Distributed Resolution of Feature Interactions for Internet Applications

Rui Gustavo Crespo [a], Miguel Carvalho [b], Luigi Logrippo [c]

[a] *INESC-ID / Technical University of Lisbon, DEEC*
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
Email: R.G.Crespo@comp.ist.utl.pt, Phone: +351-21 8418 398

[b] *Polytechnic Institute of Lisbon*
Av. Conselheiro Emidio Navarro 1, 1950-062 Lisboa, Portugal
Email: mcarvalho@cc.isel.ipl.pt, Phone: +351-21 8317 067

[c] *Université du Québec en Outaouais, Département d'Informatique et Ingénierie*
Case Postale 1250, Succ. B, Gatineau QC J8X 3X7, Canada
Email: luigi@uqo.ca, Phone: +1-819 595-3900, ext 1885

Abstract

Internet applications, such as Email, VoIP and WWW, have been enhanced with many features. However, the introduction and modification of features may result in undesired behaviors, and this effect is known as feature interaction-FI.

After a brief review of FI detection principles, we propose an interaction resolution adviser, consisting in two phases. The first phase implements an initial selection, by filtering the features that satisfy a set of formulas. We describe several strategies according to the nodes that participate in the FI resolution. The second phase selects the feature for execution, and adapts parameters, according to user policies.

The interaction resolution adviser is distributed, scalable and independent of the applications and their features.

Key words: Feature interaction; Interaction detection; Interaction resolution; Interdictions; Obligations.

[1] This research has been partially funded by the Fundação Ciência e Tecnologia of Portugal, and the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Internet applications are being enhanced with many features. A feature is defined as a unit of functionality existing in a system and is usually perceived as having a self-contained functional role [1]. The combination of features may result in undesired behaviors and this problem is known as *feature interaction*, or FI for short [2]. The FI problem, first identified in circuit-switched networks, has been studied in many Internet applications, such as Email [3,4], VoIP [5] and WWW [6].

Example 1 *Suppose that Bob instructs the Email server to execute the* ForwardMessage *feature, forwarding all messages to Carl. Suppose also, that Carl subscribes to the* AutoResponder*, by activating the Unix vacation program.*
A message that Alice sends to Bob is forwarded to Carl. Thereafter, the Email server of Carl accepts Alice's message and sends a notification message to Bob, not to the message initiator (Alice). This result goes against the initial goal of AutoResponder *feature, to notify the initiator that Carl is on vacation. The Email server of Bob, when it receives the notification message, forwards it back to Carl. The Email server of Carl detects a loop, another undesired behavior, and discards the notification message.*

The increasing number of FIs, and the inconvenience they are causing, has led industry and researchers to meet regularly at the *Feature Interactions in Telecommunications and Software Systems* workshops, eight of which have been held from 1992 to 2005. Three basic problems have been studied, *avoidance*, *detection* and *resolution*. Avoidance means to intervene at the protocol or design stages to prevent FIs, before features are executed. Detection aims at the identification of FIs, with suitable methods. In the resolution, actions are exercised over already detected FIs.
The distributed nature of Internet, with multi-vendor and multi-provider environments, and the end user capability to program and tailor features, makes it impossible to rely on avoidance.

In this paper, we focus on detection and resolution, and propose an Interaction Resolver Adviser, IRAdv, which is scalable, independent of the feature implementation, and unique for all Internet applications.
Briefly, when an application identifies several features that may process a message (for example, an incoming or outgoing SIP INVITE [7]), the list of these features is sent to the IRAdv.
Then, the adviser selects the feature to be executed on the basis of policies expressed by two sets of logical formulas. The first set guides the IRAdv in the removal of FIs. We adopt the interdiction operator to express the first set of policies, due to the localized action of such operator. The second set guides the selection of the feature to be executed, among those selected in the

filtering phase. We adopt the obligation operator to express the second set of policies, to ensure that only one candidate is selected. Finally, the adviser communicates its decision to the application.

The outline of the paper corresponds to the outline of our method. First, one needs to know, by static analysis, which pairs of features are likely to interact. Section 2 deals with this problem, with a brief review of known methods for static feature interaction detection. The information gathered at this step will be useful to program the policies of the IRAdv, as discussed below.
The rest of the paper deals with execution time FI resolution on the basis of this information. After a brief review of the problem of FI resolution, in Section 3, Section 4.1 introduces the IRAdv. A number of basic actions will have to be identified in the system (Section 4.2). When the application requests one or more features, it sends to the IRAdv a message, requesting for advice on the execution of a list of basic actions related to the features (Section 4.3). On the basis of the information gathered in the step illustrated in Section 2, we assume that the adviser has been previously programmed with policies that set priorities among basic actions, in consideration of various elements such as the other actions being requested, the status of the systems, etc. These policies cause the filtering phase (Sections 4.4 and 4.5) to "interdict" certain actions from being executed under certain conditions. All interdicted actions will be removed from the list. Then, the adviser proceeds to the final selection phase (Section 4.6), using another set of policies that identify certain features as compulsory, in view of the final contents of the list.
Finally, we discuss some implementation issues of IRAdv for an Email server (Section 5).

2 Interaction Identification

Several taxonomies have been produced to classify different types of FI [8–10]. The most used benchmark of [8] identifies three dimensions, the number of parties involved in the interaction (single or multiple), the number of network components involved in the interaction (single or multiple), and the kind of features involved in the interaction (custom or system). In this paper, we concentrate on custom features, because system feature interactions are more easily preventable at design phase.

In order to set up mechanisms for execution-time FI resolution, it is useful to know in advance which features are likely to conflict. This can be done by using one of the many static FI detection algorithms known in the literature. Formal methods, such as extended finite state automata [11], Petri-nets [12], process algebra [13], temporal logics [14] and theorem proving [15], have been used to represent features. As well, FIs can be detected with standard tools

of model checking or simulation [16]. A review of several existing methods is given in [17].
According to the analysis presented in [18–20], many feature interactions are characterized by inconsistencies that arise by the responses of two or more features.

2.1 Shared Trigger Interactions

STI [18–20] occur in cases where two features can be triggered by the same message and their action responses are inconsistent.

Example 2 *Consider the* CFA-*Call Forward Always and* CW-*Call Waiting pair of features, in the VoIP, triggered when a SIP call invitation arrives.* CFA *response calls the new destination (different from the features' subscriber) and* CW *response puts the initiator on waiting status to connect the features' subscriber. The responses are mutually inconsistent, because a call cannot be forwarded and put on hold simultaneously.*

2.2 Sequential Action Interactions

SAI [18–20] occur in the case where there is a chain of different features that trigger each other, and the actions of two of them are inconsistent.

Example 3 *The pair of* ForwardMessage *and* FilterMessage *Email features are an example of SAI.*
Suppose that Carl subscribes to FilterMessage *feature, to deny messages from Alice. If Alice sends an Email to Carl,* FilterMessage *rejects the message. However, if Bob subscribes to* ForwardMessage *feature, forwarding all messages to Carl and if Alice executes the* WriteMail *basic service to Bob, then the message is forwarded to Carl. This action is inconsistent with the* FilterMessage *intended behaviour.*

2.3 Missed trigger interactions

MSI [19] occur when the actions of one feature prevents a second feature from being executed.

