DOI:10.2298/CSIS111216033D

# Using Aspect-Oriented State Machines for Detecting and Resolving Feature Interactions

Tom Dinkelaker[1], Mohammed Erradi[2], and Meryeme Ayache[2]

[1] Ericsson R&D, Frankfurt, Germany
tom.dinkelaker@ericsson.com
[2] Networking & Distributed Systems Research Group, TIES, SIME Lab, ENSIAS, Mohammed V-Souissi University, Rabat, Morocco
erradi@ensias.ma, meryemeayache@gmail.com

**Abstract.** Composing different features in a software system may lead to conflicting situations. The presence of one feature may interfere with the correct functionality of another feature, resulting in an incorrect behavior of the system. In this work we present an approach to manage feature interactions. A formal model, using Finite State Machines (FSM) and Aspect-Oriented (AO) technology, is used to specify, detect and resolve features interactions. In fact aspects can resolve interactions by intercepting the events which causes troubleshoot. Also a Domain-Specific Language (DSL) was developed to handle Finite State Machines using a pattern matching technique.

Keywords: feature interactions, aspect interactions, aspect-oriented programming, state machines, conflict detection, conflict resolution, object-oriented programming, formal methods, domain-specific aspect languages.

## 1. Introduction

An important problem in modeling and programming languages is handling *Feature Interactions*. When composing different features in a software system, these may interact with each other. This can lead to a conflicting situation, where the presence of one feature may interfere with the correct functionality of another feature, resulting in an incorrect behavior of the system. Various techniques have been explored to overcome this problem. Among them, formal approaches have received much attention as a means for detecting feature interactions in communication service specifications.

In Software Product-Line (SPL) engineering [1], [2], the designer decomposes a software system into functional features by creating a feature model [1], [3]. But a feature model can only define a set of features and known interactions between them. Feature models do not help, when the designer overlooks a feature interaction – especially at the implementation level.

Aspect-Oriented Programming (AOP) [4] uses a special kind of modules called aspects that supports localization of code from crosscutting features. AOP has been extended with special language concepts for controlling aspect interactions [5], [6], but AOP does not support controlling feature interactions with modules that are not aspects in particular objects.

To address the above problems, in this work we propose a formal approach which uses an extension to finite state machines as the formalism for behavioral specification. The central idea behind using finite state machines as specification models is to have a strong mean to envision feature interactions. The formalism defines a process, which consists of the following steps: First, the developer gives a formal specification of each feature that extends the system's core feature, even partial specifications are allowed. Second, using a suitable composition mechanism for FSMs (e.g., the FSM's synchronized cross-product [7]), the developer makes a parallel composition of the selected feature specifications and analyzes this composition. Third, the developer can identify conflicting states by analyzing the composed specification of the global system. Forth, to resolve feature interactions, the approach uses aspect-oriented state machines to intercept, prevent, and manipulate events that cause conflicts. We suggest a new formalism for aspect oriented state machines (AO-FSM) where pointcut and advice are used to adopt Domain-Specific Language (DSL) [8] state machine artifacts. The advice defines a state and transition pattern that it applies at the selected points, i.e. it may insert new states and transitions as well as it may delete existing ones.

## 2. Case Study: Telecommunication Systems

### 2.1. Plain Old Telephone Service (POTS)

Features in Telecommunication systems are packages providing services to subscribers. The Plain Old Telephone System (POTS) is considered as a feature providing basic means to set up a conversation between subscribers. In the following we provide the design and the specification of the basic service of a telephone system (POTS). We assume that a phone is identified by a unique number, and it can be either calling or being called.

In this specification, there are three objects that constitute the telephone system: the "user", the "agent" and the "call" as shown in Fig. 1. According to our semantics, the instantiation of these objects provides three objects running in parallel. The communication between objects is based on operation calls using a rendezvous mechanism. Note that the behavior part of these objects is specified using a finite state machine model.

Fig. 1 partially specifies the behavior of the system using finite state machines (see section 4 for a more detailed presentation of the formalism). This system works as follows: Once the caller (user-1, an instance of the User behavior of Fig. 1) picks up (offhook) his phone (Agent-1, an instance of the Agent automaton), the network (designated by the object "call") responds by sending a tone. This user is then ready to dial the telephone number of the called party (using the operation "dial") using a standard telephone interface (Fig. 2). Then the network sends back a signal (operation "Ring") which causes a ring on the called phone (Agent-2, another instance of Agent). An Echo_ring is then sent to the caller (operation Echo_ring). We assume that the called user is always ready to answer a call. When the called user picks up (offhook) his phone, the ring is then interrupted and the two users engage in a conversation.

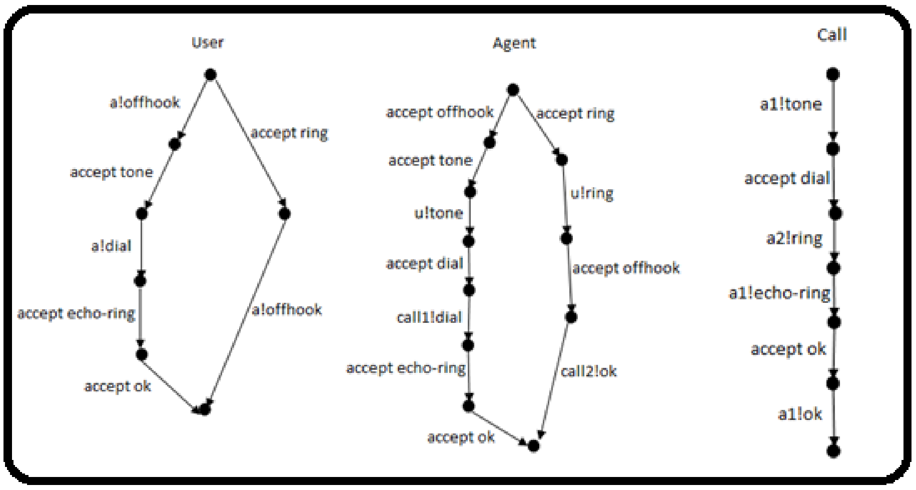

**Fig. 1.** Partial automata specifying the three objects

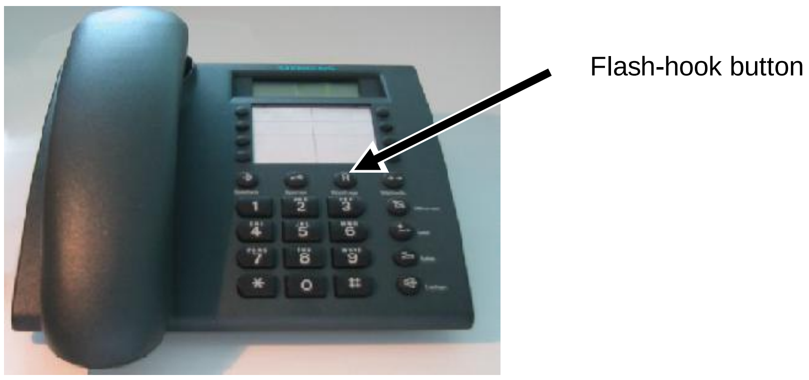

**Fig. 2.** Standard telephone interface with a flash-hook button (labeled with “R”)

## 2.2. Features available for User Selection (User Services)

According to the definition provided by Pamela Zave [9]: “in a software system, a feature is an increment of functionality, usually with a coherent purpose. If a system description is organized by features, then it probably takes the form B + F1 + F2 + F3 . . ., where B is a base description, each Fi is a feature module, and + denotes some feature-composition operation”. Therefore, telecommunication software systems have been designed in terms of features. So different customers can subscribe to the features they need. Many features can be enabled or disabled dynamically by their subscribers. Among the telecommunications features provided by a telephone system we found: Call Waiting, Three Way Calling, Call Forwarding, and Originating Call Screening.

### 2.2.1 Call Waiting (CW)

A Call Waiting feature (CW) is a service added to the basic service POTS described earlier. It allows a subscriber *A* (having the service CW) already engaged in a communication with a user *B* to be informed if another user *C* tries to reach him. *A* can either ignore the call of *C*, or press a flash_hook button to get connected to *C*. In other words, if *C* makes a call to *A*, while *A* is in communication with *B*, then *C* receives an Echo_ring, as if *A* was available, and *A* receives an “on hold” signal. Then *A* could switch between *B* and *C* by pressing the flash_hook button. If *B* or *C* hangs up, then *A* will be in communication with the user still on line. The basic service POTS to which is added the Call Waiting feature is symbolically designated by POTS + CW.

