AN EXPERIMENTAL DISTRIBUTED RESOLUTION OF WWW INTERACTIONS

Fernando Miguel Carvalho
Instituto Superior de Engenharia de Lisboa, Dept. de Engenharia Electrotécnica de Telecomunicações e Computadores
Rua Conselheiro Emidio Navarro 1, 1950-062 Lisboa, Portugal
mcarvalho@cc.isel.ipl.pt

Rui Gustavo Crespo
Technical University of Lisbon, Dept. of Electrical Engineering and Computers
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
R.G.Crespo@comp.ist.utl.pt

ABSTRACT

Likewise older Internet applications, such as Email and VoIP, the creation of greater number of WWW services make inevitable the occurrence of undesirable feature interactions.
The feature resolution on WWW must follow the basic constraints of Internet and, therefore, must be distributed.
In this paper we provide a brief introduction of feature interaction problem. Then, we depict one architecture to resolve the feature interactions and, finally, present an implementation of feature interaction resolution advisor based on deontic logics and Java.

KEYWORDS

WWW interactions, distributed resolution, deontic formulas, object-oriented implementation.

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 (Blair et al. 2002).

The combination of features may result in undesired behaviours and this problem is known as *feature interaction*, or FI for short (Bowen et al. 1989).

The FI problem, first identified in circuit-switched networks, has been studied in many Internet applications, such as Email (Hall 2000), VoIP (Lennox and Schulzrinne, 2000) and WWW (Weiss, 2003).

As a simple example of WWW FI, consider the page *redirect* feature, implemented by the meta tag in the header <meta http-equiv= "Refresh" content=newURL>. If two pages redirect to each other, a loop is formed and this FI may be considered as undesirable.

The incorporation of Internet application in Web services, such as Web mail, increase the number of feature interactions and such issue is expected to raise concerns in the community.

Three basic problems have been studied in the FI area, *avoidance*, *detection* and *resolution*.

Avoidance means to intervene at 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 WWW, with multi-provider environments, and the end user capability to program and tailor features, makes it impossible to rely on avoidance.

Recent work on FI resolution in distributed platforms adopts a two-level approach to describe features, functional and resolution (Pang, 2003). Both levels require programming skills, which should be avoided at the resolution level.

Cooperating FI resolvers have been proposed recently, with FIMA-Feature Interaction Management Agents to coordinate the resolution operations (Chentouf, 2003). In case of failure of one or more FIMAs, the resolution is compromised.

In this paper, we focus on resolution of WWW interactions, and present a scalable IR Adviser, which is implemented by the platform-independent Java programming language (Arnold and Gosling, 2000).

2. IR ADVISER

We propose to attach, to every node in the Internet, one IR Advisor. The IR Advisor adopts the client-server model and the architecture is depicted in figure 1.

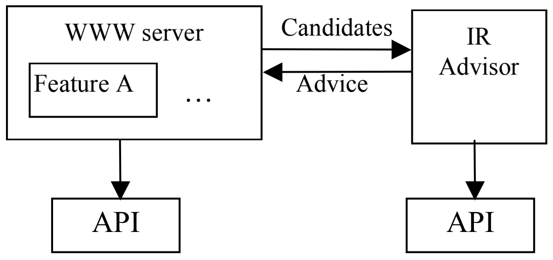


Figure 1. The architecture of IR Advisor

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

For single user FIs, it is sufficient that the local node implements the IR Advisor. For multiple user FIs, we require that all involved nodes have an IR Advisor, and that these can communicate.

The communication capability between IR Advisors is used to grant permission for message processing. That is the case for reserved access to Web pages. In this case, the IR Advisor sends to the application a request for more information and waits for the application's reply.

The IR Advisor selects the feature to be executed in two phases. First, it excludes features that raise FIs, by applying a set of constraint formulas. Then, the IR Advisor selects one feature from the set of survival candidates for execution, by applying a set of selection formulas.

2.1 Feature Representation

The number of features may be very large. To make the IR Advisor as generic as possible, each feature is represented by a set of basic actions. We have selected a group of 6 basic actions, such as *Accept*, *Deny*, *Forward* and *Send* a message. For example, the WWW *Refresh* feature is represented by the *Transfer* action.

Although the initial phase works upon basic action, their result is a set of survival features. This set is expected to be much smaller, because the goal of the initial phase is to resolve all FI by elimination.

The contraint and selection formulas may involve predefined message values, such as _init (the message initiator), _dest (the current message destination) and _carrier (the node, that currently is processing the message).

