IEEE Copyright Notice

Copyright (c) 2003 IEEE
Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Published in: ***Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering (CBSE'03),*** May 2003

"Run-Time Management of Feature Interactions"

Cite as:

Yinghua Jia and Joanne M. Atlee. 2003. Run-Time Management of Feature Interactions. In *Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering (CBSE'03)*, IEEE Computer Society, Washington, DC, USA.

BibTex:

```
@inproceedings{JiaAtleeCBSE03,
 author = {Jia, Yinghua and Atlee, Joanne M.},
 title = {Understanding and Comparing Model-Based Specification Notations},
 booktitle = {Proceedings of the Sixth ICSE Workshop on Component-Based Software
Engineering},
 series = {CBSE '03},
 year = {2003}
}
```

Printed Proceedings Only

Run-Time Management of Feature Interactions

Yinghua Jia and Joanne M. Atlee
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{ysjia, jmatlee}@se.uwaterloo.ca

Abstract

There is a push to develop feature-rich applications as collections of interconnected feature modules. The problem is that these modules are conceived as independent features, but when strung together, they may interfere with each other because they modify the same shared data (e.g., two features may inconsistently update variables that are encapsulated in a third module). We are studying how to support modular feature development via a framework that interconnects features and that automatically detects and resolves feature interactions. In this paper, we propose a component model for coordinating features and we describe a prototype framework that implements this model.

1 Introduction

The **feature interaction (FI) problem**—how to rapidly add new features to an application without disrupting existing features—is rampant in feature-rich applications, like telephony or insurance systems [2]. Integrating a new feature into an established application traditionally involves analyzing how the new feature might interact (e.g., via shared variables or inconsistent assumptions) with each of the application's existing features and then adapting the new feature to behave appropriately in concert with the existing features. The time needed for interaction analysis and feature adaptation—that is, to search for potential interactions, to analyze their resolutions, and to implement the resolutions—grows with the number of features, as the addition of each new feature aggravates the adaptation of the next feature.

Worse, features are non-monotonic extensions of the system [10]. In particular, two non-interacting features may start to conflict when a new feature is introduced. As a result, the feature interaction problem is not amenable to compositional reasoning. Even an assume-guarantee approach would be tricky, because it is impossible to foresee what assumptions are needed to guarantee interaction freedom.

Instead, we are investigating how to detect and resolve interactions at run-time via a component framework. Features are designed and implemented as independent modules. The framework is parameterized to accommodate any number and variety of feature modules. It includes special feature interaction managers (FIMs) that are responsible for monitoring the features' actions at run-time, for detecting interactions, and for dictating resolutions. The FIMs use a general-purpose strategy for resolving conflicts. Hence, features are implemented without knowledge of the other features that will be executing, and the FIMs are implemented without knowledge of the features they will be managing. Because the resolution strategies are encapsulated in the FIMs, they can be customized for different users or devices. On the down side, general-purpose resolution strategies do not produce ideal resolutions; it is an open problem to determine whether such general resolutions would be acceptable to the user.

So far, we have designed a component model for coordinating features and managing interactions [4] and have implemented a prototype framework that implements this model. Other members of our research group have built a reachability analyzer that reports features' interactions and their resolutions, according to a particular resolution strategy. This paper describes our prototype framework. As background, we give a brief overview of our component model: the information that a feature must provide to facilitate run-time detection of interactions, the type of interactions our model is able to detect, and two example general-purpose resolution strategies based on feature priority and on tolerance. We then describe the components of our framework and the protocols they use to coordinate their actions, and we provide some example scenarios. We conclude with some non-technical questions about the practicality of this approach to managing feature interactions.

2 Features, Interactions, and Resolutions

In this section, we provide a summary definition of our component model in terms of how features are modelled, how feature interactions manifest themselves, and how a general-purpose resolution strategy would realize desired interactions and resolve undesired interactions. A formal

presentation of this work appears in [4].

We treat features as independent modules that provide incremental functionality to an application. Each feature consists of a set of rules that define the feature's behavior. Each rule consists of two parts: a pre-condition that specifies the rule's enabling conditions, and post-conditions that describe the rule's actions that change the system state.

To be specific, the system state is modelled by a set of **facts**, expressed as tuples. Example facts about a telephone system would include voice connections among users, calls on hold, subscription information, call screening lists, etc. A feature rule's pre-condition is a logic formula over these facts; if this formula is satisfied by the system state's facts, then the rule is enabled. A feature rule's post-conditions are changes to the system-state; these changes are expressed as the addition of new facts (*i.e.,* tuples) or the removal of existing facts. A post-condition can also assert or retract a **constraint** that represents a feature-imposed restriction on the system state; constraints are expressed as formulae over the facts. Example constraints in a telephone system would include forbidden connections due to Call Screening, and limits on what information is displayed due to Caller ID Blocking.

