Feature interactions in adaptive multimedia applications: design-time detection and run-time negotiation

Ken Y. Chan[1], Gregor v. Bochmann[2]
Timeleap Inc. [1]
70A Brockington Crescent,
Nepean, Ontario, Canada. K1H 8A6
ken@timeleap.com [1]
School of Information Technology and Engineering (SITE) [2]
University of Ottawa
P.O.Box 450, Stn. A, Ottawa, Ontario, Canada. K1N 6N5
bochmann@site.uottawa.ca [2]

Abstract: This paper describes methods for reducing feature interactions among multimedia applications, which include but are not limited to mobile, telephony, and web applications, such that these applications become more adaptive to their environment. An application or a feature is modeled as an EFSM which is associated with a set of pre-conditions and post-conditions which we define as the signature of the feature. These conditions could have variables that could not always be assigned with values at design-time. These conditions typically represent user preferences which are available only at run or deployment time. Thus, these conditions must be checked at run-time for resolving feature interactions. A feature negotiation framework is proposed for this reason. To realize this framework, we would extend the caller preferences extension of IETF SIP protocol. Features which are expressed in the form of EFSMs are associated with their success scenarios. By combining features into different models, we could verify whether a model could satisfy their corresponding scenarios at design-time. Failures in verifying these scenarios result in feature interactions. As our case study of multimedia applications, a formal SDL model of SIP and its services has been derived from published SIP specifications for verification and validation. In addition, we would apply our feature negotiation framework to a personal multimedia web agent service. The scenarios of resolving its feature interactions using our negotiation framework would be described.

1. Introduction

Features in a telecommunication system are packages of incrementally added functionality that provide services to subscribers or administrators. In this paper, we do not make any distinction between feature and service; the two terms would be used interchangeably. These features, which are used by thousands, if not millions, of users, could interact with each other. Thus, the problem of feature interactions, which was stemmed in the telecommunication industry, remains a difficult problem to solve. There are many definitions of *feature interactions (FI's)* [1]. We consider FI's as undesirable side effects caused by interactions between features and/or their environment. By applying a combination of verification and validation and negotiation schemes, we believe we could reduce feature interactions among multimedia applications, which include but are not limited to mobile, telephony, and web applications, such that these applications become more adaptive to their environment.

Many people believe that Internet telephony would be the next major market for many carriers. Internet telephony is defined as the provisioning of telephone-like services over the Internet. With many Internet users having already embraced voice chat and webcam conferencing, Internet telephony, which also encompasses convergence of existing Internet and PSTN voice services, is the natural step forward. SIP, which is being standardized under IETF RFC 2543 [2], is designed based on the general philosophy of IETF that protocols remain open and support decentralization of control over applications in an Internet environment. SIP is generally better developed than other telephony signaling protocols in areas such as distributed call control between end-systems and inter-provider communications. These areas are known to be prone to feature interactions (FI's), thus many researchers believe feature interactions are more pronounced in Internet telephony than PSTN (Public Switched Telephone Network) [3].

Feature interaction is not a new problem and has been discussed in the research community for long time. Many solutions have been proposed to tackle the feature interaction problem in traditional telephony (the so-called Plain Old Telephone System, POTS) [4, 5, 1, 6, 7, 8, 9, 10]. Some of these techniques may be applied to IP Telephony, but further investigations are required to deal with the new feature interactions that appear in IP Telephony [3].

In general, research interests in feature interactions may be classified into the following areas [1]: (1) classification of feature interactions [11], (2) methods for detecting feature interactions, manually, semi-automatically, or automatically, at design-time [4, 9, 10] or run-time, (3) methods for avoiding feature interactions at design-time [7], and (4) methods for resolving interactions at run-time [5, 6]. Detecting and avoiding feature interactions at design-time and resolving feature interactions at run-time have been the most active areas of research [1]. Ideally, when a feature interaction is detected, the service designer would incorporate some schemes in the design to avoid the interaction. However, this is not always possible because the interaction may occur only when features are bound to dynamic data at run-time. Thus, resolving the interaction at run-time is the only solution though the process is still resource intensive.