2.2 Constraint Formulas

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 (MacCarty, 2003). The formulas have the form

$\text{Requests} \wedge \text{Conditions} \rightarrow \text{Restrictions}$ (1)

The *Requests* part is a set of actions, representing features selected by the application as candidates for execution.

The *Conditions* part identifies the values that the application status satisfy (for example, the user has enough disk space to download a file), or IR Advisor identify (for example, the Web application has granted permission for the user to download a file). By default, this part equals to true.

The *Restrictions* part identifies the single action, or the join of actions, whose execution is forbidden. This part uses the Interdiction operator.

For example, the Deny action holds a higher priority that the Accept action. This constraint formula is expressed by

Accept(_init) ∧ Deny(_init) → I Accept(_init)

The formula that forbids a forward to enter a loop is

Forward(_dest) ∧ loop(_init,_dest) → I Forward(_dest)

2.3 Selection Formulas

The selection algorithm follows a similar approach to the initial selection, with three major differences: (i) the entities under scrutiny are features, not basic actions (ii) the deontic operator of Interdiction is replaced by Obligation (iii) the *Requests* part is dropped. The formulas have the form

Conditions→Obligations (2)

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 other list of features (possibly none), *tail*.

The *Conditions* part is a classical first-order predicate formula.

The *Obligations* part is an *advise* feature action.

For example, we may select the WWW get basic feature, if presented in the list of survival features. If not present, we select the first feature.

size≠null ∧ (head=Get(_carrier)∨ Get(_carrier) in tail) → O advise Get(_carrier))
size≠null ∧ (head≠Get(_carrier) ∧ ¬ Get(_carrier)in tail) → O advise(head)
size=null→O display(”No resolution”) ∧ advise(null)

3. IMPLEMENTATION

Now we can identify the principal requirements for IR Adviser system:

- ⇨ The IR Adviser must be prepared to answer for several requests of feature candidates, made by an application;
- ⇨ It must have the modify constraint (1) and selection (2) formulas, as result of pressures for new features and new FI resolution methods;
- ⇨ The resolution process of multiple requests should run on a multi thread model, where a huge list of feature candidates, from one request, should not compromise the response of a smaller list from another request;
- ⇨ To resolve a certain feature interaction, such as blocking page views, it must be able communicate with another IR Adviser from a different node.

Finally, the IR Adviser must run all this processes without compromise its performance on giving advices to applications. To solve these requirements we have organized the IR Adviser functions in three main components, *Resolver*, *Interpreter* and *Listener*.

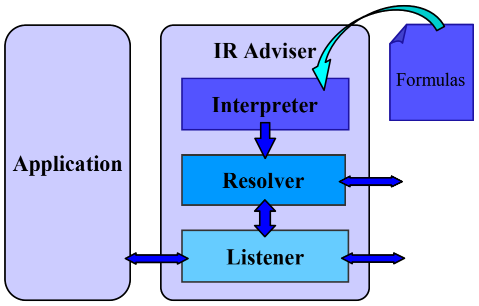


Figure 2. IR Advisor components diagram

Each component has a specific role, which will be explained on the next three subsections. Finally, on the fourth subsection we will explain how an Internet application could integrate an IR Adviser.

All Java classes developed belong to the `irs` (Interaction Resolution System) package, organized in four subpackages: irs.adv (Adviser), irs.app (Application) and irs.msg (Messages).

3.1 Resolver

The Resolver is the central component of IR Adviser architecture. It implements the IR Adviser core functionality on the determination of an advice (a selected feature) for a list of feature candidates, sent by an application.

This process runs around two distinct entities: formulas and candidates, represented by instances of the classes `Formula` and `MsgCandidates`, respectively.