### 2.2.2 Three Way Calling (TWC)

The Three Way Calling is a service which extends the basic service POTS. It allows three users *A*, *B* and *C* to communicate in the following way: Consider a subscriber *A* (having the TWC feature) who is communicating with *B*. *A* can then add *C* in the conversation. To reach this goal, *A* put first *B* on hold by pressing a button flash hook button. Then, establish a communication with *C*. And finally, press the flash hook button again, to get, *A*, *B* and *C* connected. *A* can remove *C* from the conversation by pressing the flash hook button. If *A* hangs up, *B* and *C* remain in communication. The basic service POTS to which is added the Three Way Calling feature is symbolically designated by POTS + TWC.

#### 2.2.3 Call Forwarding on Busy (CFB)

Call forwarding on busy is a feature on some telephone networks that allows an incoming call to a called party, which would be otherwise unavailable, to be redirected to another telephone number where the desired called party is situated.

#### 2.2.4 Originating Call Screening (OCS)

The OCS Feature allows a user to define a list of subscribers hoping to screen outgoing calls made to any number in this screening list. A user *A* (with the OCS feature) who registered user *B* on the list will no longer make a call to *B*, but *B* could call *A*.

### 2.3. Feature Interactions

Feature interactions could be considered as all interactions that interfere with the desired operation of a feature and that may occur between a feature and its environment, including other features. Therefore, a feature interaction may refer to situations where a combination of different services behaves differently than expected.

For instance, pressing a “tap” button can mean different things depending on which feature is anticipated. This is the case of a flash-hook signal (generated by pressing such button) issued by a busy party could mean adding a third party to an established call (Three Way Calling) or to accept a connection attempt from a new caller while putting the current conversation on hold (Call Waiting). Should the flash hook be considered the response of Call Waiting, or an initiation signal for Three-Way Calling?

Another feature interaction may occur if we consider a situation where a user *A* has subscribed to the Originating Call Screening (OCS) feature and screens calls to user *C*. Suppose that a user *B* has activated the service Call Forwarding (CF) to user *C*. In this situation, if *A* calls *B*, the intention of OCS not to be connected to *C* will be violated since the call will be established to *C* by way of *B*.

Usually, the causes of interactions may be due to the violation of assumptions related to the feature functionality, to the lack of a technical support from the network, or to problems related to the distributed implementation of a feature. Despite the lack of a formal definition of a feature interaction due to the diversity of the interactions types, the reader will find a detailed taxonomy of the features interactions in [10].

Our approach to process the feature interaction problem consists in two methods based on formal techniques. The first method is used to detect the interactions while the second resolves them. In the context of formal techniques, interactions are considered as "conflicting statements". This may be a deadlock, a non-determinism, or constraints violation which may result from states incompatibility between two interacting features. The incompatibility between states can be detected using a "Model-Checking" technique.

## 3. Problem Statement

Feature interaction is considered a major obstacle to the introduction of new features and the provision of reliable services. In practical service development, the analysis of interactions has often been conducted in an ad hoc manner.

However, the feature interactions problem is not limited to the telecommunications domain. The phenomenon of undesirable interactions between components of a system can occur in any software system that is subject to changes. This is certainly the case for service-oriented architectures. First, we can observe that interaction is at the very basis of the web services concept. Web services need to interact, and useful web services will emerge from the interaction of more specialized services. Second, as the number of web services increases, their interactions will become more complex. Many of these interactions will be desirable, but others may be unexpected and undesirable, and we need to prevent their consequences from occurring.

There is a broad body of research that addresses the problem of feature interactions. However, as elaborated in the following, there are important limitations how the state of the art can detect and resolve feature interactions.

### 3.1. OOP cannot localize crosscutting Code of Features

Object-oriented programming (OOP) enables a hierarchical decomposition of the system into classes that can be extended by other classes. Using an OO language, developers can completely describe the system behavior in form of an implementation that can be executed. However, standard OO languages (such as Java or C++) do not provide special means to control feature interactions at the implementation level.

Furthermore, there are certain features for which OOP does not allow a good *Separation of Concerns* [12] because their implementation is scattered over several classes and tangled with the implementation of other features [4]. Examples for such features are non-functional components like tracing, billing calls, or feature interaction resolution.

Because OO languages do not have the right means to implement features and manage interactions among them, developers are left alone with implementing the logic that handles crosscutting and feature interactions, which results in code that is hard to understand and maintain.

## 3.2. Feature Models can only detect anticipated Feature Interactions

Feature models (FMs) [3] are a well-known technique to model the functionality of a system. They also allow prevent interactions between features that the developer is already aware of.

For example, Fig. 3 shows an FM for the telephone system, which does not only model features selected by the user, but choices made by the vendor. The telephone system has abstract features, such as a platform, a user interface, a receiving call indicator, and a set of user services, which can be implemented by a choice of features. The telephone platform can be either analog or digital (exclusive-or). If it is analog, then there can be a digital display. It must have a bell (mandatory feature), and in addition, it may have a LED (optional feature) that indicates receiving calls e.g. when the volume of the bell is low. The user may choose from the set of services from Section 2.2 (inclusive-or).

To model constraints of valid configurations that cannot be expressed using exclusive or inclusive or, the developer uses feature constraints. For example, the feature model in Fig. 3 defines that the features CW and TWC *requires* a flash_hook button has to be selected as well. In contrast, when selecting an analog platform, this *excludes* selecting a digital display.

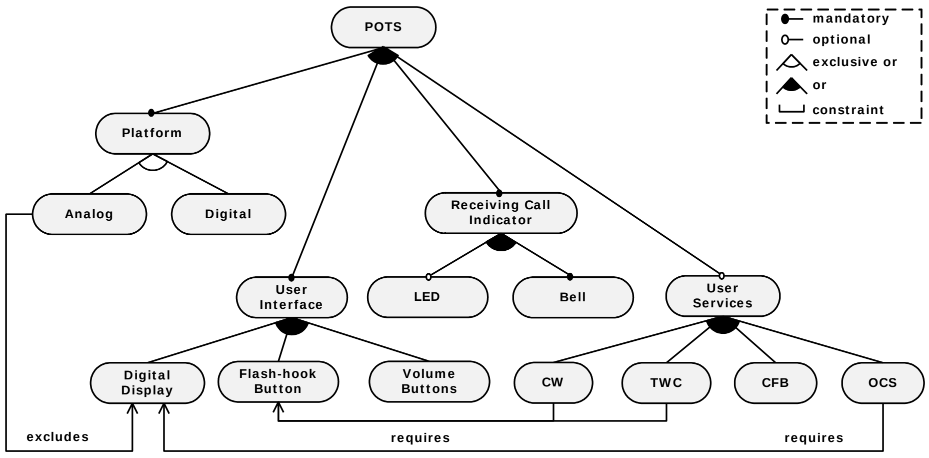

**Fig. 3.** A feature model of the telephone system (POTS)

An FM allows checking a particular selection of features, which is called a configuration, whereby a tool validates that all modeled feature constraints are met. However, an FM cannot guarantee that there is no feature interaction at the implementation-level. In case, the developer overlooks an interaction and does not model it correctly in the FM. In FMs, there is no support for formal behavioral modeling. Consequently, with FMs, developers cannot analyze the combined behavior of the features and for possible interactions between them.

### 3.3. AOP can only detect Aspect Interactions

Aspect-oriented programming (AOP) enables developers to modularize such non-functional concerns in OO languages. Important AOP concepts are *pointcut*, *join point model*, and *advice*. Pointcuts are predicates over program execution actions called join points. That is, a pointcut defines a set of join points related by some property; a pointcut is said to be triggered or to match at a join point, if the join point is in that set. It is also common to speak about join points intercepted by a pointcut. Such a join-point model (JPM) characterizes the kinds of execution actions and the information about them exposed to pointcuts (e.g. a method call). An Advice is a piece of code associated with a pointcut, it is executed whenever the pointcut is triggered, thus implementing crosscutting functionality. There are three types of advice, *before*, *after*, and *around*; relating the execution of advice to that of the action that triggered the pointcut the advice is associated with. The code of an around advice may trigger the execution of the intercepted action by calling the special method *proceed*.

However, there is a lack of a general approach to weave on code fragments of DSLs. The problem is that current AOP tools support only one JPM at a time, which is for most aspect-oriented (AO) languages one JPM for the events in the execution of an OO language [4]. Only for some DSLs, there is a domain-specific aspect language with a domain-specific JPM [13] (e.g. encompassing join points like a state transition in a state machine). Still, current AOP tools do not provide support for special quantifications for weaving aspects into programs written in several languages that have different kinds of join-point models.

For example, consider implementing a logging feature as an aspect that needs to be woven into the code of several languages for debugging, such as it need to be woven into code in Java with an Aspect-like JPM, code in SDL[1] that defines a JPM for FSMs, and code in LOTOS[2] that defines a JPM on top of protocols as communicating processes.