Example 4 *Suppose that Alice subscribes to* CFA *to Charles in the VoIP, and Bob subscribes to* AR-*Automatic Recall. If Alice calls Bob with success,*

and thereafter Bob instructs SIP client to execute AR*,* CFA *prevents Bob from talking to Alice.*

2.4 Looping interactions

In the previous three cases, inconsistency exists between action responses of two features. However, inconsistency can also exist between the results of the combination of features and basic system requirements. One such requirement is that there should not be infinite loops, and this the the case for LI [18–20].

Example 5 *Two different users, subscribing to the* ForwardMessage *to each other, form a loop.*

Although these methods detect candidates for FI, users must decide if they represent undesirable interactions that must be resolved. For example, the web Refresh feature forms a loop to itself. However, if the web page depicts a clock, Refresh becomes an acceptable interaction.
The FI methods based on these definitions have been shown to detect many practically occurring interactions.

3 Feature Resolution

When several potentially conflicting features are up for execution, the conflict must be resolved by deciding which feature can go ahead. The resolution of FIs requires two elements, a model of the relationships between features and an algorithm.
Many models of relationships between features have been proposed, from familiar data structures (such as tables [21]) to logic formulas [22]). The relationships may not express directly the possible interactions between features. It is the algorithm that knows what are the FIs and, by looking at the relationships between the features, selects the feature for execution.

The current methods of FI resolution are divided into two large classes, design and runtime resolution.
Design resolution removes interactions at the development phase of the features, but this approach requires previous knowledge of all existing features. A good overview of design-time techniques is given in [17]. However as desirable as design time techniques may be, the always evolving Internet makes runtime resolution mandatory.

Three approaches have been explored for FI runtime resolution: one phase, two phase and negotiation [17].

The one phase approach uses feature managers to decide which feature is executed, according to tables. The table may be used for a list of priorities [21,23], or the feature manager decides the activation of a second feature on the basis of the resulting status of the first feature and the relation displayed in the table [24]. Because [23] uses a stack to express priority relationships between features, relationships between interactions may be modified as result of feature execution. Tables can be replaced by state trees, where the features suggest possible responses and the feature manager "rolls back" in case of rejection [25].
Although simple, the one phase approach requires knowledge about low-level details, suffers from the scalability problem, and reveals problems in the resolution of multiple user FIs.

In the two phase approach, the feature is executed in an isolated environment and actions are taken in case of FI [26]. The isolated environment is impractical in Internet.

Three negotiation approaches have been proposed, direct (agents negotiate directly without a negotiator), indirect (dedicated negotiator controls the negotiation and proposes solutions based on experience), and arbitrated (the negotiator has the sole responsibility to find a solution).
Direct negotiation, based on distributed artificial intelligence techniques [27], consumes too much processing power and communication bandwidth.
Indirect negotiation [28] uses a centralized inference machine to select the feature from the constraints expressed by agents.
In the arbitrated approach, the features are described as agents and interact with each other, by posting their intentions to a common tuple space. The replies, from other features, can be other intentions or permission/interdiction directives. Negotiation policies explored include deontic-based ones with obligations and goals [22], fuzzy policies [29] and relational assertions [30]. The resolution of new interactions require the introduction of new policies to cover the offending cases, making the approach unscalable.

The distributed character of the Internet makes the direct negotiation the most suitable approach.
In [19] a "prune and extract" approach is proposed, where a solution space is computed and undesired traces are removed. We propose to avoid the computation of a solution space, by modeling relationships between features with constraint formulas. Our constraint formulas are created by the user or administrator according to experience (see Section 4.4).

Applications, such as Email servers, resolve some FIs such as looping. In our view, an independent FI resolver is still necessary because it makes possible to solve other types of FIs, enables users to understand and adapt the FI resolution methods to their specific needs, and for developers of new Internet applications reduces the need to duplicate code.

4.1 IRAdv architecture

The Internet architecture is based on a stack-protocol architecture [31]. The top-level applications interface with an *Application Programming Interface*-API, which is an endpoint for communication. The most widely used APIs are sockets, and the commands include establishing connections, sending and accepting messages. Features are triggered by message arrivals, such as Email delivery or SIP INVITE, or by party commands, such as web page view requests. When such an event occurs, the IRAdv is invoked.
The architecture of the *Interaction Resolver Adviser*-IRAdv follows the client-server model [31] and is depicted in figure 1.

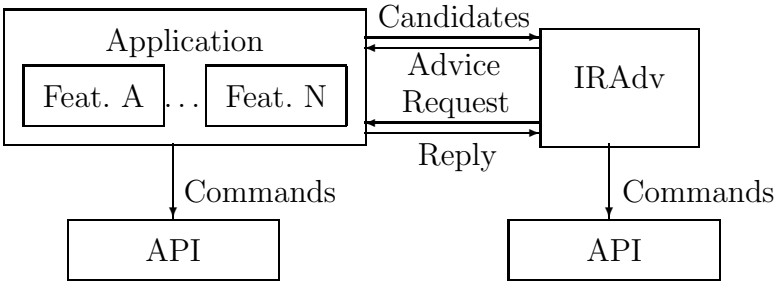

Fig. 1. The Architecture of Interaction Resolver Adviser

The interaction between the application, the client, and the IRAdv, the server, is resolved in two steps. First, the application sends to the IRAdv a list of feature candidates for execution. In return, IRAdv advises the application about which feature should be executed.

For single user FIs, it is sufficient that the local node implements the IRAdv. For multiple user FIs, we require that all involved nodes have an IRAdv, and that these can communicate.
The communication capability between IRAdvs is used to grant permission for message processing (see section 4.4.3). In this case, IRAdv sends to the application a request for more information and waits for the application's reply. This communication capability assumption is not unreasonable, since other QoS considerations will require the implementation of uniform architectural enhancements on all nodes. The nodes that do not implement communication

capabilities between IRAdvs will still be able to process messages, but without QoS guarantees.
We believe that the FI resolver should be unique in the node, to avoid inconsistencies in the FI resolution of different applications. Also, FI resolver must be unaware of the message contents.

4.2 Feature representation

The number of features can be large. For example, short lists of 17 telecommunication features, easily adapted to VoIP [8], and 10 Email features [3], are available in the literature. Real systems can have hundreds of features. Hence, there is a need to identify a compact representation of the feature space, to avoid a large matrix of relationships in the FI resolver.
Work about the compact representation of feature space has been made available recently, but is restricted to telephony [32]. We propose to compact the representation through a set of basic and abstract actions that each feature executes. Information inside the message remains hidden to the IRAdv and system actions, such as billing, are excluded from our model.
Actions, depicted in table 1, may have one, or more, parameters. For our example in table 1, all actions have one parameter that can be *_init*, the node that started the message, or *_dest*, the node to whom the message should be delivered (that may be changed later, for example as result of a forward), or *_self*, the node currently processing the message. Our set of basic actions is not, by no means, unique or exclusive. We have selected this set for the sake of illustration and other sets, probably larger, can be selected, depending on the feature set in consideration.