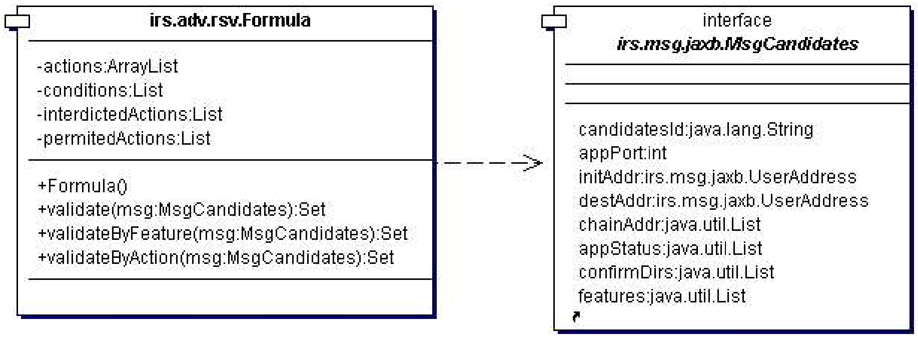

irs.adv.rsv.Formula
-actions:ArrayList
-conditions:List
-interdictedActions:List
-permitedActions:List
+Formula()
+validate(msg:MsgCandidates):Set
+validateByFeature(msg:MsgCandidates):Set
+validateByAction(msg:MsgCandidates):Set
interface
irs.msg.jaxb.MsgCandidates
candidatesId:java.lang.String
appPort:int
initAddr:irs.msg.jaxb.UserAddress
destAddr:irs.msg.jaxb.UserAddress
chainAddr:java.util.List
appStatus:java.util.List
confirmDirs:java.util.List
features:java.util.List

Figure 3. Class diagram of Formula and `MsgCandidates`

When an application sends to the IR Adviser a list of feature candidates for execution, the Listener processes that message and creates a `MsgCandidates` instance, which will be sent to the Resolver. For every message received by the Resolver, a new thread is launched for the execution of the resolution process, represented by a `Resolution` instance. When this thread has finished, an advice has been determined and it will be returned to the application through the Listener, which implements the `IResolutionListener` interface (see AppHandler class at figure 5).

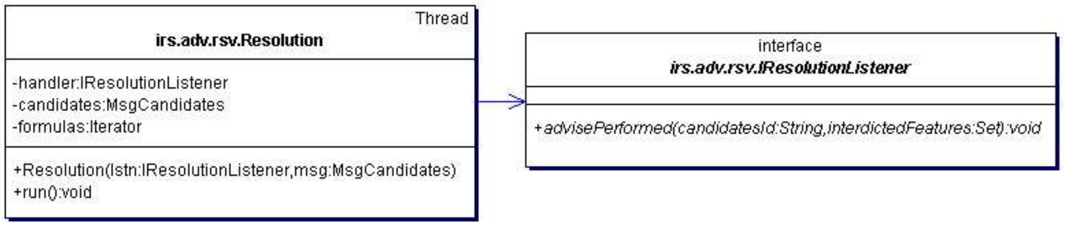

Thread
irs.adv.rsv.Resolution
-handler:IResolutionListener
-candidates:MsgCandidates
-formulas:Iterator
+Resolution(lstn:IResolutionListener,msg:MsgCandidates)
+run():void
interface
irs.adv.rsv.IResolutionListener
+advisePerformed(candidatesId:String,interdictedFeatures:Set):void

Figure 4. Class diagram of `Resolution` and `IResolutionListener`

The resolution process will cover all `Formula` instances against the `MsgCandidates` instance. The `validate` method succeeds when it verifies:

- ⇨ one correspondence between the message's candidate features and formula actions, and
- ⇨ the formula is satisfied.

In this case, the `validate` method returns a `Set` of interdicted or allowed actions, as a result of the formula verification.

The actions are constant identifiers, represented by a `String` in capital letters. The conditions are objects with a particular method, specified by the `IAdvCondition` interface, which evaluates the information contained on the `MsgCandidates` instance. The evaluation determine certain situations, like loop occurrence. The loop situation is defined by adv.Loop class, which implements the `IAdvCondition` interface.

The implementation of a condition encloses a large spectrum of situations. For instance, the grant permission between IR Advisers starts on the verification of a specific condition.

The adopted solution provides some flexibility to the introduction of new conditions.

3.2 Interpreter

The Interpreter loads the formulas contained in a file, into a collection of `Formula` instances, held in the Resolver. The (1) and (2) deontic logic formulas are expressed in a textual programming language, specified by the EBNF (Appel, 1998) syntax:

formula_list $\rightarrow$ formula$^+$
formula $\rightarrow$ actions && conditions '=>' status ';'
actions$\rightarrow$ ACTIONS id_list
conditions $\rightarrow$ CONDITIONS id_list
status $\rightarrow$ order id_list
order$\rightarrow$INTERDICTIONS | PERMISSIONS
id_list $\rightarrow$ '(' ID$^+$ ')'

For example, the formula presented in section 2.2, that forbids a forward to enter a loop, is expressed in our textual programming language by

actions(FORWARD) && conditions(adv.Loop) $\Rightarrow$ interdictions(FORWARD)

The meaning of the programming language constructs is equal to the meaning of equivalent (1) and (2) deontic formula constructs. Each interpreted formula results in a new `Formula` instance that is loaded into Resolver.