## 4. Characterization of Aspect-Oriented FSMs

In this paper, we propose a new formalism for aspect-oriented state machines (AO-FSM) which is based on finite-state machines and the Essential Behavioral Model. An AO-FSM defines a set of states and transitions like a FSM, but states and transitions do not need to be completely specified. Developers can selectively omit states, transitions, and labels, and therefore

[1] SDL: Specification and Definition Language: http://www.sdl-forum.org/SDL/index.htm

[2] LOTOS: Language Of Temporal Ordering Specification: http://language-of-temporal-ordering-specification.co.tv/

constitutes a partial FSM in which parts are missing so that it can be used as a pattern for matching against other FSMs and for manipulating them.

## 4.1. The Basic Finite State Machines (FSMs) Model

An automaton with a set of states, and its “control” moves from state to state in response to external “inputs” is called a Finite State Machine (FSM). A Finite State Machine provides the simplest model of a computing device. It has a central processor of finite capacity and it is based on the concept of state. It can also be given a formal mathematical definition. Finite State Machines are used for pattern matching in text editors, for compiler lexical analysis, for communication protocols specifications [15]. Another useful notion is the notion of the non-deterministic automaton. We can prove that deterministic finite State Machine, DFSM, recognize the same class of languages as Non-Deterministic Finite State Machine (NDFSM), i.e. they are equivalent formalisms.

***Definition 1:*** *A non-deterministic Finite State Machine is defined by a quadruplet $\langle Q, \Sigma, \delta, q_0\rangle$ where Q is a set of states, Σ is an alphabet, δ is the transition function, and $q_0$ is the initial state. The transition function is $\delta: Q\times \Sigma \rightarrow 2^Q$ where $2^Q$ is the set of subsets of Q.*

An event $\sigma \in \Sigma$ is accepted out from a state $q \in Q$ if the occurrence of σ is possible from the state q, i.e. if δ(q,σ) is not empty, we denote this by δ(q,σ)! When δ(q,σ) is empty, we write δ(q,σ)¬!. We consider a blocking state q (deadlock) if no transition is possible from this state. Formally: q is blocking $\Leftarrow\Rightarrow \forall\sigma \in \Sigma$, δ(q σ)¬!.

***Definition 2:*** *A deterministic finite state machine is defined by a quadruplet $\langle Q, \Sigma, \delta, q_0\rangle$ and corresponds to a particular case of the non-deterministic finite state machine where for any q and for any event σ, δ(q,σ) is either the empty set or a singleton. When δ(q,σ) is not empty, δ(q,σ) = {r} will be simply noted δ(q, σ) = r.*

The definitions introduced above refer to the basic formal model, but the actual notations used in our system modeling extend this model with other features in order to make it more practical and to support the requirements of our approach. Among these extensions we find: nested states, dependencies between states, and propositions. Therefore nested states, as shown in Fig.4, will be used to allow for partial automata modeling that hide parts of an automaton. Such partial specification hides the states and transitions which are not concerned by the composition and will not lead to an interaction. The dependencies between states allow indicating the order of occurrence of a given transition within different features. The propositions could be used as guards to characterize a given state according to the value of defined variables within such state.

Fig.1 gave an example of FSM. In addition, we give here additional examples concerning the partial finite state machines corresponding to the Call Waiting (CW) and Three Way Calling (TWC) features:

*A* partial formal specification of *POTS+CW is a FSM* $F_{CW}$ shown in Fig. 4. The shown states $Q_i$, for i=1 to 5, have the following semantics:

– $Q_1$: *A* and *B* are connected and start communicating.
– $Q_2$: *A* and *B* are communicating, then a call from *C* occurs on the switch of *A*.
– $Q_3$: *A* and *B* are communicating, and *A* receives the signal *call-waiting* indicating that someone is calling.
– $Q_4$: *B* is waiting, *A* and *C* are communicating.
– $Q_5$: *C* is waiting, *A* and *B* are communicating.

The events $E_i$, for i=1 to 3, have the following semantics:

– $E_1$: a call from *C* arrived on the switch of *A*.
– $E_2$: *A* receives the signal *call-waiting* indicating that someone else is calling.
– $E_3$: *A* pushes the *flash_hook* button.

A partial formal specification of POTS+TWC is the FSM $F_{TWC}$ shown in Fig. 4. The states $R_i$, for i=1 to 4, have the following semantics:

– $R_1$: *A* and *B* are communicating.
– $R_2$: *B* is waiting.
– $R_3$: *B* is waiting, *A* and *C* are communicating.
– $R_4$: *A*, *B* and *C* are communicating.

The event E3 has already been defined for the specification POTS + CW. The event $E_4$ has as its semantics:

– $E_4$: *A* is communicating with *C*.

Note that the states “in bold” $Q_1$ and $R_1$ represent nested FSM. For instance this means that the state $Q_1$ corresponds to a FSM which is a portion of the global specification, nested in this state $Q_1$.

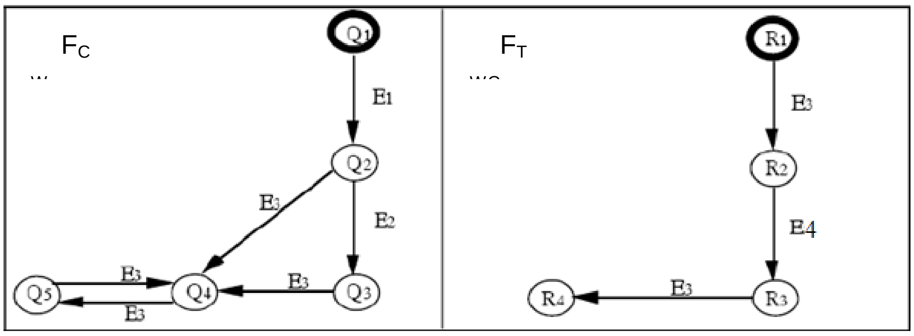

**Fig. 4.** Specification $F_{CW}$ (left) and specification $F_{TWC}$ (right)

## 4.2. Aspect-Oriented Finite State Machines (AO-FSM)

In an AO-FSM aspect, there are two parts: a *pointcut* and *advice* – like in other aspect-oriented languages for GPLs, but our pointcut and advice adapt DSL state machine artifacts. There is a composition model that defines how aspects at the meta-level are composed with base-level state machines, which is elaborated in the following.

An AO-FSM pointcut defines a state and transition pattern that selects all FSMs that the advice adapts. This pattern (at the meta-level) describes a model that needs to be observed on the execution of other (base-level) state machines. When the first part of the pattern is observed, in other words when a base-level state machine enters the initial state of the pointcut's state machine, our execution model creates a new meta-level instance of the state machine that describes the pattern and monitors the further execution of the base state machine. When observing new events at the base level, our composition model updates the meta-level instance in parallel to executing and updating the base-level instance. Finally, when the meta-level instance enters a *final state*, this means that the pattern has been recognized. In contrast, whenever the pointcut state machine does not define a matching transition for one of the observed events, the meta-level instance is deleted and garbage collected, since the pattern can no longer be fulfilled.

The advice defines a state and transition pattern that it applies at the selected points in the base-level state machine, i.e. it may insert new states and transitions as well as it may delete existing ones. As long as the meta-level instance remains in the final state, the advice is active, i.e. the changes are applied.

Fig. 5 shows visual models of all types of AO-FSMs. The upper row enumerates all pointcut types (alphabetic indices), in which only the shown parts define the pattern and omitted parts match like wildcards. The lower row enumerates all advice types (roman indices), in which only the bold parts adapt the corresponding parts of an FSM. When constructing an AO-FSM aspect, the different types of pointcut and advice types can be composed.

There are 6 different kinds of pointcuts: a) matches states with a particular label $S_p$, b) matches any state regardless of its label, c) matches a state in which a certain preposition $p_1$ is true, d) matches a state that has an incoming transition with label $E_o$, e) matches a state with an outgoing transition with a label $E_q$, and f) matches a sequence of two states with a transition that has the label $E_r$.

There are 8 different kinds of advice: i) inserts a new transition for event $E_s$, ii) inserts a new state $S_t$, iii) adds a new proposition $p_2$ to a state, iv) defines a dependency constraint $c_2$ between two states or two transitions, v) deletes the transition for event $E_u$, vi) deletes the state $S_v$, vii) deletes the property $p_3$, and finally, viii) defines a conflicting composition that results in an error message.