Table 1
Feature basic actions

Name	Purpose
Accept(_init)	The node accepts the message
Deny(_init)	The node rejects the message
Display(_init)	The node displays initiator and waits for terminator to confirm the message acceptance
Emergency(_init)	The node must accept the message, with highest priority
Forward(_dest)	The node redirects the message to another node
Send(_dest)	The node initiates a message to another node
Wait(_init)	The node puts the message on hold, for later processing

Two different features may be represented by the same basic actions. For example, the VoIP CFA and CFB-Call Forward if Busy features are both rep-

resented by the *Forward* action.
One feature may be represented by more than one action. For example, the Email AutoResponder feature is represented by *Accept* and *Send* join of actions. The feature representation may also change according to local status. For example, the application represents the FilterMessage feature by *Deny* (*Accept*) when the initiator does (does not) belong to a "black list" of nodes.

The mapping of features to actions should be defined by the entity responsible for the feature installation. The system administrator is a good candidate, because features may be hard to separate in some applications and domains.

4.3 Application requests

When the application has a request for a service, incoming or outgoing, more than one feature may be selected as candidate for service execution. Then, the application sends to the IRAdv a message with structure divided into two parts, the header and the information elements (one element for each feature candidate).

$$\begin{aligned} Candidates &::= Header\ Information \\ Header &::= ApplicationPort\ InitialAddr\ ChainIP^{+} \\ &\quad\ ApplicationStatus\ ConfirmDirective \\ Information &::= (FeatureCode(Action\ (NodeIP)^{+}))^{+} \end{aligned} \tag{1}$$

The header part contains information common to all selected features.

- The *ApplicationPort* element indicates which application requires advice. Its code is the port number attached to the associated application protocol, as registered by the *Internet Assigned Number Authority*. For example, Email applications follow the SMTP protocol [33] and communicate through port 25.
- The *InitialAddr* element contains the address of the party that initiates the message. Actions with _init parameter bind the value of *InitialAddr* to _init.
- The *ChainIP* element provides the IP addresses of all nodes that have processed the message, including the initiator. The purpose of this element is made clear in section 4.4.2.
- The *ApplicationStatus* element provides information about the application status. The structure and the meaning of the information depend on the application, and we use identifiers with lowercase letters. The status conditions represent sequences of triggered features and provide extra support in the resolution of FIs.
- The *ConfirmDirective* element indicates the kind of confirmation to be asked of the initiator node, and is described in section 4.4.3.

In our proposal, *InitialAddr*, *ChainIP* and *ConfirmDirective* values must be transmitted together with the messages sent from one node to another.
The information attachment to messages has been used by applications to resolve some FIs. For example, Email attaches the *InitialAddr* and *ChainIP* addresses to messages, respectively, in *From* and *Received* header fields, to detect looping. In section 4.4.2 we will use these values to resolve loops.

The information part contains information specific to the selected features. There can be several features, each represented by one or more actions.

- The *FeatureCode* is a unique identifier for the feature. Its purpose is to link features to the actions that represent them, and it does not participate in the FI resolution in the filtering phase.
- The *Action* element lists the actions representing the feature with *NodeIP* list containing the involved IP addresses.
  Actions with _dest parameter bind the value of *Forward* to _dest.

The IRAdv returns the feature code to be executed, or nothing (in case of failure in FI resolution).

$$Advice ::= \varepsilon \mid FeatureCode \tag{2}$$

4.4 Constraint relationships

To express relationships between features, we propose constraint formulas. Reasons for our choice includes the similarity of the representation to human knowledge, the easier implementation of the FI resolver, and the successful application of drop actions in the iptables IP packet filter [34].
The user, or the administrator, is responsible for the identification of the constraint relationships, and this work is done after the identification of the candidates for FI, as described in section 2.
The constraint formulas are expressed in the form depicted in (3). Section 4.5 further explains the semantics of formula (3). Examples are depicted in sections 4.4.1 to 4.4.4.

$$Requests \wedge Conditions \rightarrow Restrictions \tag{3}$$

The *Requests* part is a conjunction of actions, representing features selected by the application as candidates for execution.
The *Conditions* part identifies the values that the application status satisfy, or IRAdv identify (to be seen in sections 4.4.2 and 4.4.3). By default, this part equals to *true*. Examples of application status are the permission to accept a message, $permission(\_init)$, the application status (e.g. $one\_hold(\_init)$, that

states that a VoIP call is on hold), and the identification of a loop occurrence, $loop(\_init, \_dest)$.
The *Restrictions* part identifies the single action, or the join of actions, whose execution is forbidden.

Formula (3) is expressed in the first-order predicate language [35], with the *Interdiction* unary connective I ($I\mathcal{A}$ subformula means that action $\mathcal{A}$ cannot be executed). In this paper, we assume that I holds the highest operator precedence, $\neg$ the second highest, and $\rightarrow$ the lowest precedence.
We adopted the interdiction operator for (3) following our approach to FI resolution through feature elimination. Also, the interdiction operator only affects actions participating in the formula. Other actions, representing different features, may be proposed or not, but our attention is focused only on the pairs of features that interact.

There are three different types of constraint formulas which apply to different layouts of the nodes that participate in the resolution. We call these local, chain and point-to-point constraints. Local constraints are directed to the resolution of single user FIs, while chain and point-to-point constraints are directed to the resolution of multiple user FIs.
Constraint relationships may be different for different applications, even if features are represented by the same set of actions. For example, Email forward and WWW redirect features are represented by the same action (Forward). The FI between Forward and Accept may be resolved by Forward interdiction for the Email server and Accept interdiction for the WWW server.

4.4.1 Local Constraints

Formula (3) is a local constraint when it only involves the node of the feature subscriber. Interdiction and priority are two examples of local constraint formulas.

With interdiction, we simply forbid one of the actions (and hence, the feature that executes such action).

Example 6 *To resolve the FI in example 2, the first call is put on hold and all the other calls are forwarded.*
When there are no calls on hold, the first formula below only forbids the execution of the Forward *action. In this case, nothing is said about the* Wait *action, which may be decided by another formula (for example,* Wait *may not be executed in presence of an* Emergency *action), or by the final selection mechanism.*

$$Forward(\_dest) \wedge Wait(\_init) \wedge \neg one\_hold(\_init) \rightarrow I\ Forward(\_dest)$$

$$Forward(\_dest) \wedge Wait(\_init) \wedge one\_hold(\_init) \rightarrow I\ Wait(\_init)$$

The ApplicationStatus *is also used to resolve the multiple-user multiple component interaction [8] of* CW *and* ACB*-Automatic CallBack features.*
Suppose that Alice subscribes to ACB *feature that is triggered when the destination is busy, to redial the destination when it becomes idle. Suppose also that Bob subscribes to* CW *feature.*
If Alice calls Bob and if Bob is busy, Alice's call is put on hold and Alice receives a notification that she is on a wait condition. When Bob closes the call, CW *will establish a connection from Bob to Alice. After closing the second connection,* ACB *establishes a connection from Alice to Bob.*
To resolve this FI, ACB *cannot be executed when a call to the destination is on a waiting condition. We represent the* ACB *feature by the sending of a message, to ask the destination to notify the subscriber when he returns to the idle condition.*

$$Send(\_dest) \wedge waiting(\_dest) \rightarrow I\ Send(\_dest)$$

Action priorities are easily specified in local constraint formulas. We define $AHigh$ as the set of actions with priority above $ALow$ set of actions, with the formula $AHigh \wedge ALow \rightarrow I\ ALow$.