The implementation of the Interpreter lexical and syntactical analyzer was supported by the JLex and Cup tools (Appel, 1998).

3.3 Listener

Using Java RMI (Grosso, 2001), the implementation of the communication process is very simplified. However, that choice will restrict the integration between applications and the IR Adviser. For example, the Java RMI integration with any other technology different than Java will be unrealizable.

To avoid this scenario we choose an independent application communication solution supported on sockets (Tanenbaum, 2001). This option implies the specification of an explicit protocol for the exchanged messages. The protocol is specified in XML (Yergeau et al. 2004), because of its benefits on:

⇨ textual representation of object's structure and internal state;
⇨ serialization and desserialization, into and from Java objects, respectively;
⇨ tools are available for XML text analysis;
⇨ finally, XML is a standard well accepted in the scientific and industrial community.

The translation process of the XML messages into Java objects was implemented by JAXB-Java Architecture for XML Binding framework (McLaughlin, 2002).

The Listener receives messages from two different entities, applications and IR Advisers from different nodes. To handle connections from both entities the Listener will run two different threads, represented by two instances of `HandleConnections` class. For every established connection, the Listener notifies `HandleAppConnection` or `HandleAdvConnection` instance, which will create a new `AppHandler` or `AdvHandler` instance, respectively, depending if the connecter is an application or an adviser.

Once the connections are established, the communication between the Listener and the application, or another adviser, will be handled by `AppHandler` or `AdvHandler` instance, respectively.

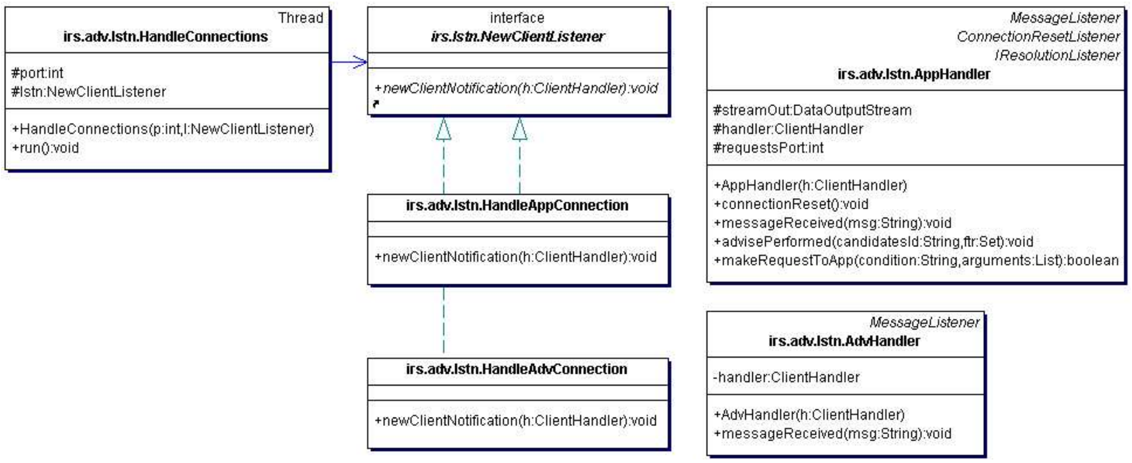
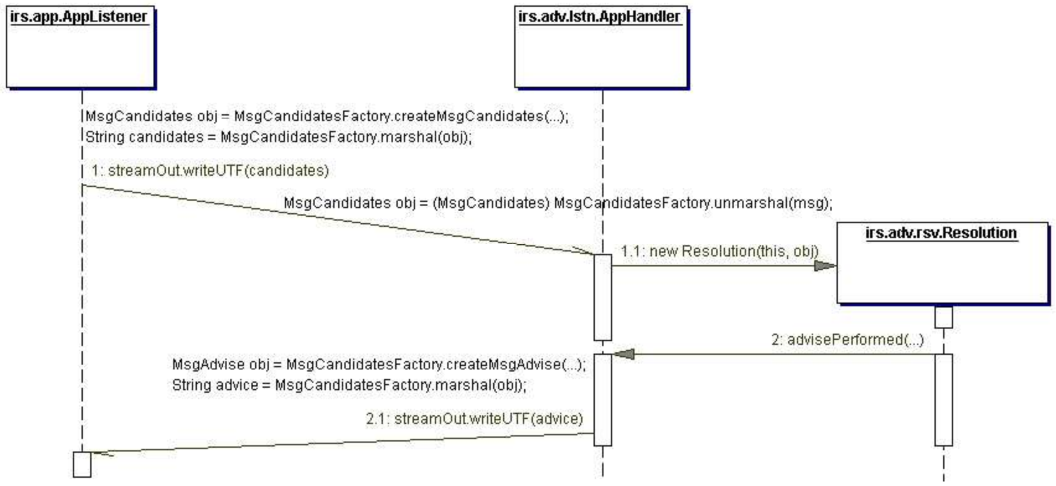