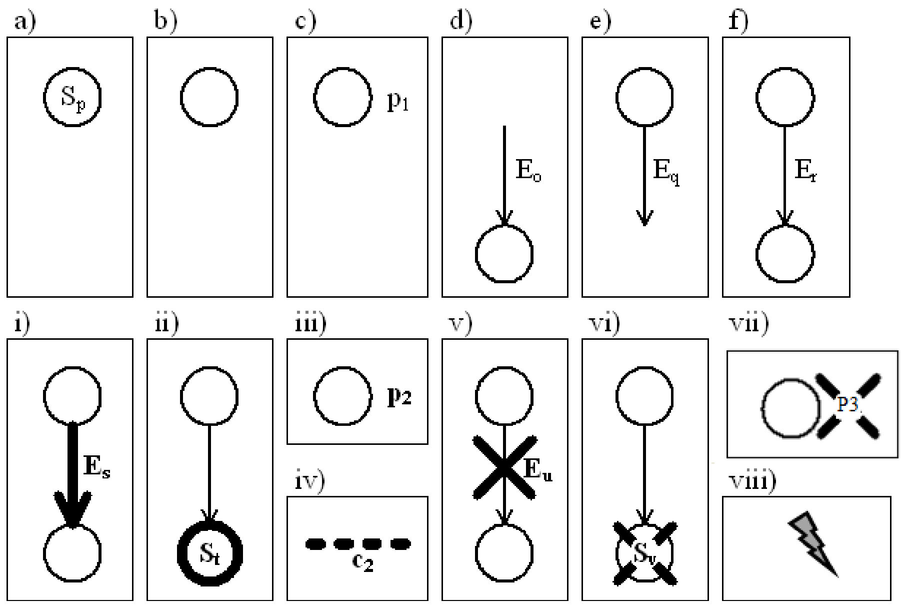

**Fig. 5.** Pointcut and advice types in aspect-oriented finite state machines

Usually, pointcut and advice compose the above basic patterns to more complex ones. For example, if we need a composite pointcut that matches a particular state Sp with an incoming transition with label Eo, then we would combine the basic patterns a) and f). For example, if we need a composite advice that adapts an existing state Sp by adding new transition Es to a new state St, then we would combine the basic patterns a), i) and ii). With these compositional semantics, rich adaption scenarios can be modeled.

To weave an aspect, we match all pointcuts and apply all advice for all FSMs. For a single FSM, the pointcut matches at every point in the FSM and applies the advice at each of these points. The adapted FSMs are then used for execution.

## 5. Resolving Feature Interactions with AO-FSMs

To control feature interactions, developers uses aspects to analyze and manipulate the behavior of a system that they compose from a set of modular feature specifications. In a nutshell, they compose specifications into a global behavior which we call an Essential Behavioral Model (EBM) of the system. The EBM consists of nested state machines that describe the composition of all features in the system. In the beginning, the EBM may expose feature interactions. To achieve a conflict-free composition of the features, developers use AO-FSM aspects to detect interactions that manifest singularities in the composed specification. There are three possible singularities: 1) the composed EBM is non-deterministic, 2) the composed EBM has contradicting prepositions, or 3) the composed EBM has blocking states. The main advantage of our approach is that feature interactions can be directly identified from the model. Finally, the developer can resolve feature interactions by eliminating singularities using AO-FSM aspect.

## 5.1. The Composition Mechanisms

A composition mechanism models the way in which features are composed together to yield a single FSM model of the system. In this section we present two possible composition mechanisms: the synchronized product and the exclusive sum. Such operations are defined as follows[3]:

***Definition 4:*** *Consider two FSMs* $A=\langle Q_A, \Sigma_A, \delta_A, q_{A0}\rangle$ *and* $B=\langle Q_B, \Sigma_B, \delta_B, q_{B0}\rangle$. *Let* $\Omega$ *be a subset of* $\Sigma_A$ *and* $\Sigma_B$, *in other words* $\Omega \subseteq \Sigma_A \cap \Sigma_B$. ***The Synchronized Product of A and B****, according to* $\Omega$, *is a FSM represented by* $A*B[\Omega] = \langle Q, \Sigma, \delta, q_0\rangle$ *defined formally as follows:*

– $Q \subseteq Q_A \times Q_B$ , $\Sigma = \Sigma_A \cup \Sigma_B$ , $q_0 = (q_{A0}, q_{B0})$
– $\forall q=\langle q_A, q_B\rangle \in Q$, $\forall \sigma \in \Omega$:
  $(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma_A)! \wedge \delta_B(q_B,\sigma_B)!)$
  $(\delta(q,\sigma)!) \Rightarrow (\delta(q,\sigma)) = (\delta_A(q_A,\sigma) \times \delta_B(q_B,\sigma))$
– $\forall q =\langle q_A, q_B\rangle \in Q$, $\forall \sigma \notin \Omega$:
  $(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma_A)! \vee \delta_B (q_B,\sigma_B)!)$
  $(\delta(q,\sigma)!) \Rightarrow (\delta(q,\sigma) = (\delta_A(q_A,\sigma)\times\{q_B\}) \cup (\{q_A\}\times\delta_B(q_B,\sigma))$

Intuitively, if *A* and *B* specifies two processes, then $A*B[\Omega]$ is the global specification of the two processes composed in parallel and have to synchronize on $\Omega$'s actions. By $A\otimes B[\Omega]$ we will note the product of the automaton *A* and *B* obtained by removing the blocking states from the *Synchronized Product* $A*B[\Omega]$. When $\Omega$ is empty, the two processes are said to be independent and their product is called the cross-product of A and B. It is denoted by $A*B[]$. When $\Omega = \Sigma_A \cap \Sigma_B$ , their product is denoted $A*B$.

***Definition 5:*** *(Sum of two FSMs, the Extension Relationship)*
*Consider two FSMs* $A=\langle Q_A,\Sigma_A,\delta_A,q_{A0}\rangle$ *and* $B=\langle Q_B,\Sigma_B,\delta_B,q_{B0}\rangle$. *The extension relation of A and B is a FSM defined formally as follows:*
– $Q \subseteq (Q_A\times Q_B) \cup Q_A \cup Q_B$, $\Sigma = \Sigma_A\cup\Sigma_B$ , $q_0 = (q_{A0},q_{B0})$

[3] Recall that a negated exclamation mark $(\delta_i (q_i, \sigma)\neg!)$ means that there is no transition defined, while an exclamation mark $(\delta_i (q_i, \sigma)!)$ means that there is a transition defined.

– $\forall q=\langle q_A,q_B\rangle \in Q \cap (Q_A \times Q_B),\ \forall \sigma \in \Sigma$:
$(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma)! \vee \delta_B(q_B,\sigma)!)$
$(\delta_A(q_A,\sigma)! \wedge \delta_B(q_B,\sigma)\neg!) \ \Rightarrow\ \delta(q,\sigma) = \delta_A(q_A,\sigma)$
$(\delta_A(q_A,\sigma)\neg! \wedge \delta_B(q_B,\sigma)!) \ \Rightarrow\ \delta(q,\sigma) = \delta_B(q_B,\sigma)$
$(\delta_A(q_A,\sigma)! \wedge \delta_B(q_B,\sigma)!) \ \Rightarrow (\delta(q,\sigma)) = (\delta_A(q_A,\sigma) \times \delta_B(q_B,\sigma))$

– $\forall q=q_A \in Q \cap Q_A,\ \forall \sigma \in \Sigma$:
$(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma)!)$
$(\delta(q,\sigma)!) \ \Rightarrow\ \delta(q,\sigma) = \delta_A(q_A,\sigma)$

– $\forall q=q_B \in Q \cap Q_B,\ \forall \sigma \in \Sigma$:
$(\delta(q,\sigma)!) \Longleftrightarrow (\delta_B(q_B,\sigma)!)$
$(\delta(q,\sigma)!) \ \Rightarrow\ \delta(q,\sigma) = \delta_B(q_B,\sigma)$

Intuitively, if *A* and *B* specify two processes, then *A*⊕*B* is the global specification of the two processes behaving exclusively.

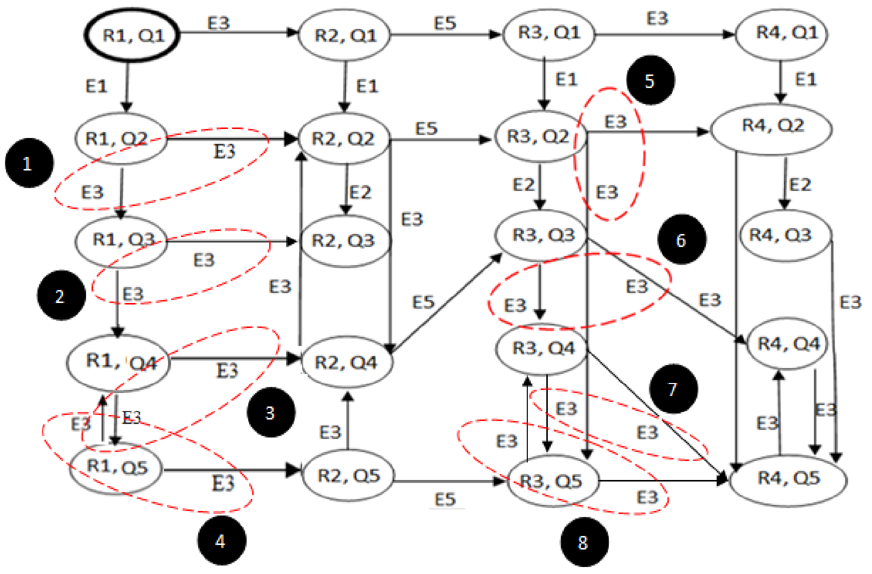

**Fig. 6.** Cross product of $F_{CW}$ and $F_{TWC}$

As an example of a composition, Fig. 6 shows the result of the cross product of the FSMs corresponding to the features Call Waiting and Three Way Calling (shown in Fig. 4). We can observe the presence of non-determinism at index i for i=1,2,3,4,5,6,7,8, which is illustrated by having at least two transitions for one event going out from the same state and leading to two different states. There are 8 cases of non-determinism in Fig. 6, which are indexed and high-lighted with the ellipses. For example, the state($R_3$,$Q_3$) has two transitions to ($R_3$,$Q_4$) and to ($R_4$,$Q_4$) but using the same event $E_3$. This non determinism reflects the presence of an interaction between the composed features (CW and TWC in this case).