In this paper we deal with design-time and run-time feature interaction resolution. As noted above, we consider FI's as undesirable side effects caused by interactions between features and/or their environment. First we propose a new classification of FI's based on the undesirable effect they produce. We consider the following cases: livelock, deadlock, unfairness, unexpected non-determinism and incoherence. Most of these effects are well-known problems in distributed systems design. We believe that this classification is useful because it suggests methods and tools, developed for distributed systems in general, for detecting and resolving the interactions. This is further discussed in Section 4.

This paper also deals with a case study of investigating feature interactions in the new SIP protocol for Internet telephony. The objective of this case study was to develop a formal specification in SDL of the basic functionality of SIP and many of its proposed features, and use the available commercial verification and validation tools to identify existing feature interactions or to show their absence. For this verification problem, we used a relatively standard, semi-automatic approach which is well-known in the context of formal description techniques [12]. Our approach included (a) automatic (partial) reachability analysis, and (b) validating manually created scenarios, written as Message Sequence Charts (MSC's), against the SIP model by automatically verifying that they are consistent with the SDL specification. We describe in Section 3 the structure of our SDL SIP specification, our approach to verification and the detection of feature interactions. In this context, the classification of feature interactions is useful because it guides our work for detection and resolution, as discussed for many examples of detected interactions.

We believe if a feature interaction is an undesirable side effect, then conflicting user requirements of different features could be the root cause of the interaction. Therefore, we detect interactions among features at design time by validating our model against the success scenarios which are written in the form of message sequence charts [13]. Some of these scenarios are derived from the call flow diagrams contained in [14]. However, we noticed that the SIP documentation does not include a precise service model. We have therefore developed an abstract user interface for Internet telephony describing the interactions between the users and the system.

Finally, we consider run-time feature interaction detection and resolution and propose a framework and protocol for this purpose. In order to detect the presence of an interaction between two features, it is important to specify the properties of the features involved in some precise manner. While in our case study mentioned above, we described the behavior of the SIP features in SDL, it appears that, for run-time interaction detection, a feature characterization at a higher level of abstraction may be more suitable. Gorse [4] proposed a feature characterization in first-order logic; in fact, he used Prolog for the purpose of automation. He characterized a feature by a set of predicates that represent preconditions, postconditions and triggers. However, his work was mainly aimed at design-time verification. Griffeth [6] proposed a run-time negotiation approach were a feature is characterized by a goal, and a goal may be implied by a more specialized goal, or by several composite subgoals. She proposes a protocol by which the system entities can exchange proposals and counterproposals about the goals to be attained. The run-time feature interaction and resolution framework proposed in Section 5 is based on this previous work. We extend the Gorse's feature characterization [4] which is coded and transmitted within the normal control messages of the application or in additional negotiation messages. We also propose a coding scheme for the feature characterization based on XML. Two example applications of our feature negotiation protocol are presented which demonstrate the feasibility of this approach. One of these applications is in the context of SIP, the other e-mail screening. In fact, this feature negotiation paradigm has some similarity with existing approaches to quality of service negotiation for adaptive multimedia applications.

This paper is structured as follows: Section 2 reviews existing classification schemes for feature interactions and introduces our new classification scheme based on the undesirable effect of the interaction. It also presents a few interactions of SIP features that have not been described before. Section 3 describes the case study of using a formal specification of SIP, written in SDL, for detecting interaction between the features in Internet telephony using commercial SDL and MSC tools for specification editing, verification and validation. It also contains a discussion of the detected feature interactions and their classification. Section 4 explains how the classification is useful for the resolution of feature interaction by giving many examples related to the SIP protocol. In Section 5, a framework and protocol for run-time feature detection and resolution (that is, negotiation) is presented with two examples of application in Internet telephony and e-mail filtering.

2. Different approaches to classifying feature interactions

2.1. Traditional approaches

A well-known approach to classifying feature interactions was presented in [11]. This approach foresees two orthogonal classifications: by nature and by cause. The classification by nature includes three dimensions: by kind of user, by the number of users, and by the number of network components. The possible combinations of these dimensions lead to five types of FI's:

- single-user-single-component (SUSC),
- single-user-multiple-component (SUMC),
- multi-user-single-component (MUSC),
- multi-user-multi-component (MUMC), and
- customer-system (CUSY).