Example 7 *The Email* FilterMessage *feature and* ReadMail *basic service are represented, respectively, by* Deny *and* Accept *actions. The* Deny *action has higher priority, and this requirement is expressed by the formula*

$$Accept(\_init) \wedge Deny(\_init) \rightarrow I\ Accept(\_init)$$

To make Emergency *take precedence over all other actions, the formula is*

$$Emergency(\_init) \rightarrow I(all \setminus \{Emergency(\_init)\})$$

where $\setminus$ *is the set difference. The interdicted actions are not directly listed, because other basic actions may be inserted later in table 1.*

4.4.2 Chain Constraints

As seen in section 2.4, looping is a possible FI. There are two possible solutions for this problem. The first, which is actually used in IP, is to place a *Time-to-live* field in packets [36]. We choose to describe another solution where every mode appends its IP address to forwarded messages. The first solution is simpler and does not require variable packet sizes, but imposes an

upper limit for the nodes in a loop.
In our case, the IP addresses are added to the *ChainIP* element. The initiator node starts an empty *ChainIP* element and all successive nodes, including the initiator, extend the $LocalAddressIP$ element with their IP address. The IRAdv checks if the IP address of the current node appears twice in the *ChainIP* list: if this is the case, the predicate value of $loop(\_init, \_dest)$ is set to *true*, otherwise is set to *false*.

The formula that forbids a forward to enter a loop is

$$Forward(\_dest) \wedge loop(\_self) \rightarrow I\ Forward(\_dest)$$

Note 1 *Because the installation of FI resolver cannot be done simultaneously in all nodes, in some cases the* ChainIP *value may be lost. In this case, administrators may take a conservative approach and set* $loop(\_self)$ *to* true*, or a liberal approach and set* $loop(\_self)$ *to* false*. In the conservative approach, only messages in the links where all nodes have IRdv can be forwarded.*

4.4.3 Point-to-Point Constraints

In point-to-point constraint formulas, the execution of a feature in one node requires extra information from another node, usually the one that initiates the communication.
For example, suppose that Alice subscribes to FilterDestination feature, to avoid the WWW browser to access Bob's page. Suppose also, that Carl subscribes to Refresh(Bob) feature. If Alice opens Carl's page, it is Bob's page that appear in her browser. To resolve this FI, the initiator WWW server must confirm the WWW read from the destination computer page.

More in general, in case of forwarding, the original sender should be given the possibility of keeping control of where the message is going. Authorization may be required at each forward or for final delivery only.
The initiator application must identify the sort of confirmation it requires for the message. The possible values, shown in (4), are *None*-no confirmation is required, *AllNodes*-confirmation is required for every node where the message is forwarded or delivered, and *Destination*-confirmation is required only for the final delivery. The confirmation is verified by the IRAdvs. The IRAdv of the initiator node requests of the initiator application permission for the other node to process information (forward or deliver the message).

$$ConfirmDirective ::= None \mid AllNodes \mid Destination \tag{4}$$

The overhead of the confirmation may be acceptable in some cases, such as

Email or SIP, and too heavy in other cases, such as video on demand or voice transmission. Therefore, performance issues may have to be considered in the selection of the *ConfirmDirective* value.

The frame messages exchanged between the two IRAdvs comply with the syntax of (5).

$$Frame ::= FrameCode\ FeatureCode\ ApplicationPort\ InitialAddr\ DestinationAddr \quad (5)$$

The *FeatureCode*, *InitialAddr* and *DestinationAddr* elements are the same of (1). Table 2 lists possible values of the *FrameCode* element and their meaning.

Table 2
Frame codes

Name	Meaning
Delivery_request	Proposes to accept the message
Delivery_acceptable	Message may be delivered
Delivery_denied	Message must not be delivered

When the terminator IRAdv receives a Delivery_acceptable, or Delivery_denied frame, it defines the predicate value of $permission(\_init)$ according to the *Code* element in the frame reply: *true* if it is *Delivery_acceptable* and $false$ otherwise.

Example 8 *Figure 2 depicts the message sequence chart for the WWW request, depicted in the second paragraph in this section, with* ConfirmDirective *equals to* Destination. *The initiator application is Alices's browser and the terminator application is Bob's WWW server.*

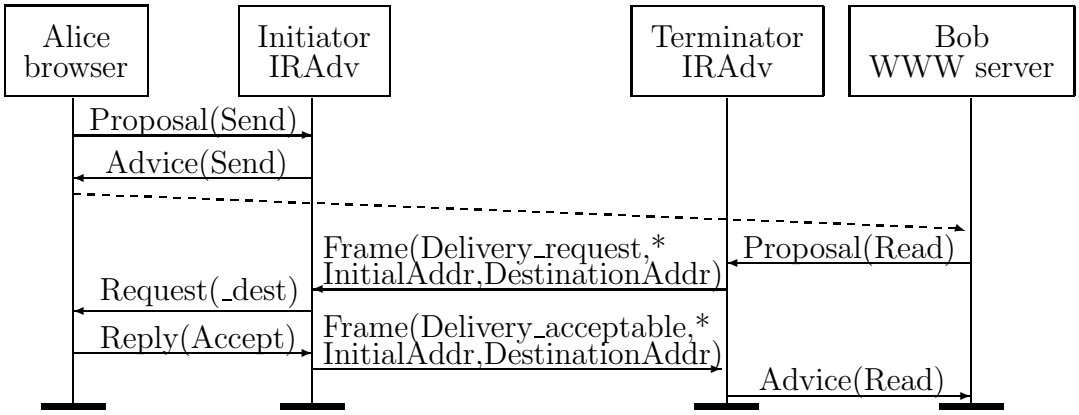

Fig. 2. WWW request with permission grant from WWW browser

Alice's WWW browser sends to the initiator IRAdv a message proposing a Send *action, which returns an advise to send the WWW read request.*
Carl's server, not depicted in the figure 2, proposes to its IRAdv to forward to Bob's node the read request, as result of Refresh *feature.*
Bob's node IRAdv sends a frame to Alice's IRAdv to ask permission to conclude the WWW request. Being irrelevant to the example, FeatureCode *and* ApplicationPort *parameters are represented in the figure by a star. The dashed vector represents the time dependency between Alice's browser and Bob's server.*
Alice's IRAdv, in turn, requests Alice's browser permission to conclude the read request, and informs Bob's IRAdv of the result. The permission is granted and Bob's server is advised to conclude the read request. If the permission had been denied by Alice's browser, Bob's server would receive a null advice.