## 5.2. The Essential Behavioral Model (EBM)

Recall that the principle of our method for managing feature interactions consists in three phases: the *global behavior specification*, the *interaction detection* and the *interaction resolution*. Interactions can be presented by states called *conflicting states*. This can be a *deadlock* (blocking) situation, a *non-determinism* or a *constraints violation* that is presented as an incompatibility between two states of features in interaction.

There are two steps that are necessary in order to design a global behavior specification for a system with a set of features:

– **Step 1:** Specify formally each feature (involved in the interaction) with the basic system service (i.e. POTS in the case of a telecommunication system). This specification can possibly be partial.

– **Step 2:** Make a composition of the features, using a suitable composition mechanism (e.g., synchronized product, a cross-product, a sum, …), leading to the global behavior defined by the EBM, which then is subject to further analysis. For instance, if the synchronized product is used, it implies making a synchronized automaton product (as shown in definition 4) of the behaviors of the composed features. Note that the synchronization alphabet could be possibly empty.

## 5.3. Interaction Detection

Interaction Detection consists in the identification of the conflicting states by analyzing the EBM automaton produced in Step 2 (see Section 5.2 above). Such states could be either a state where a given transition can lead to two distinct states (this is the case of non-determinism which is defined in definition 1), to a deadlock state (where one can execute no transition) or to a state constraints violation (i.e. a state belonging to the product of two features specifications, and that results from two incompatible states). Formally, this violation means that two incompatible states allocate different “logical” values to the same variable.

For example, there is a feature interaction when we compose the two feature specifications Call Waiting (CW) and Three Way Calling (TWC) using a cross-product operation. For instance, when *A* is in communication with *B* and *A* gets an incoming call from *C*, will the CW feature or the TWC feature be invoked?

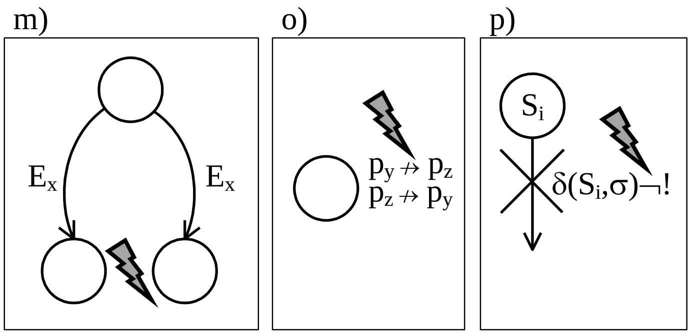

**Fig. 7.** Three detection aspects checking for composition singularities

When composing the aspects, a set of so-called *detection aspects* check the composition for possible conflicts. A detection aspect detects a singularity using a pointcut and its advice always declares a conflict, which makes the composition fail as long as the singularity is not corrected. Fig. 7 shows three detection aspects that detect the three aforementioned singularities: m) matches any state if there are more than one transition with the same event $E_x$, o) matches any state with contradicting prepositions $p_y$ and $p_z$, and p) matches every blocking state $S_i$ for which there is no outgoing transition. When necessary, developers can define their own detection aspects. Whenever one of the detection aspect' pointcuts matches in a composed system, its advice will report a conflict.

Detection aspects are in particular useful when composing many models and aspects that manipulate those models. Detecting composition singularities prevents any further incorrect processing of the system in a potentially undefined state. The above three detection aspects help automatically detecting the most important composition singularities. Therefore, the developer does no longer have to worry about them. Similar to related work on aspects interaction [5], [16], automatic feature interaction detection is enabled. However, automatic feature conflict resolution is not possible in general [5].

## 5.4. Interaction Resolution

In order to solve feature interactions, a resolution aspect can implement different resolution strategies. For instance, in our previous works we used the following ones:

– **Strategy 1**: Make a composition using an exclusive choice of the two features specifications involved in an interaction. The designer could use existing merge algorithms [15] for LTS (Labeled Transition Systems) based specifications. Such algorithm produces a specification where its behavior extends the merged ones. The definition of the "extension" relation was given in Definition 5.

- **Strategy 2**: Solve the interaction by making a precedence order upon the occurrence of certain events of the features in interaction. This allows a feature to hide some events from the other feature.

- **Strategy 3**: Establish a protocol between features involved in an interaction. This protocol consists in exchanging the necessary information to avoid the interaction. This approach is more adapted in the case where the features are dedicated to be implemented on distant sites.

However, these strategies are largely based on pre-established and rigid conventions. Therefore, in this paper we propose a more flexible and customizable approach: to use aspects also as an interaction resolution mechanism. According to this proposal, for resolving the conflict, the developer needs to specify a set of resolution aspects. Each aspect intercepts the reception of events, and removes one or more singularities (e.g. cases of non-determinism) from the composed specification. Depending on corresponding context (e.g. the path to the current state and the received events), the aspect can make a choice concerning which of the conflicting features should be active and which not. Therefore, a resolution aspect defines a pointcut and advice for the corresponding conflict resolution, which may have been detected using a detection aspect. Its pointcut matches the conflict situation. Further, its advice declares what states and transitions to remove from the composition such that it becomes deterministic. In the following we explain the suggested method in the case where an interaction occurs between the call waiting (CW) feature and the Three Way Calling (TWC) Feature specified in Section 2, when they are composed using the cross-product operation (see Fig. 6).

First, the detection aspect, in Fig. 7 at index m, identifies this non-determinism singularity. Second, the developer specifies the resolution aspect in Fig. 8. The figure illustrates the pointcut as the thin solid lines that are used as pattern to be recognized on some automaton. It illustrates two pieces of advice as the bold solid lines that indicate what will be added to the automaton. In this case, there are pointcuts that matches the paths E1;CW.E3 and E3;TWC.E3. In both cases, the advice inserts a precedence constraint over the non-deterministic transition labeled with the E3 event, depending from which FSM this transition originates from CW.E3 or TWC.E3. In other words, the resolution aspect resolves the interaction of the CW and TWC features by defining precedence between those features that depends on the sequence of previous events. Intuitively, if a call of C arrives on agent A (event E1) before A presses the flash_back button (event E3), the CW feature will be active. In this case, the left pointcut in Fig. 8 will match and temporarily remove the transition TWC.E3. Conversely, if E3 takes place before

E1, then the TWC feature will be active. In this case, the right pointcut in Fig. 8 will match and temporarily remove the transition CW.E3.

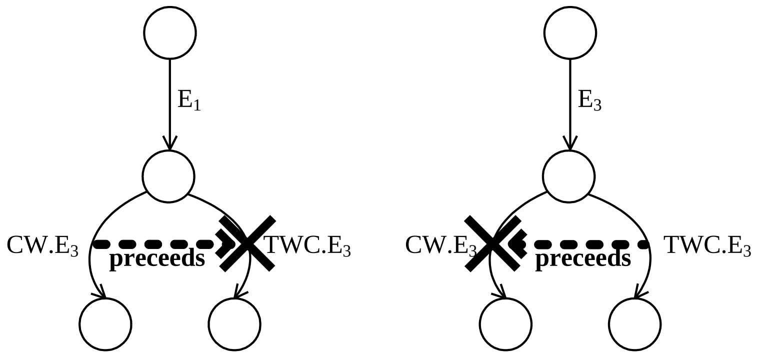

**Fig. 8.** A resolution aspect that resolves the CW/TWC interaction

In this way, these aspects can be applied to the cross-product in Fig. 6 in order to eliminate the non-determinism from index 1 and 2. In particular, in these indexes the left pointcut in Fig. 8 applies, while the corresponding a vice gives precedence to CW E3 transition with respect to TCW one. Thus, the interaction is resolved in favor of CW.