For the classification by cause, [11] suggests the following general causes: violation of feature assumptions, limitations of network support, and intrinsic distributed system problems. Instead of discussing these general causes, we describe in the following four detailed causes that are also mentioned in [11] and are specializations of the above general causes:

- *Resource contention (RSC)* is definitely a well-known distributed system issue. It is defined as the attempt of two or more nodes to access the same resource. In the context of FI's, an accessing node may be the feature running on a network component. Resource is an abstract term; it needs not be a physical entity. For example, Call Waiting (CW) and Three Way Calling (TWC) in POTS may be considered as contending for the flash hook signal simultaneously.
- *Violation of Feature Assumptions (VFA)* relates to the assumptions that are the basis for the design and operation of a telecommunication feature, similarly as for any software feature. There are many types of assumptions; one example that is particularly worth examining is assumptions about data availability [11]. Features such as Terminating Call Screening (TCS) cannot function properly without the caller-id of the caller. If the caller-id were made unavailable in the case of Private Call (PC), TCS would permit the call request. In the case of Operating Assisted Call (OAC) and OCS, the hiding of the original caller-id by OAC allows the call to bypass the OCS restriction.
- *Resource Limitation (RSL)* is definitely a common cause of FI's in POTS, because most of the traditional POTS end-user devices (e.g. basic phones) have limited user interfaces and computational power. If the classical example of resource limitation were revised (e.g. Call waiting and Three Way Call), the confusion of the flash hook signal can also be attributed to the lack of separate buttons for the two features; thus that feature interaction can be also classified under resource limitation.
- *Timing and Race conditions (TRC)* are common, particularly in distributed real-time systems like POTS running on PSTN. A race condition is defined as a condition where two or more nodes have non-mutually exclusive read and modify access to a shared resource. As a result, the value or status of the shared resource may be undefined (different values in the same context) depending on the timing of the accesses between the nodes. A classical example is between Automatic Callback (AC) and Automatic Recall (AR). The AR feature makes a call on behalf of the original caller when the callee becomes idle again. Both AC and AR features depend on the busy signal of the callee. The timing of the busy signal received by the two features would either allow one of the calls to go through on the single line

of the POTS phone, or reject both call requests because both ends are calling each other simultaneously.

We note that Lennox [3] introduced two feature interaction categories for IP telephony services called cooperative and adversarial interactions. They are incidentally similar to, but not the same as, the concepts called cooperative and interfering interactions defined by Gorse [4]. We will discuss below how they are related to the new classification categories presented in Section 2.2.

2.2. Classifying feature interactions by symptom

We note that the traditional approaches of classifying feature interactions by nature and by cause may be useful, but these classifications do not lead explicitly to a general scheme for detecting or resolving the interactions. A classification process (taxonomy) for FI's is most useful if it leads to methods of detecting, preventing, or resolving FI's. While understanding the causes of interactions could give us some ideas on how to detect feature interactions, the causes of interactions are too vague for formulating precise design rules to prevent interactions, and policies to resolve them; for example, a resource contention scenario like presenting a voice greeting and call waiting tone to the same user may or may not be considered as a FI. Also how does a service designer know which "resource" to declare for checking resource contention? Thus, the causes cannot be construed as the most efficient means to derive methods for detecting FI's. We propose another classification of FI's, which is *by symptom or effect*. This new categorization enables FI's to be associated with some of the well-known distributed system properties. As a result, we believe existing verification techniques and tools for detecting pertinent distributed system properties may be used to facilitate detection of FI's..

This classification focuses on the effects or symptoms of the problems which can be used to narrow down the search for detecting feature interactions. Common distributed system properties such as livelock, deadlock, fairness, and non-determinism have a well-defined meaning, and tools can detect some of these properties for a given system. The category called incoherent interactions is also introduced in the following subsection.

2.2.1. Livelocking (LLCK) interactions

Livelock is a well-defined concept that occurs when the affected processes enter state transition loops and make no progress. In the case of Automatic Callback (AC) and Auto Recall (AR), if both features receive the busy signal from the callee and initiate the call simultaneously, both calls would not go through on single-line phones because both ends are busy making calls. Both features are in a livelock situation because they may always get the busy tone when they repeat the process and never complete the call. A livelocking interaction needs not be permanent; in most cases, the relative timing of the two processes leads to an eventual exit from the livelock loop.