Note 2 *Because the installation of FI resolver cannot be done simultaneously in all nodes, in some cases the* ConfirmDirective *value may be lost. In this case, the default approach may depend on the terminator application. For example, the content of some WWW pages may require an explicit confirmation from the initiator and the default value for* $permission(\_init)$ *would be* false.

Example 9 *The formula that resolves the multiple-user single-component interactions [8] of* ReadMail *and* FilterDestination*, and of* Call Forward *and* Originating Call Screening *is*

$$Accept(\_init) \wedge \neg permission(\_init) \rightarrow_I Accept(\_init)$$

4.4.4 Features represented by multiple actions

Actions of different features are connected by the $\wedge$ operator, and actions representing the same feature are joined together by the *join* predicate. The *join* predicate is a result of the possible representation of features by more than one action, and we don't want actions to be transferred from one feature representation to another. If $\mathcal{A}$ $\mathcal{B}$ and $\mathcal{C}$ are actions, $join(\mathcal{A}, \mathcal{B}) \wedge \mathcal{C}$ is equivalent to $\mathcal{C} \wedge join(\mathcal{A}, \mathcal{B})$ or to $\mathcal{C} \wedge join(\mathcal{B}, \mathcal{A})$, but not to $\mathcal{A} \wedge join(\mathcal{B}, \mathcal{C})$. Because we use a first-order predicate language, parameters of the *join* predicate are constants that mirror table 1 actions.

Example 10 *To interdict* AutoResponder *feature to send a reply to the postmaster, the formula is*

$$join(Delivery(\_dest), Send(\_self)) \wedge \_dest = postMaster \rightarrow_I Send(\_self)$$

The semantics of the I operator is very simple: $I\mathcal{A}$ is satisfied if, and only if, action $\mathcal{A}$ is not executed.
The satisfaction of a restriction part must be verified when the request and the condition are evaluated to true. The interdiction effect is, simply, to mark the removal of the feature associated to the basic actions from the list of candidates. After the evaluation of all existing formulas, the marked features are removed from the list and the remaining features are presented to the final selection phase.

The order in which the constraint formulas are evaluated is irrelevant. The evaluation only marks features for removal, and the removal of feature candidates for execution is implemented after the evaluation of all formulas. Also, the evaluation of the formulas does not change the application status.
The number of constraint formulas is limited, and each formula is verified only once. Hence, resolution is completed in linear time.
The identification of a new action only implies the insertion of new formulas concerning the interaction between the new action and the others. Constraint formulas, which exist before the introduction of the new action, do not involve features represented by the new action. The inclusion of a new feature implies, at most, the introduction of new constraint formulas that resolve the new FIs that now may occur. Therefore, new actions and new features do not require the modification of existing constraint formulas, and our approach is scalable.

The fact that several actions can be joined in a feature raises the question of what will happen if only some actions in a feature are interdicted.

- The *interdiction maximization* approach interdicts a feature if, at least, one of the actions that represents it in the join is interdicted.
- The *conditional interdiction* approach interdicts an action join if, at least, one of the actions that is joined is interdicted, but not all, and there are actions outside the join that are not interdicted.
- The *survival maximization* approach interdicts an action join only when all of the actions that are joined are interdicted.

Example 11 *Suppose that an user subscribes to Email* AutoResponder *and* FilterMessage *features. Suppose also, that a member of his "black list" sends a message to the user. In this case, the* ReadMail *basic service and the* AutoResponder, FilterMessage *features are represented, respectively, by* Accept, join(Accept,Send) *and* Deny *actions.*
By the constraint formula of example 7, the Accept *action is interdicted. The interdiction maximization and conditional interdiction approaches result in interdiction of the* join(Accept,Send) *action. In this case, IRAdv would advise*

the Email server to execute the FilterMessage *feature.*

Because the node administrator may select different approaches for the join interdiction, the decision on removing a join is taken after the single basic actions are marked for removal.
The resolution phase removes interactions by interdicting features. There is no guarantee, however, that the set of surviving features is single. Therefore, there is a need for a second phase only if the cardinality of the surviving feature set is non-singular.

4.6 Selection Algorithm

The execution of the filtering phase results in a (possibly empty) set of "survivor" actions. Actions are linked to features, as defined by (1), hence the selection algorithm chooses the feature to be executed.
When the number of surviving features is greater than one, each feature may be executed without FI occurrence. If there are no surviving features, the filtering mechanism of the resolution phase cannot satisfy the goal of selecting one feature. This potential problem is debated in section 7.

It is useful to think that these features are organized in an arbitrarily ordered list, to facilitate the choice. The system administrator may identify some features that have higher priority. In this case, these features may have to be reordered in the list, but again, if there are several, only the first is chosen.

The selection algorithm follows a similar approach to the initial selection, with two major differences. The entities under scrutiny are features, not basic actions, because the goal of the selection algorithm is the identification of one single feature that the application should execute. The deontic operator of interdiction is replaced by obligation, because it enables the selection of one feature independent of the number of surviving features.

The selection formulas are expressed in the form shown in (6).

$$Conditions \rightarrow Obligations \tag{6}$$

The formula is expressed in the usual first-order predicate language [35], with the addition of the *Obligation* unary connective O.
The IP addresses *_init* and *_dest*, as defined in section 2, are obligation formula variables. In the final selection phase, we add two variables *_self* and *_carrier* which bind, respectively, to the IP addresses of the subscriber's node and of the last node that sent the message.
The size of the filtered sequence is stored in the variable *size*, and its values

can be *null*, *one* or *many*. The selection algorithm has access to the results of the initial selection algorithm, through a pair of variables: the head of the filtered sequence, *head*, and the *tail*, which is the list of the remaining features (this can be empty).
The predicates include the advice for feature execution-*advise(feature)*, a message display to the administrator-*display(message)*, and the existence of a specific feature in the tail list-$\mathcal{F}$ *in tail*. *null* is used in case the filtering algorithm removes all features.

The *Conditions* part is a classical first-order predicate formula. To make the final selection algorithm decidable, the conditions must be complete (i.e., their disjunction must be a tautology) and mutually exclusive (i.e., the conjunction of any pair of predicates must be a falsity) [35]. It is responsibility of the administrator to assure that this is the case.
The *Obligations* part is an *advise* feature action. The administrator may also introduce a conjunction with another action, the *display* of a message. Both actions must contain the O operator.
$O\mathcal{A}$ is satisfied if and only if action $\mathcal{A}$ is executed. We may take a conservative approach that, if the *Conditions* part is not satisfied, the relative actions are not executed. In this case, for each $\mathcal{C} \rightarrow O\mathcal{A}$, the selection algorithm automatically generates a $\neg\mathcal{C} \rightarrow I\mathcal{A}$ formula.