## 6. Implementation

This section describes the proof-of-concept implementation of the AO-FSM approach proposed in the previous sections. The proof-of-concept is provided as a domain-specific aspect language AO4FSM. On the one hand, we have implemented composition operators for state machines. On the other hand, we are using the implementation of AO4FSM to detect and to resolve feature interactions.

With this implementation, concrete solutions for feature interactions can be implemented by using aspects whose pointcuts detect conflict situations and advices to handle those situations.

We have implemented a prototype of AO4FSM in the Groovy language [18] using the POPART framework [17] that allows embedding DSLs and developing aspect-oriented extensions for those DSLs in form of plug-ins. Further, we have implemented the examples presented in [7] and which were used as a running example in this paper as a case study.

The implementation of AO4FSM is structured into four parts: the Embedded DSL, the Domain-Specific Join Point Model, the Domain-Specific Pointcut Language, and the Domain-Specific Advice Language. Each part will be elaborated in the following.

## 6.1. Embedded FSM DSL

The implementation of AO4FSM is based on an embedding of a small FSM DSL for describing finite state machines. The idea of embedding a language is to describe its syntax and semantics using an existing language – the host language – which is in our case Groovy. Basically, a language is implemented as a library, instead of implementing it with a parser and a compiler. Then the DSL programs are evaluated by invoking this library. Note that it is out of the scope of this paper to completely present the embedding of the FSM DSL. We therefore refer the reader who is interested in the general approach to [17].

The following excerpt in Lst. 1 gives a rough idea of how we embed FSMDSL into a Groovy class called `FSMDSL`. For the keyword in FSMDSL's syntax, the class defines methods, such as `fsm`, `state`, `transition`, and `when`.

```
public class FSMDSL {
  private State currentState;
  public StateMachine fsm(Map params, Closure body) {
    StateMachine stateMachine = new StateMachine(…);
    body.delegate = this; body.call();
    return stateMachine;}
  public State state(Map params, Closure stateDefinition){

  public void transitions(Closure transitionDefinitions) {…}
  public void when(Map params) {State from = currentState;
    def t = new Transition(from, params.to, params.event);
    from.addTransition(t);
…
```

**Lst. 1.** Groovy Code for embedding FSMDSL

Our host language is *Groovy.* Indeed we chose Groovy because it allows embedding DSLs (such as FSM DSL). Furthermore, Groovy is lightweight, dynamic, and provides a higher level of abstraction, but at the same time, you can mix Groovy code with Java. If one needs to extend a DSL with Aspects, like in our FSM DSL, one can do so by exploiting the dynamicity of Groovy provided by its Meta-Object Protocol (MOP) [17]. Despite the additional flexibility provided by using Groovy's MOP, Groovy only enhances Java instead of replacing it. Hence, our implementation runs on every standard JVM.

## 6.2. Domain-Specific Join Point Model

From a design point of view, Fig. 9 shows the join point types defined for AO4FSM. There are five join point types that represent points in the execution of a FSM program. The joinpoint *stargingStateMachine* is triggered when a state machine is started. The *EntringState* joinpoint could be reified once a FSM enters into a given state. And when it exits the given state, *ExitingState* joinpoint is then reified. The *ResetStateMachine* joinpoint is reified when moving back to the starting state. As far as concerned, the *EventReceiving* joinpoint, it is reified when the FSM receives an event.

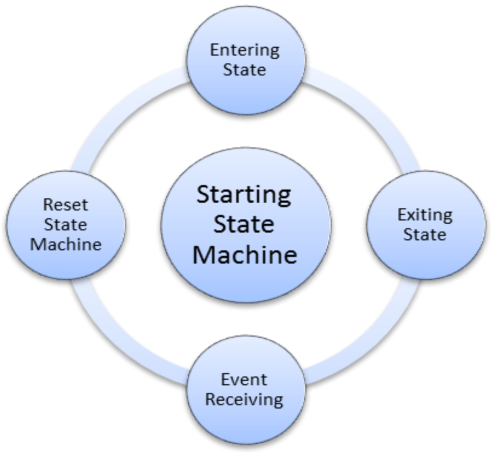

**Fig. 9.** The Join point types of AO-FSM

```
joinPointContext = new HashMap();
State thisTargetState =
(State)instrumentationContext.receiver;
joinPointContext.thisFSM =  thisTargetState.getOwningFsm();
joinPointContext.thisTargetState  = thisTargetState;
def joinPoint  = new EntringStateJoinPoint(joinPointContext);
joinPointContext.thisJoinPoint  = joinPoint;
```

**Lst. 2.** Groovy Code for reifying an EntringStateJoinPoint instance

As shown in the Lst. 2, each join point holds a set of parameters that defines its *context*. For instance, the context of the *EntringState* join point refers to the name of the FSM and to the targeted State as parameters. To reify a join point at runtime, the POPART framework will execute this code, which creates an instance of the class `EntringStateJoinPoint`.

### 6.3. Domain-Specific Pointcut Language

The other important components of our implementation design are the pointcuts. In POPART, each pointcuts is implemented as a class which inherits the `Pointcut` class. In FSMDSL, pointcut sub-classes match the current state parameters with the context of a corresponding join point. It returns "true" if the pointcut matches – and "false" if not. All pointcuts implement a similar pattern, as shown in Lst. 3 for the `EventReceivingPointcut`. This pointcut only matches `EventReceivingJoinPoint` with an event name given by `expectedEvent`.

In addition to join point types mentioned in the Fig. 9, we added a more history-based pointcut to AO4FSM: the *Stateful* pointcut. This pointcut uses a state machine as a pattern that should match the execution trace of an observed state machine. The pattern state machine is only internally visible for the pointcut, to keep track of the events received by an observed state machine.

```
public class EventReceivingPointcut extends Pointcut {
  String expectedEvent; //given by constructor
    public boolean match(JoinPoint jp) {
    if (jp instanceof EventReceivingJoinPoint) {
      EventReceivingJoinPoint esjp =
        (EventReceivingJoinPoint)jp;
      if (esjp.event.euqals(expectedEvent))
        return true;
      else
        return false;
    }
  }
}
```

**Lst. 3.** Excerpt of **`EventReceivingPointcut`** which matches reifications of **`EventReceivingJoinPoint`**

One can use the `StatefulPointcut` shown in Lst. 4 to detect the occurrence of patterns in FSMs, such as non-determinism. The first thing that the pointcut will do is that it makes sure that the *EventReceiving* joinpoints is triggered. Then it verifies that the current event matches the event and the name of the current State are the same once in the context of the join points (respectively: `esjp.getEvent()` and `esjp.getCurrentState()`). If the first state matches, we have to check the event that leads us to get out that the state does match as well and so on until we arrive to the final state in our pattern. Then we have to check if the current state in the pattern is final or not, if it is the case then the pointcut matches, and then its corresponding advice is applied.

```
public class StatefulPointcut extends Pointcut {
  StateMachine stateMachinePattern;
  public boolean match(JoinPoint jp) {
    State pCurrentState = stateMachinePattern.
                            getCurrentState();
    String pCurrentStateName = pCurrentState.getName();
    if (jp instanceof EventReceivingJoinPoint) {
      EventReceivingJoinPoint esjp = (…)jp;
      String jpCurrentStateName =
        esjp.getCurrentState().getName();
      if (((jpCurrentStateName == pCurrentStateName)) &&
          (pCurrentState.getEvents().
                        contains(esjp.getEvent()))) {
        stateMachinePattern.receiveEvent(esjp.getEvent());
      } else if((jpCurrentStateName == "*") &&
        pCurrentState.getEvents().contains(esjp.getEvent())){
          stateMachinePattern.receiveEvent(esjp.getEvent());
      }
      if(jpCurrentStateName.equals(
            stateMachinePattern.getFinalState().getName())) {
         return true;
      }
    }
    else
      return false;
  }
}
```

**Lst. 4.** Excerpt of **`StatefulPointcut`** to track an execution history

### 6.4. Domain-Specific Advice Language

The last part of the implementation is the advice language. This language deals with making changes to FSMs to which pointcuts have been matched. When implementing an advice using this language, developers can avoid the problem of non-determinism as mentioned earlier in this paper. For example, to resolve a non-determinism, the advice can remove one of the non-deterministic outgoing edges as suggested by the resolution aspect in Fig. 8.

In the implementation of the advice language, we heavily exploit the Groovy language features. We use closures and aspects that advise the FSMDSL language implementation in order to embed the semantics of the advice languages. One can think of the advice language of AO4FSM as a DSL that produces aspects for Groovy that change the implementation of the embedded FSMDSL. As the FSMDSL is changed by aspects, the evaluation of FSMDSL programs is adapted as well.