2.2.2. Deadlocking (DLCK) interactions

Deadlock is another well-known distributed system concept that occurs when two or more processes are in a blocked state because they require exclusive access to a shared resource that belongs to the other, or wait for a message from the other process that will never be sent. An example is described in Figure 1 involving CW and CFB at A communicating with CW and CFB at B.

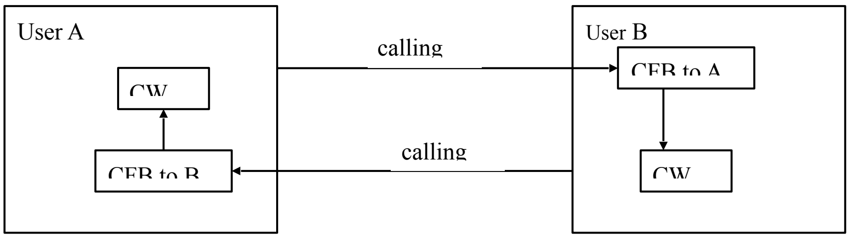

Figure 1: Call waiting and CFB at A versus Call waiting and CFB at B (deadlock)

If A and B call each other simultaneously and are programmed to forward to each other on busy, then both callers would keep hearing the phone ringing at the other end (and perhaps also hear the CW tone). They would be deadlocked at the ringing state and no progress would be made until one of them hangs up. If both ends do not subscribe to CW service, the call would be forwarded to each other on busy (or a call loop). Both users would be presented with a busy tone instead.

2.2.3. Incoherent interactions (ICOH)

An incoherent interaction is a form of a violation of feature assumptions (or properties). This term was first introduced in [4] to describe "the identification of specific incoherence properties" between the affected features. Furthermore, an incoherent interaction can be caused by resource contention and resource limitation. The two properties from the classical case of CFB and OCS is a good example: user A would successfully call user C via CFB even though an OCS entry at A is supposed to forbid any outgoing call to C. The CFB and OCS are programmed with contradictory assumptions.

2.2.4. Unfair interactions (UFR)

Fairness is also a well-known distributed system concept in which processes of equal priority should be assigned fair scheduling so that they eventually proceed and have fair access to shared resources. A novel case of unfair feature interaction occurs between Pickup (CP) and Auto Answer (AA, also known as Call Forward to Voicemail). If one of the CP destinations has subscribed to AA that would unconditionally forward any incoming calls to the voicemail, the call would always be answered by the destination with AA first.

2.2.5. Unexpected non-determinism (NDET)

This category is introduced here because it is an observable feature interaction. It occurs when a feature with non-deterministic behavior triggers other features that have normally deterministic behaviors, but due to this triggering behave in a non-deterministic fashion. The users are usually confused by the behavior of the affected features because they do not expect such non-deterministic behavior, which is usually caused by timing and race conditions among the features. An example of this type of feature interaction is between Automatic Call Distribution (ACD) and Call Pickup (CP) (see Figure 2). The ACD feature allows incoming calls to the subscriber be redirected to one of the pre-programmed destinations (e.g. destination A or B). The redirecting policy can be a random selection or a deterministic algorithm. The CP feature allows the subscriber of the service to inform a list of destinations (destination A and B) that an incoming call has been put on hold, and is ready to be picked up by one of the destinations. It is conceivable that the switching element Y sends a call pickup message to all destinations and then another switching element called X with ACD would redirect the call to a particular destination (e.g. B). When destination A decides to pick up the call, the incoming call is no longer available because the call has already been redirected to destination B. In summary, the non-deterministic call redirecting policy of ACD affects the first-come-first-served call pickup policy of CP.

