Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 809–816

ISBN 978-83-60810-22-4

Using Aspect-Oriented State Machines for Resolving Feature Interactions*

Tom Dinkelaker
Software Technology Group
Technische Universität Darmstadt, Darmstadt, Germany
Email: dinkelaker@cs.tu-darmstadt.de

Mohammed Erradi
Networking & Distributed Systems Research Group, TIES, SIME Lab,
ENSIAS, Mohamed V-Souissi University, Rabat, Morocco
Email: erradi@ensias.ma

***Abstract*[1]—**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.

I. 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 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 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 pointcuts and advices 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.

II. Problem Domain: Telecommunication Systems

A. 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 Figure1. According to our semantics, the

* This work was partially supported by: the EMERGENT project (01IC10S01N), Federal Ministry of Education and Research (BMBF), Germany, and the DAAD (German Academic Exchange Service) program.

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.

This system works as follows (Fig. 1): Once the caller (user-1) picks up (offhook) his phone (Agent-1), 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. Then the network sends back a signal (operation "Ring") which causes a ring on the called phone (Agent-2). 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.

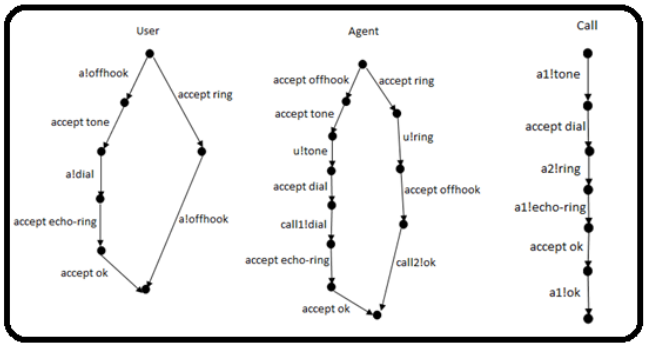

Fig. 1 Partial automata specifying the three objects

B. 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.

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.

A partial formal specification of *POTS+CW is an FSM* F_{CW} shown in Figure 2. The states Qi, 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 4, 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.

2) Three Way Calling (TWC[2])

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.

A partial formal specification of POTS+TWC is the FSM F_{TWC} shown in Figure 3. The states Ri, for i=1to 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 events E_i, for i=3 and 4, have already been defined for the specification POTS+CW.

The event E_5 has as its semantics :

- E_5 : A is communicating with C.

Note that the states "in bold" Q1 and R1 represent nested FSM. For instance this means that the state Q1 corresponds to an FSM which is a portion of the global specification, nested in this state Q1.

[2] The abbreviation TWC for Three Way Calling should not be confused with trust-worthy computing.

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.

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.

C. 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.

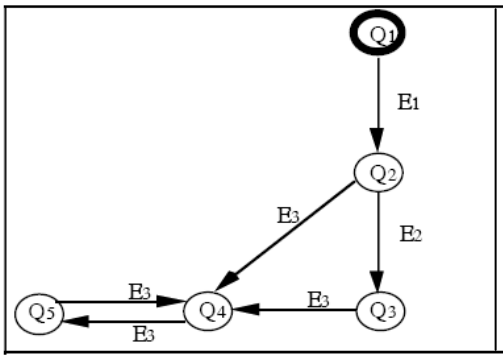

Fig. 2: Specification F_{CW} of POTS+CW

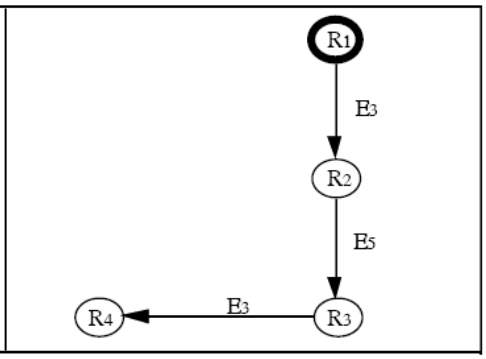

Fig. 3: Specification F_{TWC} de POTS+TWC

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 [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.

III. 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, however, 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.

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[3] that defines a JPM for FSMs, and code in LOTOS[4] that defines a JPM on top of protocols as communicating processes.

IV. Behavioral Modeling of Features

This paper proposes to model software using models that defines details of the behavior of a system and each of its features. As elaborated in the following, the proposed formalism is based on finite state machines (Section IV.A). It defines the basic system in a behavioral model (Section IV.B) and it defines the behavior of features using aspects (Section IV.V).

A. Finite State Machines (FSMs)

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 [16]. 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 $\delta(q,\sigma)$ is not empty, we denote this by $\delta(q,\sigma)!$

When $\delta(q,\sigma)$ is empty, we write $\delta(q,\sigma)\neg!$. We consider a blocking state q (deadlock) if no transition is possible from this state. Formally: q is blocking $\Leftrightarrow \forall\sigma \in \Sigma, \delta(q\ \sigma)\neg!$.