Example 12 *Consider a simple selection algorithm that selects* AutoResponder *with a notification message sent to the initiator, if present in the list sent by the filtering phase. If not present, the selection algorithm selects the head feature. If the list is empty, a warning statement is sent to the administrator. The* AutoResponder *feature is represented with one parameter, the node address to where the notification message is sent.*

$$
\begin{aligned}
&size \neq null \wedge (head = AutoResponder(\_carrier) \vee \\
&\quad AutoResponder(\_carrier)\ in\ tail) \ \rightarrow O\ advise(AutoResponder(\_init)) \\
&size \neq null \wedge (head \neq AutoResponder(\_carrier) \wedge \\
&\quad \neg AutoResponder(\_carrier)\ in\ tail) \ \rightarrow O\ advise(head) \\
&size = null \rightarrow O\ display(cat(''Empty\ resolution\ in\ terminal\ '', \_self)) \wedge \\
&\quad O\ advise(null)
\end{aligned}
$$

where cat is a function that catenates two strings.
In example 1, Alice may ask herself why she receives the notification from Carl when she only knows Bob. Clearly, this problem results from the distributed nature of the Internet and users may adopt other selection algorithms.

5 Implementation Issues

We tested our FI adviser approach with the open source James-Apache Java Enterprise Mail Server. James has been used for many Email applications, such as spam detection and SMS notification. The prototype is available from http://comp.ist.utl.pt/FI-resolver.htm.
Section 5.1 provides a brief presentation of the James architecture. The implementation of our proposal required the James adaptation for feature personalization (section 5.2), and the postpone of feature execution (section 5.3). Section 5.4 indicates how the confirmation directives are fixed and section 5.5 describes some experimental results of our implementation.

5.1 James architecture

James is made of two agents, message transfer and mail processing.
The message transfer agent moves Email messages between nodes and follows Internet protocols such as SMTP [33], mail message formats [37] and message retrieve (POP3 [38] and IMAP [39]).
The mail processing agent, named SpoolManager, is an implementation of the Mailet Java API [40]. The Mailet API defines interfaces for *matchers*, which determine whether a message should be processed, and *mailets*, which implement features. Email messages are processed by a chain of *processors*, the first and the last ones named *root* and *transport*. Processors are defined in the *config.xml* file, with processor tag and name attributes.

Each *processor* has zero or more *mailet* child elements. Mailet attributes match and class define, respectively, conditions for selecting an Email message, and the object class that processes the Email message. If at least a match occurs, the message is processed by the mailet. If not, the message is passed to the following *mailet* in the chain.

Example 13 *The* transport *processor checks if the Email recipient is for a local account. Condition* RecipientIsLocal *is checked for every recipient in the RCPT TO command, and one instance of* LocalDelivery *class processes the Email message.*

```
<process name="transport">
  <mailet match="RecipientIsLocal" class="LocalDelivery"/>
</process>
```

Because James does not support feature customization, we extended the mailet container with three new mailets, that provide functionalities for feature subscription, feature removal, and feature match/execution.

We store feature parameters in a tree hierarchy form. Each feature is linked to a directory, which contains subdirectories for every user that subscribes to the feature. The addresses necessary for the service execution are filenames, stored in the user subdirectory. For example, if Alice requires *FilterMessage* to block messages from Bob@demo, an empty file names Bob@demo is stored in the Features\FilterMessage\Alice directory.
User subscription, modification and unsubscription of features is similar to the mailing lists manager Majordomo [41]. Messages are sent to the localhost, with username equals to the name of the feature subscribed (if unsubscribed, the feature name is prefixed with *Un*) and address in the subject line. In the example of the previous paragraph, Alice forbids the Email server to accept messages from Bob, by sending an Email message to FilterMessage@localhost with subject equals to Bob's address.

Example 14 *The configuration of the FilterMessage feature is depicted next*

```
<mailet match="MatchSingleRecipient=FilterMessage@localhost"
        class="AddAddressToUserFeature">
  <folder>../apps/james/var/features/FilterMessage </folder>
  <subject>New address added to the FilterMessage list: </subject>
  <content>Send a message to UnfilterMessage@localhost with this address
           on subject, to reset.</content>
</mailet>
```

Folder *element contains the directory where feature data is stored.* Subject *and* content *elements contain, respectively, the header and the acknowledge message sent to the FilterMessage subscriber.*

We also developed a new kind of matcher, *MatchUserWithFeature*, to check if an Email recipient matches users listed in the feature data repository.

Example 15 *The following extract of config.xml depicts FilterMessage match and the comment is sent to a log file, in case of execution.*

```
<mailet match="MatchUserWithFeature=
                ../apps/james/var/features/FilterMessage"
        class="FilterMessage">
  <folder>../apps/james/var/features/FilterMessage </folder>
  <subject>Message Filtered: </subject>
  <content>Your message was filtered by destination user. </content>
</mailet>
```

The object, that actually checks if a recipient is a member of the blocked users, is designated in the attribute class.

5.3 IRAdv integration

As explained in section 5.1, James executes the feature as soon as there is a match. However, IRAdv architecture (section 4.1) requires all feature candidates for execution to be presented to him all at once. To solve this inconsistency, James was modified to the architecture depicted in figure 3.

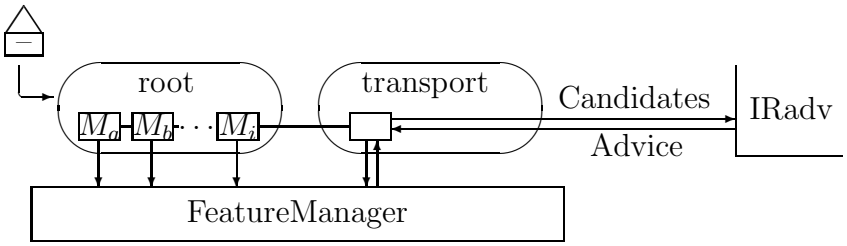

Fig. 3. Extended James architecture

In case of a match, mailets do not execute the feature, but invoke a method at FeatureManager object, which stores the feature as a candidate for execution. At the end of the mailet chain, a special mailet asks advice to IRAdv and executes the selected feature.

5.4 Confirmation directives

Confirmation directives, issued by the Email sender (see section 4.4.3), are inserted in the subject header with <directive> tag, directive class and type. Confirmation types may be *ALL_NODES*-sender must confirm message processing in all nodes, and *DESTINATION*-sender must confirm only message delivery at the final destination.

The IRAdv implementation was tested in Windows 2000 and XP in 1.6 GHz PCs, connected by a 100 Mbps LAN.
We tested three different FI resolution cases and compared delivery time with, and without, IRAdv resolver. Email deliveries suffered varying degrees of degradation, according to the processing cases exercised by IRAdv. In some particular cases, the Email delivery was faster with IRAdv. In the worst case, with automatic delivery confirmation from the sender, delivery time increased twofold.

5.5.1 Simple resolution

Simple resolution formulas are special cases of (3), with no conditions and no action joins.
We experimented four different cases with IRAdv holding the Accept/Deny formula, listed in the example 7. The cases add a feature to the previous cases and are: (SR1) no services subscribed, (SR2) ForwardMessage subscribed, (SR3) SR2 features plus FilterMessage subscribed, and (SR4) SR3 features plus AutoResponder subscribed.