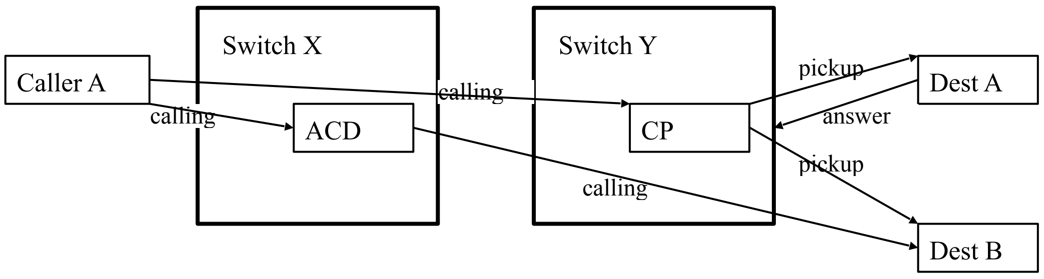

Figure 2: ACD and CP unexpected non-deterministic interaction

2.3. Further examples of feature interactions

We mentioned earlier that Lennox [3] proposed the classification of cooperative and adversarial interactions. Cooperative FI's are multi-component interactions (SUMC or MUMC) where all components share a common goal, but have a different and uncoordinated way of achieving it [3]. They correspond to livelock, deadlock, unfairness, and unexpected non-deterministic multi-component interactions. The fact that features contend for a single resource or for each other's resources in order to establish call connections, resulting in violations of safeness and liveness properties, is a form of cooperative interaction. Let us examine the example of *Request Forking (RF) and Auto-answer (or voicemail)* [3] which is a potential cooperative FI in SIP. Request forking is unique to SIP; it allows a SIP proxy server P to attempt to locate a user by forwarding a request to multiple destinations in parallel. The first destination to accept the request will be connected, and the call attempts to the others will be cancelled. The Auto-Answer (AA) or Call Forward to Voice Mail is designed to accept all incoming calls while the user is unavailable. Both features are intended to ensure that the user does not miss the call. However, since AA would answer the call request almost immediately, RF would always forward the call request to the same AA destination. Therefore, the user who may be closer to the other destinations would never be able to answer the call, and hence the intention of RF is ignored. To resolve this violation of fairness, one may introduce a delay timer to AA (e.g. answer after 4 rings) such that all destinations would be given a fair chance to answer the call.

Lennox' adversarial FI's, by contrast, are multi-component interactions (SUMC, MUMC) where all components disagree about something to be done with the call (e.g. violation of feature assumptions) [3]. They are difficult to detect and correspond to the category of incoherent interactions in our classification. The following interactions, specific to SIP, belong to this category and have not been mentioned in the literature.

Timed ACD and Timed Terminating Call Screening (TCS) is a potential adversarial FI. With IP telephony, service designers can rapidly create innovative features; for example, time or calendar-based forwarding and screening features. In this example, TCS restricts incoming calls that are originating from certain callers at the destination. ACD may be programmed to distribute different kinds of incoming calls to different sets of destinations depending on the time of the day (e.g. calls from A or B to destinations S or T in the daytime, and calls from C or D to destinations X or Y in the nighttime). However, destinations S and T are programmed to screen calls from A or B in daytime and X and Y to screen calls from C and D at nighttime. Although ACD is intended to select the best route for incoming calls, TCS at various destinations is programmed to screen calls from these destinations at given time periods. As a result, their policies contradict with each other and no calls between these callers and callees can ever be completed in the conflicting time period. When time is involved in incoherent interactions, the solution to such conflicting policies is not trivial.

Call Screening and Register is another form of multi-component interaction unique to SIP and involves dynamic address changing in SIP. A user agent or proxy may add, delete, or modify the current contact address of a user stored in the registrar. The address can be changed at any time by sending a "Register" request message to the registrar. Since a user cannot possibly stay current with the latest address change of another user, the call screening features would be rendered useless. It is a form of incoherent interaction. This interaction involves a debate of privacy for the call screener and the caller. These security issues are beyond the scope of this paper.

Dynamic Addressing and User Mobility and Anonymity is another adversarial FI that centers on the controversial issue on the balance between the lack of address scarcity in the Internet and the correct programming of features that depend on reliable addresses. As it becomes the norm that a user owns more than one intelligent wireless and/or wired personal communication device, user mobility is a serious issue in designing reliable services. Furthermore, anonymous email accounts and email spamming are big problems in the Internet. Could anonymous accounts and telemarketing spam calls emerge in the SIP world when SIP becomes popular? If SIP addresses were just as easy to obtain as email addresses, the call screening and forwarding feature in SIP would need to be more sophisticated than in POTS.