Thread
irs.adv.lstn.HandleConnections
#port:int
#lstn:NewClientListener
+HandleConnections(p:int,l:NewClientListener)
+run():void
interface
irs.lstn.NewClientListener
+newClientNotification(h:ClientHandler):void
irs.adv.lstn.HandleAppConnection
+newClientNotification(h:ClientHandler):void
irs.adv.lstn.HandleAdvConnection
+newClientNotification(h:ClientHandler):void
MessageListener
ConnectionResetListener
IResolutionListener
irs.adv.lstn.AppHandler
#streamOut:DataOutputStream
#handler:ClientHandler
#requestsPort:int
+AppHandler(h:ClientHandler)
+connectionReset():void
+messageReceived(msg:String):void
+advisePerformed(candidatesId:String,ftr:Set):void
+makeRequestToApp(condition:String,arguments:List):boolean
MessageListener
irs.adv.lstn.AdvHandler
-handler:ClientHandler
+AdvHandler(h:ClientHandler)
+messageReceived(msg:String):void

Figure 5. Class diagram of Listener classes

The applications messages are translated by the Listener into `MsgCandidates` instances, that will be sent to the Resolver. The Resolver returns to the application an advice. This process is asynchronous and the responses from the Resolver for the requests could be made on a different order. Figure 6 depicts the message sequence diagram of the overall process, where the Internet application is represented by an `AppListener` instance.

The description of the elements, entities, and relationships of the exchanged messages are specified by the `MsgCandidates.xsd` schema file, which is parsed by JAXB binding compiler. To make the translation process between XML messages into Java objects and vice versa easier, we provide an auxiliary factory class, `MsgCandidatesFactory`. We can see the use of these class services on the sequence diagram of figure 6.

Messages from other IR Advisers are posted by their Resolver's, which send a request to be evaluated by the Application of this node. This is the example of the grant permission for reserved access to Web pages. In this case the Listener will redirect the request to its Application and return back the response to the Resolver of the other IR Adviser.



```
MsgCandidates obj = MsgCandidatesFactory.createMsgCandidates(...);
String candidates = MsgCandidatesFactory.marshal(obj);
1: streamOut.writeUTF(candidates)
MsgCandidates obj = (MsgCandidates) MsgCandidatesFactory.unmarshal(msg);
1.1: new Resolution(this, obj)
2: advisePerformed(...)
MsgAdvise obj = MsgCandidatesFactory.createMsgAdvise(...);
String advice = MsgCandidatesFactory.marshal(obj);
2.1: streamOut.writeUTF(advice)
```

Figure 6. Messages sequence diagram between an application and the adviser

3.4 Internet Application

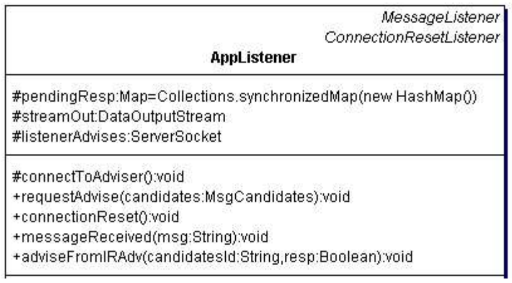
To be able to integrate an IR Adviser, an Internet application must satisfy the following requirements:

- ⇨ Be able to connect to the adviser listening port, to post the list of feature candidates. This port number is predefined in the adviser properties file;
- ⇨ Provide a port where the adviser will connect and return the advices. This port number should be sent by the application to the adviser, thought the previous socket connection;
- ⇨ Handle the connection of the adviser on the previous port;
- ⇨ Respect the XML format specified in `MsgCandidates.xsd` schema, related with all exchanged messages between the application and the adviser.