A feature executes by examining the current state of the system, determining if any of its rules is enabled in the current state, and, if so, applying the enabled rules' post-conditions, thereby updating the system state. These steps are repeated until the feature terminates. Each application session's features execute synchronously, evaluating and updating the system state simultaneously; application sessions execute asynchronously.

A feature interaction occurs when features' post-conditions conflict with one another. Because post-conditions may add/remove facts from the system state or may add/remove constraints on the system state, there are four types of feature interactions:

1. New facts are inconsistent with each other
2. New facts violate existing constraints
3. New constraint is violated by existing facts
4. New constraint is unsatisfiable with existing constraints

Constraints are logic formulae and facts make up a logic model, so interactions of types 2 and 3 can be detected by testing if the constraints evaluate to true with respect to the new or old system state, respectively. New facts are simply tuples that are added to or removed from the system state, so interactions of type 1 can be detected by testing if added facts are present in and removed facts are absent from the new system state. Interaction type 4 is a satisfiability problem.

As an initial attempt at designing a general-purpose strategy for resolving feature interactions, we proposed in [4] to resolve interaction types 1 and 2 by feature priority [5, 6] and to tolerate [9] interaction types 3 and 4. Resolution by priority works by assigning a total-priority ordering on the features and resolving interactions in favour of the higher-priority feature: given a conflict between the post-conditions of two features' rules, all of the post-conditions of the higher-priority feature's rule are applied and none of the post-conditions of the lower-priority feature's rule are applied. Interaction types 3 and 4, both of which involve newly asserted constraints, are tolerated because doing so allows more enabled rules to react to a system state and because tolerating such interactions does not affect the soundness of the algorithms for detecting and resolving interactions. This way a violated constraint can continue to reject new facts that re-violate the constraint. For example, the Teen Line feature asserts a constraint at 16:00 that prohibits calls made during homework hours, but this constraint affects only new calls made after 16:00 and does not suddenly terminate an established call. A side effect of tolerating interaction types 3 and 4 is there is no need to detect these interactions—which is a major benefit, given that it is expensive to detect interactions of type 4.

Given a set of interacting enabled rules, the resolution strategies described above will approve for execution the maximal subset of non-conflicting rules, such that every rejected rule conflicts (via interaction types 1 or 2) with some higher-priority approved rule. Features are given priority over those services and over older features that they are designed to override. Enterprises can assign higher priority to emergency features (*e.g.,* 911 operator), and users can customize how non-emergency features are prioritized. Hence, our model relies on conflict resolution to realize desired interactions, which are due to feature integration, as well as to resolve undesired interactions.

3 Framework

In this section, we present a framework for feature interaction management. The architecture of the framework is shown in Figure 1. In the following two subsections, we shall describe the functionalities of the components depicted in Figure 1 and the coordination protocol enforced by our framework, respectively.

3.1 Components

In our framework, we refer to the instance of an executing application as an *application session*, or simply *session*. Each session in our framework is uniquely identified by a Session ID and is monitored by a Feature Interaction Manager (FIM). A session comprises several interconnected components. For example in a telephone system, each user is represented by an Agent that includes the user's subscribed features and by an Interface module (IM) that interprets inputs from and outputs to the user's telephone device. A call between a caller and a callee is an application session that is realized by a dedicated FIM monitoring

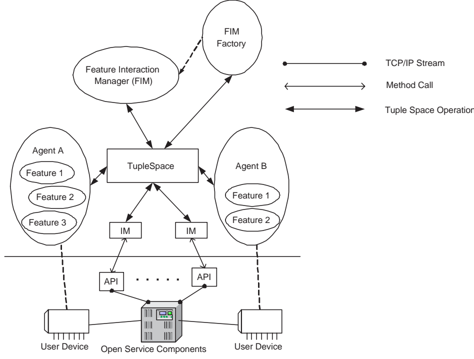


Figure 1. The Framework

this call. To detect and resolve interactions among the features in a session, the session's Agents announce the their features' intended actions to the monitoring FIM, which selects for execution a subset of non-conflicting actions. The communication protocol that coordinates these components is presented in detail in Section 3.2. Here, we introduce the functionalities of each individual component.