To implement the keywords in the advice language, we use *aspect templates*, which are closures whose evaluation returns a Groovy aspect that changes the behavior of some methods in FSMDSL. Such changes will remain dynamic, they do not change the static structure of the state machine, because there will be no changes conducted in the real FSM instance, but they will be only applied by the aspects. Hence, we can consider those changes as if they were applied to a "copy" of the real FSM. This part of the implementation refers to the patterns described in Fig. 8. Indeed each advice tries to change the behavior of an existing method in the `State` class.

For instance, the `removeTransitionAdvice` is the keyword that represents the advice type at index v) from Fig. 5, whose implementation is shown in Lst. 5. The `removeTransitionAdvice` changes the behavior of a method called `State.handleEvent`, which is responsible for looking up the transition from the `current` to the `next` state. The advice eventually changes the behavior of the `handleEvent` method. If for the `current` state an event is received, that matches the `event` name passed in the advice template's arguments, and if it finds a corresponding transition in the `current` state, then it will do nothing but it will proceed as if the transition does not exist. Otherwise, the advice calls `proceed`, which will execute the `handleEvent` as normal, i.e. without the change.

```
removeTransitionAdvice = { current, next, event ->
  aspect(name:"generatedName_" +
         "removeTransitionAdvice\$instante\$" +
         current+"\$"+next+"\$"+event, perInstance:current) {
    around(method_execution("handleEvent")){
      if(matchesRemovedTransition(current,next,event)){
       //omitting proceed ignores the transition
      else {
       return proceed() //proceeds the transition }
…
```

**Lst. 5.** Implementation of the `AddTransitionAdvice`

## 7. Discussion

To validate the approach, we use the AO4FSM prototype to automatically detect the interactions, and we have developed a resolution aspect to revolve these interactions. We could achieve the objectives stated in our introduction, namely the support of the separation of concerns (in particular crosscutting features), the formalization of the behavior, and how to deal with interactions. With the current prototype, conflicts can be successfully detected and resolved. However, correct results depend on whether the developer completely specifies the model and correctly implements aspects with the AO-FSM tool.

In the remainder of this section, we discuss the details about generality and limitations of our approach with respect to our model (Section 7.1) and the current prototype implementation (7.2).

### 7.1. Discussion about the Model

When using aspects for feature interaction detection and resolution, the developer has to decide case by case what are all encountered interaction singularities in a composed global behavior, which may have combinations of the basic singularities shown in Fig. 7. For example, there can be a non-deterministic transition with the same label (instance of index m) to a state which has two contradicting prepositions (instance of index o) within a composition. Another example is the case where there are more than one blocking states (instance of index p). The compositionality can lead to complex scenarios of interactions. Still, the detections aspects will seek all instances of the three kinds of interactions and combinations thereof.

Regarding the totality of feature interaction detection, also detecting other kinds of singularities is conceivable. Possible singularities can be derived from properties observed by the theory of finite state machine and graph theory. There are properties like connectivity and cycles. Some of these properties could indicate a singularity of a possible interaction. Other properties are not relevant for interactions in our model. For example, with respect to the connectivity property from graph theory, all FSMs in an EBM are by definition a *connected graph*, i.e., there is a path from any node to any other node in the graph. It is trivial to conclude that all synchronized cross-product are *connected*. Therefore, this property does not need to be observed in our model.

An example of a property which could be relevant is the presence of cycles. A cycle in a directed graph is a path from one node back to the same node, which in our EBMs could indicate whether the execution of a feature's behavior will not terminate. For instance, if there is no cycle, the behavior will *terminate* (i.e. there are only finite sequences of input that are recognized by the FSMs), or it stays alive (*liveness* property, i.e., something good can happen). It can be interesting in certain domains, such as *security* or *safety*, whether an EBM has such a property or not. If there are two EBMs that are free of cycles, still there may be composition that has cycles, i.e., the composed behavior may not terminate or a liveness property is violated, which indicates a feature interaction.

The current three detection aspects do not consider these more complicated scenarios. They are out of scope of the paper because for the time being we found no interesting situation in the software product-line scenarios we focus on. Still, at the current stage, we cannot draw universally valid conclusions from the case study. A larger case would be more convincing. At the end, only a formalization proof of the formalism in a proof assistant (like Isabelle or Coq) would give absolute guarantees.

With respect generality, the detection capabilities of our three detection aspects are limited. But, if it is needed, researchers and developers can define new kind of detection aspects. Since our pointcut language is based on finite state machines and from automata theory, we can derive that interaction detection for all properties of the composed automaton in an EBM (the global behavior) that can be recognized by another finite state machine (the pointcut). The class of properties that can be recognized by finite state machines is well known: it can recognize properties expressed by regular expressions over the input alphabet, Linear-Time Temporal Logic (LTL), and similar classes. With respect to generality, it can be considered as an advantage that the approach is based on formal methods of Automata Theory, which allows us to draw such conclusions from a large body of theoretic research.

The resolution capabilities in our model are complete with respect to finite state machine, because developers can add and remove all elements in an FSM, which are accessible as first-class objects in our aspect language. Developer can insert and delete both states and transitions. Developer can manipulate transitions by changing the incoming and outgoing state. We did not impose any restrictions with respect to the model into the aspect language about what could be manipulated in AO4FSM advice.

### 7.2. Discussion about the Implementation

Our prototype implementation only covers feature detection and resolution at design time. For save feature implementation, our approach could easily be integrated with a code generator from state machines to C or Java code.

Various practicable limitations need to be addressed by future work; the expressiveness of the model is confined by state machines and therefore systems whose behavior can be formalized as a regular language. The approach could be extended for models with richer semantics, which consequently would make it more complicated. Because we build the synchronized product of FSMs, the approach suffers from the well-known state explosion problem when using FSMs for modeling. Therefore, the prototype can only be used to analyze small models. In future work, we want to reduce synchronized products by finding equivalent states. Another limitation is that it currently does not nicely integrate with standard modeling notations, such as UML. In future work, we would like to support for importing UML state charts and let the developer enhance them to EBMs.

## 8. Related Work

Our work is related to the works in the field of FOP, AO modeling, and model driven development.

FOP [11] provides language support for implementing modular features that encapsulate basic functionality. Similar to FOP, our EBM and AO-FSM allow modular specification of features. While FOP uses so called *lifters* for inheriting features into a composition, we build on the sum for inheriting FSMs and the synchronized product for composing them. While FOP is an approach at the implementation level, we focus on the specification of features. FOP allows defining known interactions. In contrast, EBM and AO-

FSM allow automatic detecting of interactions that the developer is not aware of.

Aspect-oriented modeling has come up with various modeling notations into which aspects are woven. There are AO state machines [13] and other AO models available. However, they have been little explored in the context of detecting feature interactions in behavioral models. AO models can only detect conflicts involving aspects, but they cannot detect interactions between base features as we do.

An ongoing trend in language research is to extend more and more base languages with aspects, like we did for our state machine base language (FsmDSL). Recently, aspects for Petri nets have been proposed [20].

There are also a number of other extensible compilers and language workbenches that can be used for extending existing base languages with aspects, namely the Aspect SandBox [25], Reflex [22], JAsCo [21], the AspectBench Compiler [23], JAMI [24], and AspectASF [26]. These extensible language infrastructures mostly support only extensions to general-purpose language, but not to DSLs.

Achieving better modularization of language implementations in language engineering is a central subject of research in recent years. There are parser generators, such as ANTLR [28] and compiler-compilers, such as SableCC [29] that enable modular and extensible language implementations. In their specification languages, they use language constructs, such as inheritance, that enable better modularity in the language's specifications and their implementations. Aspect-oriented modularization itself has been proposed to be used in language engineering to improve modularity in language specifications [26], [30] and implementations [23]. Such special specification language constructs can be used to implement aspect-oriented language extensions in a modular way [27].

All above mentioned language implementation approaches use an external tool e.g. compiler that generates from the specification language (meta-language) the executable code in a particular target language. In contract to these approaches, we embed the DSL and aspects as internal DSLs. There is no external tool because DSL programs are processed with the same compiler as host programs. This fact allows us to use the extension constructs of the host language (inheritance and meta-object protocol) to extend the state machine DSL with aspect-oriented syntax and semantics using a modular AO language implementation.

Model-driven development proposes various kinds of models – not only FSMs. Life-Sequence Charts [19] are similar to AO-FSM. Such models are often used for code generation. While standard model notations do not adequately consider interactions, there are a few special models that allow expressing such constraints for a restricted set of domains, such as telecommunications for which special DSLs are available. Currently, developers are left alone to encode constraints on the modeled feature using constraint languages for which often there is no complete support for code generation. In contrast to this, possible domains for EBM and AO-FSM are not limited.