Data is depicted on the left of figure 4. White and black columns represent, respectively, the Email delivery time without and with IRADv.

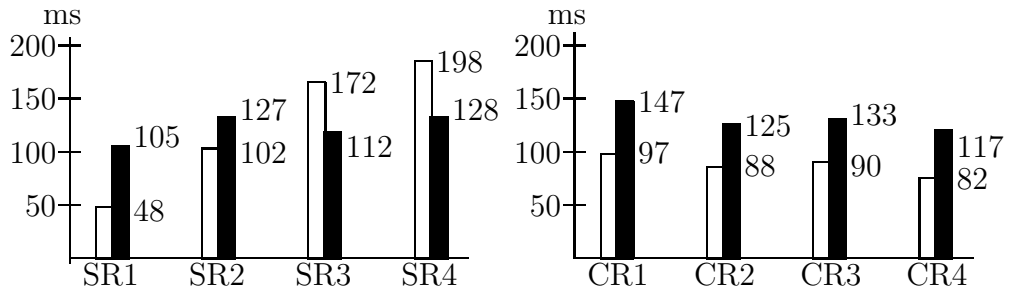

Fig. 4. Simple and Conditional resolution experiments

In cases (SR1) and (SR2), we verified that the communication between the James server and IRAdv takes between 40 and 60 ms. Cases (SR3) and (SR4) reveal a decrease in the total processing time, because the communication between the James server and IRAdv takes less time than the analysis of conditions for the three unnecessary features.

5.5.2 Conditional resolution

For this set of experiments we added, to the Simple resolution formulas, formulas with conditions. The formulas are: (CR1) AutoResponder reply interdicted

to the *Postmaster* sender, as depicted in example 10 (CR2) AutoResponder reply interdicted, if sender belongs to an Email distribution list, (CR3) FilterMessage interdicted, if sender is the president, and (CR4) FilterMessage interdicted, if sender belongs to an Email distribution list.
Data depicted on the right of figure 4 shows that IRAdv increases the Email delivery time between 43% (case CR4) and 52% (case CR1).

5.5.3 Chain resolution

This experiment checks the chain constraints, described in section 4.4.2.

We used the loop detection formula between two parties, and results are similar to the (SR2) experiment.
For the experiment described in example 9, the Email delivery time increased by 113%, from 48 to 102 ms, due to the need to obtain permission for processing the Email in every node.

6 Related Work

Distributed FI resolution has been centered on traditional telecommunication systems, where the system architecture is rather uniform.
The *Agent Architecture* of [42] restricts itself to conventional telephony over a fixed network.
The DFC-*Distributed Feature Composition* virtual architecture [43] represents features as boxes that may be composed in a pipe-and-filter network. DFC requires detailed description of features and call processing. Moreover, pipe-and-filter architectures tend to serialize features' reactions to each message, which causes features to miss key message parts, that they would otherwise react to.

A method for the integration of FI detection has already been published [19], where features candidate for execution are forked and wait from the Feature Manager for a decision to continue or to die. Rules direct the Feature Manager to prune trace subtrees, in order to curb state explosion and remove interactions. However, the approach does not allow permission grants from parties.

The adaptation of features to the distributed Internet system, with different applications, multi-vendor and multi-provider environments raises new challenging problems for FI resolution.
Recent work on FI resolution in distributed platforms adopts a two-level approach to describe features, functional ("hard-logic") and resolution ("soft-logic") [4]. Both levels require programming skills, which we feel should be

avoided at the resolution level.
The idea of cooperating FI resolvers has been proposed recently, with central FIMA-*Feature Interaction Management Agents* to coordinate the resolution operations [44]. In case of failure of one or more FIMAs, the resolution is compromised.

7 Perspectives

Internet is nowadays essential to a wide range of activities. Moreover, the Internet is continuously expanding, with new services enhancing the most popular applications. However, problems also limit the expansion of the Internet and its usefulness. Among others, we focus on the undesired modification of feature behavior after the incorporation of new features.
In this paper, we list simple methods to identify FIs, and describe our proposal for a FI resolution adviser, scalable and independent of the applications and their features.

The set of features and application statuses may be large. In our prototype, the canceling features are reported to a log file. There is a need to check if, for a given set of constraint formulas Φ, there is a subset of features and a subset of status values that make Φ completely eliminate all feature candidates for execution. In cases like the one of section 4.4.3, the complete elimination causes no problems. However, there is a need to check if the complete elimination is due to the wrong choice of constraint formulas. For example, if $\Phi = \{\mathcal{A} \wedge \mathcal{B} \rightarrow IA, \mathcal{B} \wedge \mathcal{C} \rightarrow IB, \mathcal{C} \wedge \mathcal{A} \rightarrow IC\}$ and if the features represented by actions $\mathcal{A}$ $\mathcal{B}$ and $\mathcal{C}$ are candidates for execution, then no feature would be selected for selection.
The decision problem is known as a SAT-*Boolean satisfiability problem*. Clearly, a greedy algorithm would have an exponential complexity. Research is needed to check if more efficient algorithms may be used. Such algorithms would make it possible to notify node administrators about problems that may occur in FI resolution prior to adviser use.

Acknowledgements

This work was completed at the School of Information Technology and Engineering University of Ottawa/Canada, whom the authors thank for the support. We express our gratitude for the remarks and ideas of Prof. Daniel Amyot and Jacques Sincennes of University of Ottawa, and Tom Gray of Pinetel.

References

[1] L. Blair, T. Jones, S. Reiff-Marganiec, A Feature Manager Approach to the Analysis of Component-Interactions, in: 5th Intl Conference on Formal Methods for Open Object-based Distributed Systems, 2002, pp. 233–248.

[2] T. Bowen, F. Dworak, C. Chow, N. Griffeth, G. Herman, Y.-J. Lin, The feature interaction problem in telecommunication systems, in: 7th Intl Conference on Software Engineering for Telecommunication Systems, 1989, pp. 59–62.

[3] R. Hall, Feature Interactions in Electronic Mail, in: M. Calder, E. Magill (Eds.), 6th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 2000, pp. 67–82.

[4] J. Pang, L. Blair, Separating Interaction Concerns from Distributed Feature Components, in: U. Assmann, E. Pulvermueller, I. Borne, N. Bouragadi (Eds.), Electronic Notes in Theoretical Computer Science, Vol. 82, Elsevier, 2003.

[5] J. Lennox, H. Schulzrinne, Feature Interaction in Internet Telephony, in: M. Calder, E. Magill (Eds.), 6th Intl Workshop on Feature Interactions in Telecommunication and Software Systems, IOS Press, 2000, pp. 38–50.

[6] M. Weiss, Feature Interactions in Web Services, in: D. Amyot, L. Logrippo (Eds.), 7th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 2003, pp. 149–156.

[7] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, H. Handley, E. Schooler, SIP:Session Initiation Protocol, rfc 3261, Internet Engineering Task Force, 2002.