A **Feature** component (or simply Feature) stores a set of rules, which specify the services provided and constraints imposed by an application feature. These rules are parameterized by Agent ID and Session ID. A *feature instance* is an instantiation of a feature component for a particular user in an application session. Features behave as described in the previous section: they iteratively examine the system state (stored in the TupleSpace), determine which of their rules are enabled, and write the enabled rules' post-conditions to the TupleSpace. Features never change the system state directly. Instead, they propose changes via their post-conditions, and the FIM approves the features' proposed changes and directs the other components to realize these changes.

The **Agent** component (or simply Agent) is a package of Feature components. An Agent may represent a user, a role, or a device. It always executes, either participating in an application session or listening for a request to start a new application session. At the start of a new application session, participating Agents instantiate their feature components with the appropriate Agent ID and Session ID.

Each **FIM** component manages one application session. It collects the proposed actions from the participating Agents, decides which of these proposed actions can be applied to the system, and changes the system state accordingly. Thus, the FIM component implements the interaction resolution strategy. By separating resolution strategies from Features, our framework enables customization of resolution strategies. It also enables Agents and Features to be developed without knowledge of each other and without knowledge of the resolution strategy. The conflicts among Agents and Features are resolved at run-time, and thus if resolutions affect features' behavior, the features are affected only by actual interactions.

The **FIM Factory** component can be viewed as an FIM generator. Whenever it receives notification of a new application session, it spawns a new FIM process to monitor the new session.

The **TupleSpace** (*TS*) is a distributed shared-memory mechanism for inter-component communication. In addition to ordinary read and write operations, the *TS* provides a subscribe-notification mechanism whereby a component can provide to the *TS* (subscribe) a tuple template, and whenever a tuple satisfying the template is inserted into the *TS*, the component is notified of this event (notification). In our framework, the *TS* stores system-state information comprising facts (including message), constraints, and feature instances' post-conditions to be approved by the FIM. The *TS* also contains actions (*e.g.*, output message Tuples, enable-voice Tuples and disconnect-call Tuples) for the Interface Module to perform, and it contains semaphore Tuples that constrain components' access to the *TS* (described in Section 3.2).

The **Interface Module** (IM) component is responsible for interpreting the external events from devices into Tuples and placing the Tuples into *TS*. Conversely, the IM component is also responsible for realizing features' functionality by monitoring the *TS* for approved feature actions and by invoking appropriate methods provided by the Open Service's APIs.

The **Open Service** component(s) manages multiple User Devices and realizes the features' actions. It behaves as a bridge between the IM and the User Devices by notifying the IM of any event of its interest and by providing to the IM the API that operates on the User Devices. As such, service functionality is implemented in the Open Service component(s) and features act primarily as policies, specifying when services execute. Because the FIM, and not features, specifies which actions to be performed, features have to be truthful to the FIM regarding their intentions.

3.2 Protocol

Because all Features, Agents, IMs, and FIMs share the information stored in the *TS*, we need a communication protocol to synchronize their access to the *TS*.

In general, the components take turns accessing data in the *TS*. First the Agents' features evaluate their rules' pre-conditions with respect to the tuples in the *TS*. Then, they insert into the *TS* the post-conditions of any enabled rule. After all Agents have completed their analyses, the FIM is enabled to evaluate the post-conditions and to detect and resolve interactions. The FIM updates the *TS* with the approved actions from the resolution step, leaving the *TS* in a state ready to be analyzed by the Agents again. At this point, the IMs are notified of the changes to be reflected to

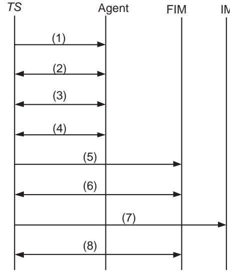


Figure 2. Step by Step Protocol

User Devices through the APIs provided by the Open Service Component, as shown in Figure 1.

This protocol is depicted as a message sequence chart in Figure 2, whose explanation is as follows. Numbers in the text correspond to numbers of the messages in the Figure. (1) The TS notifies all Agents in a session $S1$ that the system state is stable and ready to be examined. (2) Each Agent involved in session $S1$ compares the pre-conditions of each of its feature rules with the Tuples in the *TS*. (3) For each rule whose pre-condition is satisfied, the Agent writes that rule's post-conditions to the *TS*. (4) When the Agent's features have finished reacting to the current system state, the Agent decrements by 1 the semaphore tuple's $count$ value. When all of the session's Agents have finished, the semaphore's $count$ value will be zero. (5) The session's FIM receives notification from the *TS* that the semaphore tuple's $count$ value has reached zero. (6) Next, the FIM removes from the *TS* all of the *post-condition*s related to session $S1$, identifies the maximum subset of non-conflicting *post-condition*s using the provided resolution strategy, and applies the approved actions to the *TS*. (7) The IM component is notified of the changes to the *TS*, which it translates into appropriate calls to the Open Services' APIs. (8) Finally, the FIM resets the semaphore tuple's $count$ value to the number of Agents involved in the session, indicating that the FIM has finished arbitrating among features' intensions.