3. Design-time detection of feature interactions for Internet telephony

In this section deals with a case study [15, 16] where our objective was the detection of feature interactions in the context of Internet telephony, based on the Session Initiation Protocol (SIP) proposed by the IETF. For this purpose, we developed a formal model, in SDL, of the basic functionality of the SIP protocol [2] as well as of several additional features proposed in [14]. In the following, we first give an overview of SIP and then we describe the SDL model of SIP that we developed. We also describe how the model was checked for consistency with a number of interaction scenarios that were described in the form of Message Sequence Charts, and how feature interactions can be detected based on this model.

3.1. Overview of SIP

SIP (Session Initiation Protocol) is an application-layer multimedia signaling protocol standardized under IETF RFC 2543 [2]. Similar to most World Wide Web protocols, SIP has an ASCII-based syntax that closely resembles HTTP. This protocol can establish, modify, and terminate multimedia sessions that include multimedia conferences, Internet telephony calls, and similar applications. There are additional IETF drafts that describe other important extensions to SIP in efforts to realize VoIP deployment. In particular, examples of telephony features are described in [14].

In SIP terminology, a call consists of all participants in a conference invited by a common source. A SIP call is identified by a globally unique call-id. Thus, for example, if several people invite a user to the same multicast session, each of these invitations will be a separate call. However, after the multipoint call has been established, the logical connection between two participants is a call leg, which is identified by the combination of "Call-ID", "To", and "From" header fields. The sender of a request message or the receiver of a response message is known as the client, whereas the receiver of a request message or the sender of a response message is known as the server. A user agent (UA) is a logical entity which contains both a user agent client and user agent server, and acts on behalf of an end-user for the duration of the call. A proxy is an intermediary system that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy may process or interpret requests internally or by passing them on, possibly after translation, to other servers.

What makes SIP interesting and different from other VoIP protocols are the message headers and body. Like HTTP, a SIP message, whether it is a request, response, or acknowledgement, consists of a header and a body. The Request-URI names the current destination of the request. It generally has the same value as the "To" header field, but may be different if the caller has a more direct path to the callee through the "Contact" field. The "From" and "To" header fields indicate the respective registration address of the caller and of the callee. The "Via" header fields are optional and indicate the path that the request has traveled so far. Only a proxy may append or remove its address as a "Via" header value to a request message. The "Record-Route" request and response header fields are optional fields and are added to a request by any proxy that insists on being in the path of subsequent requests for the same call leg. It contains a globally reachable Request-URI that identifies the proxy server. "Call-Id" represents a globally unique identifier for the current call session. The Command Sequence ("CSeq") consists of a unique transaction id and the associated request method. It allows user agents and proxies to trace the request and the corresponding response messages associated to the transaction. The body of a SIP request is usually a SDP header [17], which contains the detailed description of the multimedia streams of the session.

The first line of a response message is the status line that includes the response string, code, and the version number. It is important to note that the "Via" header fields are removed from the response message by the corresponding proxies on the return path. When the calling user agent client receives this success response, it would send an "ACK" message back to the callee.

The message header fields that we have included in our SDL model are: "Request-URI", "Method", "Response Code", "From", "To", "Contact", "Via", "Record-Reroute", "Call-Id", and "CSeq". We believe these fields are most important to FI's because they convey the state of all the participants in the session. There are many header fields available in SIP and can become very complex, as documented in [2].

3.2. Modeling SIP services and protocols in SDL and MSC

In this section we describe how we modeled SIP services and their FI's. The Specification and Design Language (SDL) [18] was chosen as the modeling language for the following reasons: (1) the SIP protocol and services are state-oriented and map well to the communicating extended finite state machine model of SDL, (2) SDL is well supported by commercial software tool vendors like Telelogic [19] whose tools are used by many telecommunication software developers, and (3) various verification and validation techniques are available in SDL tools. Our approach to model SIP and its services starts with defining the use case and test scenarios using Message Sequence Charts (MSC) [13]. Next, we will describe the structural and behavior definitions of the SDL model. Then we will discuss the verification and validation process. We have described the basic SIP protocol and many additional services like CFB, CW, and OCS in SDL [20], but only selected diagrams will be presented in this paper.