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, $\delta(q,\sigma)$ is either the empty set or a singleton. When $\delta(q,\sigma)$ is not empty, $\delta(q,\sigma) = \{r\}$ will be simply noted $\delta(q, \sigma) = r$.*

For all FSM A, the set of accepted traces will be designated by L_A.

Definition 3: *Consider 2 FSMs $A=\langle Q_A, \Sigma_A, \delta_A, q_{A0}\rangle$ and $B=\langle Q_B, \Sigma_B, \delta_B, q_{B0}\rangle$ respectively accepting regular languages L_A and L_B,* ***the sum of A and B*** *is designated $A\oplus B$ accepting the regular language $L_A \cup L_B$. Moreover, if A and B are deterministic then $A\oplus B$ is also deterministic. Intuitively if A and B specifies 2 processes, then $A\oplus B$ is the global specification of the two processes operating in an exclusive manner.*

Definition 4: *Consider 2 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 Ω be a subset of Σ_A and Σ_B, in other words $\Omega \subseteq \Sigma_A \cap \Sigma_B$.* ***The Synchronized Product of A and B****, according to Ω, is an FSM represented by $A*B[\Omega] = \langle Q, \Sigma, \delta, q_0\rangle$ defined formally as follows:*

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

When Ω is empty, two processes are said to be independent and their product is denoted $A*B[]$. When $\Omega = \Sigma_A \cap \Sigma_B$, their product is denoted $A*B$. Intuitively, if A and B specifies 2

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

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

processes, then A∗B[Ω] is the global specification of the 2 processes composed in parallel and have to synchronize on Ω's actions.

Note that A⊗B[Ω] is the product of the automaton A and B obtained by removing the blocking state from the *Synchronized Product A∗B[Ω]*.

Definition 5: *(Sum of two FSMs, the Extension relationship)*
Consider two FSMs $A=\langle Q_A, \Sigma_A, \delta_A, q_{A0}\rangle$ *and* $B=\langle Q_B, \Sigma_B, \delta_B, q_{B0}\rangle$ *which accept respectively the regular languages* L_A *and* L_B*. The sum of A and B noted A⊕B accepts the regular language* $L_A \cup L_B$*. In addition, if A and B are deterministic then A⊕B is deterministic.*

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

B. Essential Behavioral Model (EBM)

The principle of our method for managing feature interactions, consists in three phases: the global behavior specification, the interactions detection and the interactions 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.

B.1. Global Behavior Specification: this phase consists in two steps:

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 parallel composition of the features, leading to a global behavior called an Essential Behavioral Model (EBM), to be analyzed. This implies making a synchronized automaton product (as shown in definition 4) of the behaviors of the composed features. The synchronization alphabet could be possibly empty.

B.2. Interactions Detection:

Identify conflicting states by analyzing the EBM automaton produced in Step 2. 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.

The method of interaction resolution consists in three strategies. Recall that these strategies need to be applied during the specification phase. One among these strategies could be chosen and used depending on the type of interaction.

B.3. Interactions Resolution:

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 [17] 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.

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 II.

Using simultaneously both services (Call waiting and Three Way Calling) is formally represented by $\mathbf{F}_{CW||TWC}$ which is the product (Definition 4) of two FSMs F_{CW} and F_{TWC}. In other words, $F_{CW||TWC}=F_{CW}*F_{TWC}[]$ (Definition 4). Ω is empty since here we consider the case (event E_3) where pushing the *flash_hook button by A is considered by one among the provided features and not by both of them simultaneously.* The states of $F_{CW||TWC}$ will be designated by ⟨Qi, Rj⟩ where Qi and Rj are respectively the states of F_{CW} and F_{TWC}.

The interaction (ambiguity) is detected by the presence of a non-determinism on the states ⟨Qi, Rj⟩ of $F_{CW||TWC}$ where i=2,3,4,5 and j=1,3,4. Intuitively, when he pushed the *flash_hook* the subscriber A could not know if the signal is interpreted (executed) by the feature CW or by the TWC.

V. Aspect-Oriented Finite State Machines (AO-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 an FSM, but states and transitions do not need to be completely specified. Developers can selectively omit states, transitions, and labels, and therefore 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.

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. An AO-FSM pointcut defines a state and transition pattern that selects all FSMs that the advice adapts. 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.

Fig. 2 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 a FSM. When constructing an AO-FSM aspect, the different types of pointcut and advice types can be composed.

