UML verification, WP1 and WP2 are not feasible (due to verification complexity and lack of support for concurrent regions by the verification tool respectively), while WP2.1 is feasible. For the simple *ECOMM* case study, we observed that WP1 and WP2.1 are feasible, while WP2 is not (due to lack of support for concurrent regions by the verification tool).

6.1 Future Work

We propose the following directions for future research:

- Further optimizations to the weaving processes.
- Improving expressivity of our AO modeling language by
  - Implementing EAOP [10] composition operators and studying their affect on verification complexity.
  - Adding *introductions* (ala AspectJ [19]) to WRL.
- Experimenting with more case studies to empirically evaluate the effect of the weaving processes on verification complexity, and the expressivity of our AO modeling language.

- Investigating UML verification tools to determine the feasibility of verifying larger models.

Bibliography

[1] J. Aldrich, "Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming," in *FOAL: Foundations Of Aspect-Oriented Languages* (C. Clifton, R. Lämmel, and G. T. Leavens, eds.), pp. 7–18, March 2004.

[2] J. H. Andrews, "Process-Algebraic Foundations of Aspect-Oriented Programming," in *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, (London, UK), pp. 187–209, Springer-Verlag, 2001.

[3] L. Bergmans and M. Akşit, "Principles and Design Rationale of Composition Filters," [13], pp. 63–95.

[4] G. S. Blair, L. Blair, A. Rashid, A. Moreira, J. Araújo, and R. Chitchyan, "Engineering Aspect-Oriented Systems," [13], pp. 379–406.

[5] L. Blair, G. Blair, J. Pang, and C. Efstratiou, "'Feature' Interactions outside a Telecom Domain," in *Proceedings of Workshop on Feature Interactions in Composed Systems, ECOOP2001*, June 2001.

[6] L. Blair and M. Monga, "Reasoning on AspectJ Programmes," in *3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society* (B. Bachmendo, S. Hanenberg, S. Herrmann, and G. Kniesel, eds.), March 2003.

[7] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Comput. Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[8] C. Clifton and G. T. Leavens, "Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning," in *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)* (R. Cytron and G. T. Leavens, eds.), pp. 33–44, March 2002.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from Java source code," in *International Conference on Software Engineering*, pp. 439–448, 2000.

[10] R. Douence, P. Fradet, and M. Südholt, "Composition, reuse and interaction analysis of stateful aspects," in *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (K. Lieberherr, ed.), pp. 141–150, ACM Press, March 2004.

[11] R. Douence and M. Südholt, "A model and a tool for Event-based Aspect-Oriented Programming (EAOP)," Tech. Rep. 02/11/INFO, Ecole des Mines de Nantes, 2002.

[12] T. Elrad, M. Akşit, G. Kiczales, K. Lieberherr, and H. Ossher, "Discussing Aspects of AOP," *Comm. ACM*, vol. 44, no. 10, pp. 33–38, oct 2001.

[13] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[14] R. E. Filman and D. P. Friedman, "Aspect-Oriented Programming Is Quantification and Obliviousness," [13], pp. 21–35.

[15] *Foundations of Software Engineering (FOSE)*, ACM Press, October 2004.

[16] R. J. Hall, "Feature Interactions in Electronic Mail.," in *FIW* (M. Calder and E. H. Magill, eds.), pp. 67–82, IOS Press, 2000.

[17] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)," *Lecture notes in computer science*, 2004.

[18] D. Jackson, "Alloy: a lightweight object modelling notation," *Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," *Comm. ACM*, vol. 44, no. 10, pp. 59–65, October 2001.

[20] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, (New York), pp. 49–58, ACM Press, 2005.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proc. of European Conf. on Object Oriented Programming*, no. 1241 in Lecture Notes in Computer Science, Springer-Verlag, 1997.

[22] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff, "Second Feature Interaction Contest," in *Proc. 6th. Feature Interactions in Telecommunications and Software Systems* (M. H. Calder and E. H. Magill, eds.), (Amsterdam, Netherlands), pp. 293–324, IOS Press, May 2000.

[23] S. Krishnamurthi, K. Fisler, and M. Greenberg, "Verifying Aspect Advice Modularly," in FOSE04 [15].