These steps are repeated until the application session terminates.

The mechanism that we used in our framework to synchronize components' access to the *TS* is a session-specific semaphore tuple, which is also stored in the *TS*. The semaphore tuple contains a field $capacity$, which indicates the number of Agents involved in the session, and a field $count$, whose value indicates how many Agents have yet finished their analyses. When the *TS* is ready to be evaluated by the Agents, the tuple's $count$ value is set to the Tuple's $capacity$ value. Each Agent decrements the tuple's $count$ value by one as the Agent completes its analysis of the system state. This decrement operation is guaranteed by the *TS* to be atomic, so that only one Agent is allowed to access (read and write) the semaphore tuple at any time. When the $count$ value reaches zero, the FIM has exclusive access to the *TS*. When the FIM has completed updating the system state with the features' approved reactions to the previous system state, the FIM sets the Tuple's $count$ value back to the $capacity$ value.

An Agent could at the same time participate in more than one session. To synchronize all of the Agent's feature instances' access to the system state, the Agent will not start evaluating its feature instances until all of its participating sessions are ready for analyses (*i.e.*, the semaphore Tuples' $count$ value for every participating session is greater than zero).

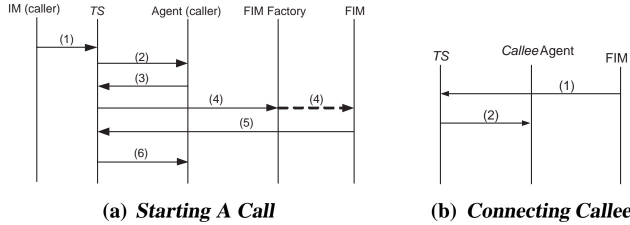
Careful readers might have noticed that this communication protocol could deadlock, since an Agent or the FIM could crash and not respond appropriately to the semaphore tuple. This situation could be avoided by setting up a timeout value for each Agent and for the FIM.

4 Examples

In this section, we describe a prototype implementation of our framework that we use to manage features and feature interactions in an IP telephone system. In this system, the basic telephone service is provided by *Originating Call Model* (OCM) and *Terminating Call Model* (TCM) Feature components, which represent caller and callee functionality, respectively, and by *Plain Old Telephone Service* (POTS) Feature component, which specifies the system behavior once a voice connection has been established (*i.e.*, once callers and callees are indistinguishable from each other). **Agent** components represent the telephone users (in our implementation, the phones do not support features, so they do not have their own Agents). Each feature instance contains the rules for one feature, instantiated for one agent's view of one session. We use Meridian PBX Option 11c as our Open Service component, which provides APIs for receiving external events, for controlling phones' tones, and for switching voice channels. The **IM** components are responsible for translating external events from the Meridian PBX into Tuples and for realizing features' proposed actions by invoking appropriate methods provided by Meridian's APIs. The current implementation of our framework can detect and resolve interactions caused by conflicting actions; it can also capture some constraint violations. In the subsequent sections, we first illustrate how the components involved in a session are initialized to coordinate their access to the *TS*, then we give two examples of capturing conflict and constraint violation interactions. Due to the space limit, the steps related to semaphore Tuple manipulation (step 4, 5 and 8 in Figure 2) are omitted in section 4.3 and 4.4.

4.1 Starting A Call

In this example, we demonstrate how the Agent and FIM components are initialized for coordination when a new call session is started. These initialization steps are shown in



(a) ***Starting A Call*** **(b)** ***Connecting Callee***

Figure 3. Starting a call and connecting callee

Figure 3(a). As before, numbers in the textual explanation correspond to message numbers in the Figure.