The establishment of the two socket connections is the first step taken by the application on its initialization. Then, the communication process is asynchronous and obeys to sequence diagram of figure 6.

The `AppListener` class is an implementation of the communication mechanisms for Java based applications. This class could be inherited to extend the functionality of its base methods:

- ⇨ `connectToAdviser` – performs the establishment of the two connections to adviser;
- ⇨ `requestAdvise` – sent a list of feature candidates to adviser, in a MsgCandidates instance;
- ⇨ `adviceFromIRAdv` – invoked for each received advice.



```
#pendingResp:Map=Collections.synchronizedMap(new HashMap())
#streamOut:DataOutputStream
#listenerAdvises:ServerSocket
```

```
#connectToAdviser():void
+requestAdvise(candidates:MsgCandidates):void
+connectionReset():void
+messageReceived(msg:String):void
+adviseFromIRAdv(candidatesId:String,resp:Boolean):void
```

Figure 8. `AppListener` class diagram

The application customization is responsible for the features execution tracking. Overriding the `adviseFromIRAdv` method makes possible to perform extra actions for the rejected features, such as notifying the Email postmaster.

Currently, we have implemented one IR advisor at PC/Linux under Together IDE. Initial testing focused Email features with JAMES-Java Entreprise Mail Server, from Apache Software Foundation. The experimentation works within a LAN, and we are testing cases where the IR advisor is installed only in part of the network.

4. CONCLUSION

When the intermediate nodes do not have an IR Adviser, as expected there is some degree of service degradation as result of the FIs. However, when the IR Adviser is available in all nodes that process the message, the preliminary results show an improvement of the system behaviour.

Prior to the public release of our system, we intend to study the advisor efficiency. We expect the performance will not decrease in larger extension because the resolution is linear and the number of features is expected to be lower than one hundred.

We intend to extend our experimentation with HTTP servers, taking advantage of our previous experience introducing our Java classes on the JAMES server architecture.

ACKNOWLEDGEMENT

We express our gratitude to Luigi Logrippo, Professor at the Université du Québec en Outaouais /Canada for many suggestions and improvements of our work. The work was implemented with support from the Instituto Superior de Engenharia de Lisboa and Technical University of Lisbon.

REFERENCES

Appel, A. W., 1998, *Modern Compiler Implementation in Java*. Cambridge University Press. UK.

Arnold, K. and Gosling. J., 2000, *The Java Programming Language*, 3rd edition. Addison-Wesley , Menlo Park, USA..

Blair, L., et al., 2002 A Feature Manager Approach to the Analysis of Component-Interactions, *Proceedings of 5th Int'l Conference on Formal Methods for Open Object-based Distributed Systems*, Enschede, The Netherlands, pp 233-248.

Bowen, T.F., et al., 1989. The Feature Interaction Problem in Telecommunication Systems, *Proceedings of 7th Int'l Conference on Software Engineering for Telecommunication System*s, pp 59-62.

Chentouf, A. et al, 2003. Experimenting with Feature Interaction Management in SIP Environment. *Telecommunication Systems*, Vol. 24 No. 2, pp 251-274.

Grosso,W., 2001, *Java RMI*. O'Reilly. Sebastopol, USA.

Hall, R.J., 2000, Feature Interactions in Electronic Mail, *Proceedings of 6th Int'l Workshop on Feature Interations in Telecommunication and Software Systems*, Glasgow, Scotland, pp 67-82.

Lennox, J. and Schulzrinne, H., 2000, Feature Interaction in Internet Telephony; *Proceedings of 6th Int'l Workshop on Feature Interactions in Telecommunication and Software Systems*, Glasgow, Scotland, pp 38-50.

McLaughlin, B., 2002, *Java & XML Data Binding*, O'Reilly. Sebastopol, USA.

MacCarty, B., 2003, *RedHat Linux Firewalls*, Addison-Wesley. Reading, USA.

Pang, J. and Blair, L., 2003, Separating Concerns from Distributed Feature Components; *Proceedings of Workshop on Software Composition*, Warsaw, Poland

Tanenbaum, A.S., 2001, *Modern Operating Systems* 2nd edition. Prentice-Hall, Englewood Cliffs, USA.

Weiss, M., 2003, Feature Interactions in Web Services, *Proceedings of 7th Int'l Workshop on Feature Interations in Telecommunication and Software Systems*, Ottawa, Canada, pp 149-156.

Yergeau, F. et al., 2004, *Extensible Markup Language (XML) 1.0* 3rd edition, W3C.