[24] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[25] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary Design of JML: A Behavioral Interface Specification Language for Java," Tech. Rep. 98-06i, 2000.

[26] K. Lieberherr, D. Orleans, and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Comm. ACM*, vol. 44, no. 10, pp. 39–41, oct 2001.

[27] X. Liu, G. Huang, and H. Mei, "Feature Interaction Problems in Middleware Services.," in *FIW* (S. Reiff-Marganiec and M. Ryan, eds.), pp. 313–319, IOS Press, 2005.

[28] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud, "Using Aspects to Abstract and Modularize Statecharts," in *The 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004* (O. Aldawud, G. Booch, J. Gray, J. Kienzle, D. Stein, M. Kandé, F. Akkawi, and T. Elrad, eds.), October 2004.

[29] H. Masuhara and G. Kiczales, "Modular crosscutting in aspect-oriented mechanisms," in *ECOOP 2003—Object-Oriented Programming, 17th European Conference* (L. Cardelli, ed.), vol. 2743 of *lncs*, (Berlin), pp. 2–28, sv, jul 2003.

[30] M. Monga, F. Beltagui, and L. Blair, "Investigating feature interactions by exploiting aspect oriented programming," Tech. Rep. comp-002-2003, Lancaster University, Lancaster, LA1 4YR, 2003.

[31] M. Monga, "On Aspect-Oriented Approaches," in *European Interactive Workshop on Aspects in Software (EIWAS)* (K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, eds.), Sept. 2004.

[32] S. Nakajima and T. Tamai, "Weaving in Role-Based Aspect-Oriented Design Models," in *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004* (A. Moreira, A. Rashid, E. Baniassad, B. Tekinerdoğan, P. Clements, and J. Araújo, eds.), 2004.

[33] I. Ober, S. Graf, and I. Ober, "Validating timed UML models by simulation and verification," in *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS 2003), a satellite event of UML 2003, San Francisco, October 2003*, October 2003.

[34] I. P. Paltor and J. Lilius, "vUML: A Tool for Verifying UML Models," in *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99* (R. J. Hall and E. Tyugu, eds.), IEEE, 1999.

[35] J. Pang and L. Blair, "An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Features," in *Proc. 2nd Int'l Workshop on Aspect Oriented Programming for Distributed Computing Systems (ICDCS-2002), Vol. 2* (M. Akşit and Z. Choukair, eds.), July 2002.

[36] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications ACM*, pp. 1053–1058, Dec. 1972.

[37] T. Reenskaug, *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[38] M. Rinard, A. Salcianu, and S. Bugrara, "A Classification System and Analysis for Interactions in Aspect-Oriented Programs," in FOSE04 [15].

[39] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, second ed., 2005.

[40] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid, "Classifying and documenting aspect interactions," in *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, eds.), (Bonn, Germany), pp. 23–26, Published as University of Virginia Computer Science Technical Report CS–2006–01, March 2006.

[41] T. Schäfer, A. Knapp, and S. Merz, "Model checking UML state machines and collaborations.," *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 3, 2001.

[42] M. Sihman and S. Katz, "Superimpositions and Aspect-oriented Programming," *The Computer Journal*, vol. 46, no. 5, pp. 529–541, September 2003.

[43] e. a. Sullivan K., Griswold W. G., "On the criteria to be used in decomposing systems into aspects," in *ESEC/FSE '05: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 5–9, ACM Press, 2005.

[44] P. Tarr, H. Ossher, S. M. Sutton Jr., and W. Harrison, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," [13], pp. 37–61.

[45] N. Ubayashi and T. Tamai, "Aspect Oriented Programming with Model Checking," in *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)* (G. Kiczales, ed.), pp. 148–154, ACM Press, apr 2002.

[46] A. Wasowski, "Flattening Statecharts without Explosions," in *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 257–266, ACM press, 2004.

[47] M. Weiser, "Program Slicing.," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[48] J. Zhao and M. Rinard, "Pipa: A Behavioral Interface Specification Language for AspectJ," in *Proc. Fundamental Approaches to Software Engineering (FASE'2003), LNCS 2621*, pp. 150–165, Springer-Verlag, April 2003.