3.2.1. An abstract user interface

Communication services are defined in terms of abstract interactions with the user (or the system components using the service in question). This is analogue to the definition of system requirements in terms of use cases, that are sequences (sometimes called scenarios) of interactions with the users. We note that the IETF SIP standard does not include a proper definition of the "SIP service", concentrating only on the SIP messages exchanged between the different system components. For example, an INVITE message in SIP serves as more than just a call setup message; it may be used for putting the call on hold in the middle of a call (mid-call features) [14].

We have therefore developed a set of abstract user interactions for Internet telephony. This set of interactions represents an "Abstract User interface" between the user and the local IP-telephony equipment. The modeling of the user-observable behavior at these interfaces is essential for describing FI's. The user interactions include such signals as Offhook, Onhook, Dial, SelectLine, Flash (flash hook), CancelKey, RingTone, AlertTone, TransferKey which relate to the actions that are available on most telephone units on the market.

3.2.2. Use case scenarios and test cases

Since the IETF drafts have provided a call flow diagram or success scenario for each sample service in graphical notation [14], we translate these scenarios into message sequence charts. However, we note that these call flow diagrams only include the sequence of exchanged SIP messages at the protocol level. They do not represent service scenarios in the sense of use cases. Following standard practice of software engineering, we think that it

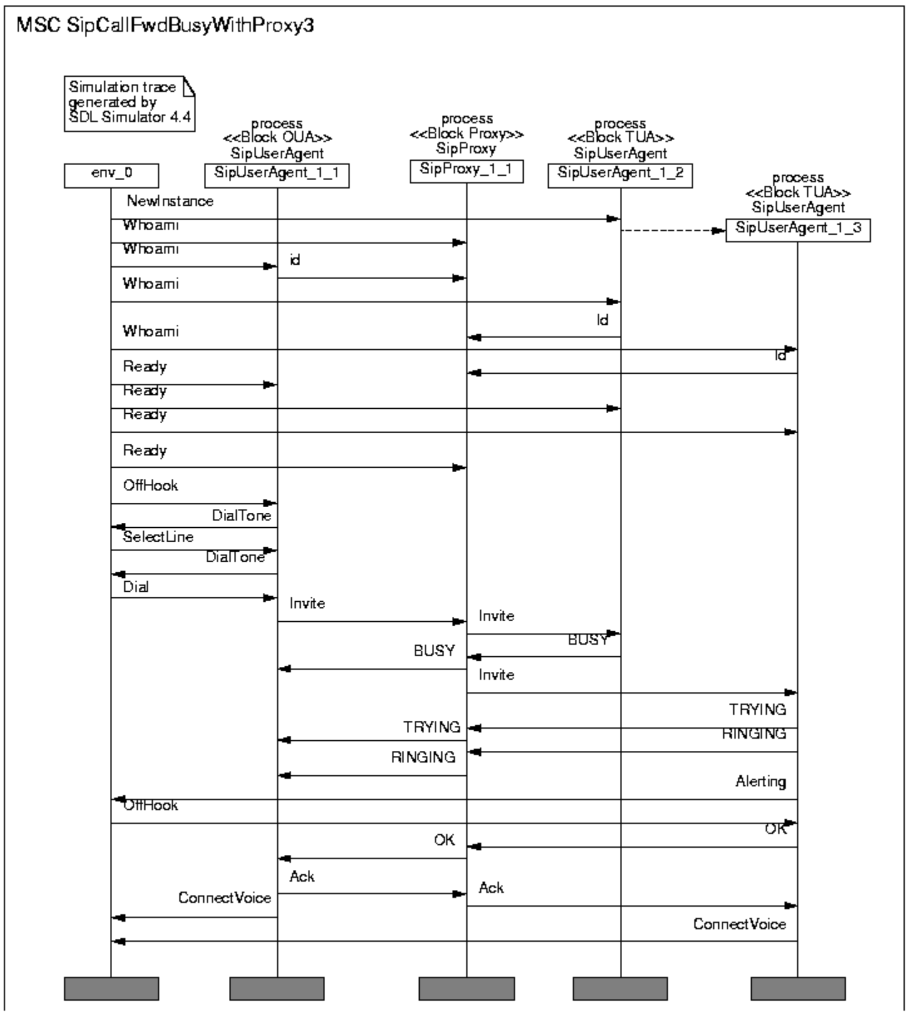

Figure 3: Call Forward Busy "Service and Protocol" Scenario

is important to define service usage scenarios at the interface between the user and the system that provides the communication service. We have therefore defined an abstract user interface which represents the interactions at the service level. These interactions between the users and the SIP system describe use case scenarios of SIP services. The users are the actors of the use case scenarios and are represented by the environment "env_0" in SDL.

The sequence chart shown in Figure 3 represents the Call Forward Busy "service and protocol scenario". The sequence chart is the combination of the use case scenario with the corresponding scenario of exchanged SIP messages from [14]. It is written with response codes as message names and without message parameters so that the chart can fit in this paper. The complete MSC can be used as a test case for validating the SDL specification of the SIP protocol, as explained in the next subsection.

As a matter of fact, we have written one or more message sequence chart (service and protocol scenario) as test cases for each service and have used the Telelogic tool to verify our SDL model can realize these scenarios of message exchanges.

3.2.3. Definition of the system structure