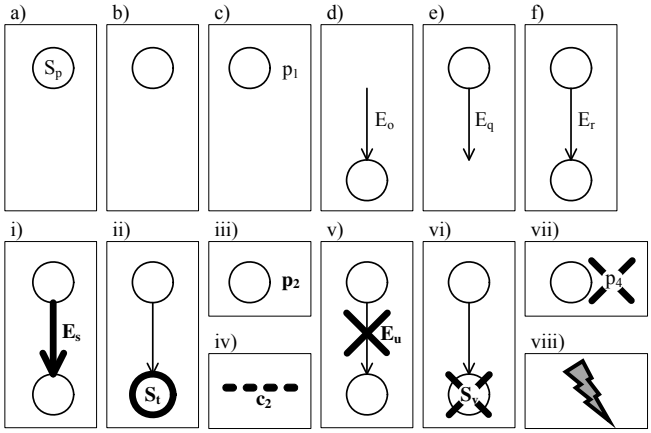

Fig. 2: Types of aspect-oriented finite state machines

There are 6 different kinds of pointcuts: a) matches a labeled state, b) matches any state, c) matches a state that meets a certain preposition, d) matches a state with an incoming transition, e) matches a state with an outgoing transition, and f) matches a sequence of two states with a transition.

The are 6 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 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.

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.

VI. 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, when they compose specifications into an Essential Behavioral Model consisting of nested state machines, they uses AO-FSM aspects to detect interactions that manifest in 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.

For example, there is a feature interaction when we compose the two feature specifications: Call Waiting (CW) and Three Way Calling (TWC). 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?

To identify interactions, the system composes all models using the FSM synchronized cross-product operator of Definition 4, which corresponds to the parallel composition of the state machines of such specifications. It composes feature specifications with the core feature and the aspects.

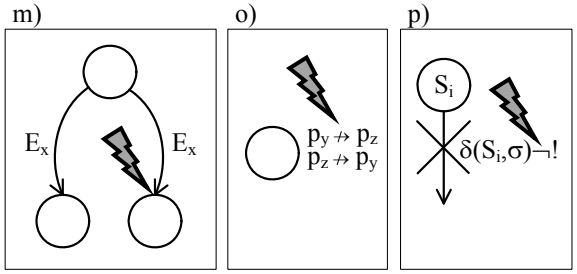

Fig. 3: Three detection aspects checking for composition singularities

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. 3 shows three analysis 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 aspects' 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 [5].

To resolve the conflict, the developer need to specific a set of resolution aspects. Each aspect intercepts the reception of events, and removes a singularity (e.g. non-determinism) from the composed specification. Depending on corresponding context (e.g. the current state and the received events), the aspect can make a choice which of the conflicting features should be active and which not.

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.

For example, consider the feature interaction between CW and TWC. First, the detection aspect in Fig. 3 at index m identifies this non-determinism singularity. Second, the developer specifies the resolution aspect in Fig. 4. That resolution aspect resolves the interaction of the CW and TWC features by defining a precedence between those features that depends on the sequence of previous events. Intuitively, if a call of C arrives on agent A (event E_1) before A presses the flash_back button (event E_3), the CW feature will be active. In this case, the left pointcut in Fig. 4 will match and temporarily remove the transition TWC.E_3. Conversely, if E_3 takes place before E_1, then the TWC feature will be active. In this case, the right pointcut in Fig. 4 will match and temporarily remove the transition CW.E_3.

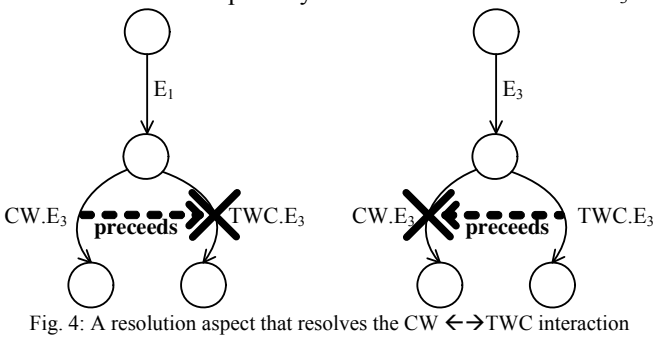

Fig. 4: A resolution aspect that resolves the CW ←→TWC interaction from Section IV.B

VII. Discussion

To validate the approach, we have implemented a prototype of AO-FMS 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. As a proof of concept, the AO-FSM prototype automatically detects the interaction from Section IV.B, and we have developed a resolution aspect to revolve this interaction. We could achieve objectives stated in the introduction, namely the support for separation of concerns (in particular crosscutting features), the formalization of behavior, and dealing with interactions. With the current prototype, conflicts can successfully be detected and resolved. However, correct results depend on whether the developer completely specifies the model and correctly implements aspects with the AO-FSM tool.

Furthermore, 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.

Our prototype implementation only covers feature detection and resolution at design time. For save feature implementation, our approach could easily integrate 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 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.

VIII. Related Work

Most similar is the work 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 for implementation, 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. They can only detect conflicts involving aspects, but they cannot detect interactions between base features as we do.

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 expressions such constraints for a restricted set of domains, such as telecommunications for which special DSLs are available. Currently, developers are lefts 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.

IX. 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) based on finite-state machines and an Essential Behavioral Model (EBM). The EBM defines states and transitions as an FSM, but states and transitions do not need to be completely specified.

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.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. The authors would also like to thank Yassine Essadraoui who has contributed to the implementation of the prototype of AO-FSM 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.