(1) The caller starts a call by going off-hook (*e.g.*, by picking up the handset), which is detected as an external event and written into *TS* as a phone-off-hook event Tuple by the IM of the caller. (2) The Caller Agent is notified of this event, and (3) sends a request via *TS* to the FIM Factory to spawn a new FIM for the call session. (4) The FIM Factory allocates an FIM for this new call session. (5) The new FIM creates a semaphore Tuple with initial $count$ 1 (since only the caller is involved in the call at this point) and places the Tuple in the *TS*, after which (6) the caller Agent is notified that the FIM is ready to manage the call (the semaphore has been initialized). At this point, the Agent's features may start evaluating the current system state, and the Agent and the FIM coordinate their access to the system state using the protocol that we discussed in section 3.2.

4.2 Connecting the Callee

A callee Agent becomes involved in a call session when it receives notification of an invitation message (a Tuple in the *TS*) from the caller Agent. The callee Agent must enter the same application session as the caller and be monitored by the same FIM, for the FIM to detect and resolve interactions involving the feature instances of the callee Agent. There are two steps required to join the callee Agent to the call session, as shown in Figure 3(b). (1) When the FIM approves the caller's call invitation to the callee, the FIM increases the semaphore's $capacity$ from 1 to 2 and sets the semaphore's $count$ value to 2. This change reflects the callee Agent being added to the call session, thereby by increasing the number of agents involved in the call to 2. (2) The callee Agent receives notification of the invitation message from *TS*. The callee Agent strips the call session ID from the invitation message, and use this call session ID and its Agent ID to instantiate the features, after this, the Agent participates in the same call session as the caller Agent.

4.3 CFB Overriding TCM

Telephone features such as *Call Forwarding On Busy* (CFB) realize their functionalities by overriding the functionalities of the basic telephony features (OCM, TCM and POTS). In our framework implementation, we simulate this overriding by assigning the features higher priority than the services they explicitly override and by using a priority-based resolution strategy. Suppose user Tom subscribes to

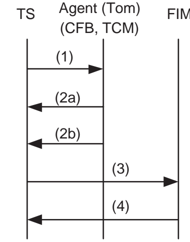
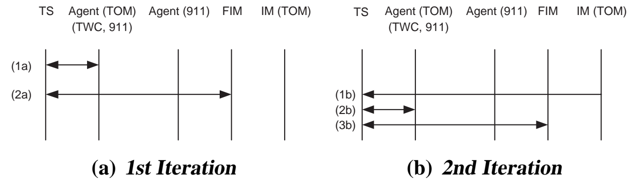


Figure 4. CFB Overriding TCM



(a) ***1st Iteration*** **(b)** ***2nd Iteration***

Figure 5. Constraint Violation

the feature CFB in addition to basic telephone features (see Figure 4). (1) If Tom is already involved in a call when a second call comes in, (2a) the TCM feature of Tom's Agent reacts to the call situation by proposing to send a call-rejected message to the caller; at the same time, (2b) the CFB feature of Tom's Agent reacts to the call situation by proposing to send a call-redirected message to the caller. (3) The monitoring FIM collects the proposed actions, finds these two proposed actions conflict with each other, and ignores the TCM's proposed action since it has lower priority. (4) The FIM adds the call-redirected message action to the *TS*.

4.4 TWC Violates 911hold Constraint

Suppose user Tom made a call to 911 emergency. Suppose also that Tom subscribes to *ThreeWayCalling* (TWC) feature, and he presses hook to start a three-way call, thereby putting the call with 911 on hold. The TWC feature should not get executed because the action of putting 911 on hold violates a constraint asserted by 911 that forbids users from putting the 911 operator on hold.

Two iterations are involved in capturing TWC violating 911hold constraint as shown in Figure 5(a) and 5(b) respectively. The steps involved are: (1a) once a call been made to 911, the 911 feature of Tom's Agent reacts to the call situation by proposing to assert 911hold constraint; (2a) the monitoring FIM collects the proposed actions, finds no conflicts, writes the constraint Tuple into *TS*; (1b) Tom flashes hook trying to start a three-way call, and this action is interpreted as flash-hook event and written into *TS* by Tom's IM; (2b) the TWC feature of Tom's Agent reacts to the call situation by proposing to put the call with 911 on hold, start a new call, ... etc. ; (3b) The monitoring FIM collects the proposed actions, finds the action of putting the call with 911 on hold violates the 911hold constraint, since 911 as an emergency number prohibits anyone putting a 911 operator on hold. Therefore, Tom cannot make the three-way call.