[8] E. Cameron, N. Griffeth, Y.-J. Lin, M. Nilson, W. Schnure, A feature interaction benchmark for IN and beyond, in: L. Bouma, H. Velthuijsen (Eds.), Intl Workshop on Feature Interations in Telecommunications Systems, IOS Press, 1994, pp. 1–23.

[9] J. Blom, Formalisation of Requirements with Emphasis on Feature Interaction Detection, in: P. Dini, R. Boutaba, L. Logrippo (Eds.), 4th Intl Workshop on Feature Interations in Telecommunication Systems, IOS Press, 1997, pp. 44–50.

[10] R. Hall, Feature Combination and Interaction Detection via Foreground/ Background Models, in: K. Kimbler, L. Bouma (Eds.), 5th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 1998, pp. 232–246.

[11] D. Méry, J. Gibson, Telephone feature verification: Translating SDL to TLA+, in: A. Cavalli, A. Sarma (Eds.), 8th SDL Forum, 1997, pp. 103–118.

[12] M. Nakamura, Y. Kakuda, T. Kikuno, Petri-net based detection method for non-deterministic feature interactions and its experimental evaluation, in: K. Cheng, T. Otha (Eds.), 3th Intl Workshop on Feature Interations in Telecommunication Systems, IOS Press, 1995, pp. 138–152.

[13] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, T. Ware, Feature description and feature interaction analysis with Use Case Maps and LOTOS, in: M. Calder, E. Magill (Eds.), 6th Feature Interactions in Telecommunications and Software Systems, IOS Press, 2000, pp. 274–289.

[14] J. Blom, R. Bol, L. Kempe, Automatic Detection of Feature Interactions in Temporal Logics, in: K. Cheng, T. Otha (Eds.), 3th Intl Workshop on Feature Interations in Telecommunication Systems, IOS Press, 1995, pp. 1–19.

[15] A. Gammelgard, J. Kristensen, Interaction Detection, a Logical Approach, in: L. Bouma, H. Velthuijsen (Eds.), Intl Workshop on Feature Interations in Telecommunications Systems, IOS Press, 1994, pp. 178–196.

[16] M. Calder, A. Miller, Using SPIN for feature interaction analysis-a case study, in: 8th Intl SPIN workshop on model checking of software, Springer-Verlag, 2001, pp. 143–162.

[17] M. Calder, M. Kolberg, E. Magill, S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks 41 (1) (2003) 115–141.

[18] R. G. Crespo, L. Logrippo, T. Gray, Feature Execution Trees and Interactions, in: The 2002 Intl Conference on Parallel and Distributed Processing Techniques and Applications, 2002, pp. 1230–1236.

[19] M. Calder, M. Kolberg, E. Magill, D. Marples, S. Reiff-Marganiec, Hybrid Solutions to the Feature Interaction Problem, in: D. Amyot, L. Logrippo (Eds.), 7th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 2003, pp. 295–312.

[20] N. Gorse, L. Logrippo, J. Sincennes, Formal Detection of Feature Interactions with Logic Programming and LOTOS, Software and System Modeling 5 (2) (2006) 121–134.

[21] N. Fritsche, Runtime resolution of feature interactions in architectures with sperated call and feature control, in: K. Cheng, T. Otha (Eds.), 3th Intl Workshop on Feature Interations in Telecommunication Systems, IOS Press, 1995, pp. 43–63.

[22] R. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, S. Mankovski, Feature-Interaction Visualization and Resolution in a Agent Environment, in: K. Kimbler, L. Bouma (Eds.), 5th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 1998, pp. 135–149.

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

[24] M. Cain, Managing Run-Time Interactions Between Call-Processing Features, IEEE Communications 30 (2) (1992) 44–50.

[25] D. Marples, E. Magill, The Use of Rollback to Prevent Incorrect Operation of Features in Intelligent Network based Systems, in: K. Kimbler, L. Bouma (Eds.), 5th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 1998, pp. 115–134.

[26] S. Tsang, E. Magill, Behaviour Based Run-Time Feature Interaction Detection and Resolution Approaches for Intellinge Networks, in: P. Dini, R. Boutaba, L. Logrippo (Eds.), 4th Intl Workshop on Feature Interations in Telecommunication Systems, IOS Press, 1997, pp. 254–270.

[27] H. Velthuijsen, Distributed Artificial Intelligence for Runtime Feature-Interaction Resolution, IEEE Computer 26 (8) (1993) 48–55.

[28] N. Griffeth, H. Velthuijsen, The Negotiating Agents Approach to Runtime Feature Resolution, in: L. Bouma, H. Velthuijsen (Eds.), Intl Workshop on Feature Interations in Telecommunications Systems, IOS Press, 1994, pp. 217–235.

[29] M. Amer, T. Karmouch, S. Gray, T. Mankovski, Feature interaction resolution using fuzzy policies, in: M. Calder, E. Magill (Eds.), 6th Feature Interactions in Telecommunications and Software Systems, IOS Press, 2000, pp. 45–63.

[30] J. Hay, J. Atlee, Composing Features and Resolving Interactions, in: ACM Intl Symposium on Foundation of Software Engineering, 2003, pp. 110–119.

[31] D. E. Comer, Computer Networks and Internets, 4th Edition, Prentice-Hall, 2004.

[32] D. Pinard, Reduce the Feature Interaction Problem in Communication Systems Using an Agent-Based Architecture, in: D. Amyot, L. Logrippo (Eds.), 7th Intl Workshop on Feature Interations in Telecommunication and Software Systems, IOS Press, 2003, pp. 13–22.

[33] J. B. Postel, Simple mail transfer protocol, RFC 821, IETF (Aug. 1982).

[34] B. MacCarty, RedHat Linux Firewalls, Addison-Wesley, 2003.

[35] A. G. Hamilton, Logic for Mathematicians, Cambridge University Press, 1988.

[36] A. B. Johnston, SIP: Understanding the Session Initiation Protocol, Artech House, 2003.

[37] D. H. Crocker, Standard for the format of arpa internet text messages, RFC 822, IETF (Aug. 1982).

[38] J. Myers, M. Rose, Post office protocol - version 3, RFC 1939, IETF (May 1996).

[39] M. Crispin, Internet message access protocol - version 4rev1, RFC 2060, IETF (Dec. 1996).

[40] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, 3rd Edition, Addison-Wesley, 2000.

[41] A. Schwartz, Managing Mailing Lists, O'Reilly, 1998.

[42] I. Zibman, C. Woolf, P. O'Reilly, L. Strickland, D. Willis, J. Visser, Minimizing Feature Interactions: An architecture and processing model approach, in: K. Cheng, T. Otha (Eds.), 3th Intl Workshop on Feature Interations in Telecommunication Systems, IOS Press, 1995, pp. 65–83.

[43] M. Jackson, P. Zave, Distributed Feature Composition: A Virtual Architecture for Telecommunications Services, IEEE Trans. on Software Engineering 24 (10) (1998) 831–847.

[44] A. Chentouf, S. Cherkaoui, A. Khoumsi, Experimenting with Feature Interaction Management in SIP Environment, Telecommunication Systems 24 (2) (2003) 251–274.