## 9. Conclusion

In this paper, we suggested a formal approach to detect and resolve feature interactions within a distributed software system. The approach is based on a new formalism for aspect-oriented state machines (AO-FSM) and language implementation AO4FSM based on finite-state machines and Essential Behavioral Models (EBM). An EBM defines states and transitions as a FSM, but states and transitions do not need to be completely specified and therefore it allows defining partial FSM models.

A specific mechanism for interactions detection and a strategy for feature interaction resolution were presented. The implementation of this mechanism and its associated strategy were made using the AO-FSM formalism. Therefore, the pointcut defines a state and transition pattern that selects all FSMs that the advice adapts, while the advice defines a state and transition pattern that it applies at the selected points. In fact, the approach uses aspect-oriented state machines to intercept, prevent, and manipulate events that cause conflicts.

**Acknowledgements.** This work was partially supported by the EMERGENT project (01IC10S01N), Federal Ministry of Education and Research (BMBF), Germany. A support was also provided by the DAAD (German Academic Exchange Service) and the Moroccan CSPT Research Program.

The authors would like to thank Professor Mira Mezini and the Software Technology Groups at the Technische Universität Darmstadt. Two of the co-authors have worked as part of her group while conducting this research.

The authors would like to thank the anonymous reviewers for their valuable comments and recommendations.

The authors would also like to thank Yassine Essadraoui who has contributed to the implementation of the prototype of AO4FSM and the telephone case study as part of his Master’s thesis.

## References

1. Clements, P. and Northrop, L., “Software product lines”, Addison-Wesley, 2001.
2. Pohl, K. and Böckle, G. and Van Der Linden, F., “Software product line engineering: foundations, principles, and techniques”, Springer-Verlag New York Inc, 2005.
3. K. Czarnecki and A. Wasowski. “Feature diagrams and logics: There and back again” in Proc. 11th Int. Software Product Line Conference (SPLC 2007), Washington, DC, USA, 2007, pp. 23–34.
4. Kiczales, G. and Lamping, J. and Mendhekar, A. and Maeda, C. and Lopes, C. and Loingtier, J.M. and Irwin, J.: “Aspect-oriented programming” in Proc. Europ. Conf. on Object-Oriented Programming, Springer, 1997, pp. 220–242.
5. G. Kniesel, “Detection and Resolution of Weaving Interactions. TAOSD: Dependencies and Interactions with Aspects”, In Transactions on Aspect-Oriented

Software Development V, pp. 135–186, LNCS, vol. 5490, Springer Berlin / Heidelberg, 2009.

6. Tanter, E., “Aspects of composition in the Reflex AOP kernel”, Software Composition, Springer, 2006, pp. 98–113.
7. M. Erradi and A. Khoumsi, “Une approche pour le traitement des interactions de fonctionalités des systèmes téléphoniques”, in Proc. Colloque Francophone International sur l'Ingénierie des Protocoles (CFIP'95), Rennes, France, 1995.
8. M. Mernik, J. Heering, and A.M. Sloane, “When and how to develop Domain-Specific Languages” ACM Computing Surveys (CSUR), vol. 37, no. 4, 2005, pp. 316–344.
9. Pamela Zave, “Feature Interaction”, http://www2.research.att.com/~pamela/fi.html
10. E.J. Cameron, N.D. Griffeth, Y.-J. Lin, M. Nilson, W.K. Schnure, et H. Vlethuijsen. “A feature Interaction Benchmark for IN and beyond”, Feature Interactions in Telecommunications Systems, Eds. L.G. Bouma and H. Velthuijsen, IOS Press, Amsterdam, 1994.
11. Prehofer, C.: “Feature-oriented programming: A fresh look at objects” in Proc. ECOOP, Springer, 1997, pp.419–443.
12. Parnas, D.L., “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.
13. M. Mahoney, T. Elrad, “A Pattern Story for Aspect-Oriented State Machines”, LNCS, Vol. 5770, 2009.
14. G. v. Bochmann, “Finite State Description of Communication Protocols”, Computer Networks, Vol. 2 (1978), pp. 361-372.
15. F. Khendek and G. v. Bochmann, “Merging Behavior specifications”, Proc. FORTE'1993, Boston, USA.
16. W. Havinga, I. Nagy, L. Bergmans, M. Aksit, "A graph-based approach to modeling and detecting composition conflicts related to introductions". In Proc. International Conference on Aspect-Oriented Software Development, ACM, 2007.
17. T. Dinkelaker, M. Eichberg, and M. Mezini, „An Architecture for Composing Embedded Domain-Specific Languages”. In Proc. Aspect-Oriented Software Development ACM New York, 2010.
18. D. König, A. Glover, “Groovy in Action”. Manning, 2007.
19. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design, vol. 19, no. 1, pp. 45–80, 2001.
20. T. Molderez, B. Meyers, D. Janssens and H. Vangheluwe, “Towards an Aspect-oriented Language Module: Aspects for Petri Nets”, In Proc. Workshop on Domain-specific Aspect Languages, ACM New York, 2012.
21. D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: “An Aspect-Oriented Approach tailored for Component-based Software Development.” In AOSD, pages 21-29, 2003.
22. E. Tanter. From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming. PhD thesis, Université de Nantes, France, 2004.
23. P. Avgustinov, J. Tibble, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam, “abc: An extensible AspectJ Compiler.” In AOSD, pages 87-98, 2005.
24. W. Havinga, L. Bergmans, and M. Aksit, “Prototyping and Composing Aspect Languages using an Aspect Interpreter Framework.” In ECOOP, pages 180-206, 2008.

25. H. Masuhara, G. Kiczales, and C. Dutchyn, “A Compilation and Optimization Model for Aspect-Oriented Programs.” In CC 2003, volume 2622 of LNCS, pages 46-60, 2003.
26. P. Klint, T. van der Storm, and J. Vinju, “Term Rewriting Meets Aspect Oriented Programming.” In Proc. of Processes, Terms and Cycles: Steps on the Road to Infinity, Springer, LNCS 3838, 2005.
27. E. Van Wyk, “Aspects as modular language extensions.” In Electronic Notes in Theoretical Computer Science, volume 82(3), pages 555-574, Elsevier, 2003.
28. T. Parr, “The definitive ANTLR reference: building domain-specific languages.” The Pragmatic Bookshelf, 2007.
29. E.M. Gagnon and L.J. Hendren, “SableCC, an object-oriented compiler framework.” In Proc. Of Technology of Object-Oriented Languages, pages 140-154, IEEE, 1998.
30. M. Mernik, X. Wu, and B. Bryant, “Object-oriented language specifications: Current status and future trends.” In ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs (ERLS), 2004.

**Tom Dinkelaker** holds a PhD and a German Diploma in computer science from the Technische Universitaet Darmstadt, Germany. His research focuses on the implementation of embedded domain-specific languages and aspect-oriented programming languages. To provide support for customizing language semantics and implementation strategies, in his thesis, he explored the potentials of using meta-object protocols to enable open language semantics. Tom has embedded a set of languages that are language product-lines, i.e., their syntax and semantics can be extended by language developers or end users in order to customize them for special domains. Tom is now working at Ericsson R&D in the Customer Care team. At Ericsson, he develops the next generation of business support systems that delivers features on top of a flexible architecture that customers adapt for individual needs.

**Mohammed Erradi** has been a professor in Computer Science since 1986. He has been leading the distributed computing and networking research group since 1994 at ENSIAS (Ecole Nationale d’Informatique et d’Analyse des Systèmes) of Mohammed V-Souissi University (Rabat Morocco), and was head and founding member of the Alkhawarizmi Computing Research laboratory. Before joining ENSIAS, Professor Erradi has been affiliated with the University of Sherbrooke and the University of Quebec in Canada. His recent main research interests include Communication Software Engineering, Distributed Collaborative Applications, and Reflection and Meta-level Architectures. He obtained his Ph.D. in 1993 at University of Montreal in the area of Communicating Software Engineering under the supervision Professor Gregor Von Bochmann. He is currently the Principal Investigator of a number of research projects grants. Among the topics of these projects we find: Collaborative environment for Telediagnosis in NeuroScience, Cloud Computing Security, Security Policies composition, Adaptive Wireless Sensor Networks, Vertical Handover in Mobile Networks. Professor Erradi has published more than 60 papers in international conferences and journals. He has organized and chaired five international scientific events and has been a member of the program committee in multiple international conferences.

**Meryeme Ayache** is a young security researcher, she graduated in 2012 from ENSIAS (Ecole Nationale Supérieur d'Informatique et d'Analyse des Systèmes, Rabat, Morocco) specializing in Security of Information Systems. She participated in the implementation of a project on "Behavioral Modeling with Aspect-Oriented State Machines" as an internship within the Software Technology Group of the Technical University of Darmstadt.  Her interest is on mobile computing and security.

*Received: December 16, 2011; Accepted: July 18, 2012.*