5 Related Work

A number of architectural approaches have been proposed to resolve feature interactions. For example, [7, 8] identify interactions as deviations from features' "signature" behaviors stored in the FIM (one FIM per involved party in a call). [3] proposes an architecture that supports separate logic for features and calls. Feature interactions are represented as conflicting instantiated feature execution states, and interaction detection requires *a priori* knowledge of features, thus violating feature modularity. [11, 1] propose agent-based architectures that detect and resolve feature interactions through agent negotiation (event-dispatching and precedence rules, fuzzy logic). None of the above approaches detects or resolves constraint violations.

There are also approaches that focus on automatic off-line analysis using formal methods (specification languages and verification tools). These approaches detect and resolve feature interactions by controlling explicitly how groups of features behave together. As we discussed in the introduction, this effort grows exponentially with the number of features, because feature introduction is non-monotonic and thus new features have to be analyzed with all combinations of existing features.

Our work is also related to rule-based systems. A key difference is that rule-based systems tend to resolve interactions by introducing new rules to cover the offending cases, thus leading to a proliferation of rules. Such an approach is analogous to the approaches described in the previous paragraph, in which an application includes code that deals explicitly with how groups of features should behave when executed together. Such approaches do not scale to systems that have hundreds and thousands of features. Hence, we are investigating the effectiveness of general-purpose resolution strategies.

6 Questions and Future Work

There are some obvious questions about this approach's feasibility that need to be tested—most notably, questions about performance, about the usability of the feature-rule language, and about how best to allocate and coordinate distributed Feature Interaction Managers.

The approach also needs to be extended to deal with a wider variety of interaction types. The approach currently handles conflicts and constraint violations that occur in practice and that need to be resolved. There are temporal types of interactions that we choose not to resolve and thus do not need to detect; these interactions include **data interactions**, whereby one feature reacts to data values written by another feature, and **race conditions**. The next interaction type we intend to address is features' non-conflicting reactions to the same input event, where the non-conflicting reactions leave the system in an undesirable state. We believe we can address these interactions using invariant system constraints.

A more fundamental question is whether we can identify or create a general-purpose resolution strategy whose resolutions will be acceptable to users. At least in the telephony domain, users are accustomed to having interaction-specific resolutions built into the system, as part of the feature integration process. As such, most users are not even aware of the feature interaction problem and are used to high-quality functionality. A general-purpose detection and resolution strategy is likely to produce satisfiable resolutions most of the time, sub-optimal resolutions some of the time, and no resolution on rare occasions. Thus, with this approach to feature integration, there is a tradeoff between users' satisfaction with overall system behavior vs. rapid feature development and maintainability.

References

[1] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii. Feature-Interaction Resolution Using Fuzzy Policies. In *Feature Interactions in Telecommunications Systems*, pages 94–112, 2000.

[2] E. Cameron, N. D. Griffeth, Y. Lin, M. E. Nilson, and W. K. Schnure. A Feature Interaction Benchmark for IN and Beyond. In *Feature Interactions in Telecommunications Systems*, pages 1–23, 1994.

[3] N. Fritsche. Runtime Resolution of Feature Interactions in Architectures with Separated Call and Feature Control. In *Feature Interactions in Telecommunications III*, pages 43–64, 1995.

[4] J. D. Hay and J. M. Atlee. Composing Features and Resolving Interactions. In *Foundations of Software Engineering*, pages 110–119, 2000.

[5] S. Homayoon and H. Singh. Methods of Addressing the Interactions of Intelligent Network Services with Embedded Switch Services. *IEEE Communications*, 26(12):42–70, December 1998.

[6] D. Marples and E. Magill. The Use of Rollback to Prevent Incorrect Operation of Features in Intelligent Network Based Systems. In *Feature Interactions in Telecommunications and Software Systems V*, pages 115–134, 1998.

[7] S. Tsang and E. H. Magill. Detecting Feature Interactions in the Intellingent Network. In *Feature Interactions in Telecommunications Systems*, pages 1–23, 1994.

[8] S. Tsang and E. H. Magill. Run-Time Feautre Interaction Detection. In *Feature Interactions in Telecommunications Systems*, pages 254–270, 1997.

[9] A. van Lamsweerde and E. Letier. Integrating Obstacles in Goal-Driven Requirements Engineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 53–63, 1998.

[10] H. Veldhuijsen. Issues of Non-Monotonicity in Featur-Interaction Detection. In *Feature Interactions in Telecommunications III*, pages 31–42, 1995.

[11] I. Zibman, C. Woolf, P. O'Reilly, L. Strickland, D. Willis, and J. Visser. Minimizing Feature Interactions: An Architecture and Processing Model Approach. In *Feature Interactions in Telecommunications III*, pages 65–84, 1995.