A system specification in SDL is divided into two parts: the system and its environment. An SDL specification is a formal model that defines the relevant properties of an existing system in the real world. Everything outside the system belongs to the environment. The SDL specification defines how the system reacts to events in the environment that are communicated by messages, called signals, sent to the system. The behavior of a system is the joint behavior of all the process instances contained in the system, which communicate with each other and the environment via signals. The process instances exist in parallel with equal rights. A process instance may create other processes, but once created, these new process instances have equal rights. SDL process instances are extended finite-state machines (FSMs). An extended finite-state machine (EFSM) is based on an FSM with the addition of variables, which represent additional state information, and signal parameters. The union of the FSM-state and additional state variables represent the complete state space of the process.

The relationship between modeled entities, their interfaces, and attributes are considered parts of the structural definition. A SDL system represents static interactions between SIP entities. The channels connected between various block instances specify the signals or SIP messages that are sent between user agents and/or proxies. Block and process types are used to represent SIP entity types such as user agent (SipUserAgentType) and proxy (SipProxyType).

A SIP User Agent contains both, what is called, in SIP, a user agent client (UAC) and a user agent server (UAS). Since a user agent can only behave as either a UAC or UAS in a SIP transaction, the user agent is best represented by the inheritance of UAC and UAS interfaces. The inheritance relationship is modeled using separate gates (C2Sgate and S2Cgate) to partition the user agent process and block into two sections: client and server. The "Envgate" gate manages the sending and receiving of "Abstract User" signals between the user agent and the environment (see Figure 5). An instantiation of a block type represents an instance of a SIP entity such as user agent or proxy, and contains a process instance that describes the actual behavior of the entity. The process definition contains the description of all the state transitions of the features to which the SIP entity has subscribed. In addition, each SIP entity must have a set of permanent and temporary variables for its operations. In the case of a user agent, the permanent variables store the current call processing state values of the call session. The temporary variables store the values of the consumed messages for further processing.

Similar to a user agent, a proxy consists of client and server portion. It tunnels messages between user agents but also intercepts incoming messages, injects new messages, or modifies forwarding messages on behalf of the user agent(s). A proxy is also a favorite entity in which features are deployed. A SIP Proxy has one gate that interacts with the environment (Envgate), and four gates that interact with user agents or proxies: client-to-proxy (C2Pgate), proxy-to-client (P2Cgate), server-to-proxy (S2Pgate), and proxy-to-server (P2Sgate) (see Figure 5).

In our SDL model, we use different SDL systems to represent different structural bindings between SIP entities and to simulate a particular set of call scenarios. The most complex system in our telephony model (see Figure 4) realizes the concept of originating and terminating user endpoints. It contains an originating user agent block, a proxy block, and a terminating user agent block. The originating block contains all the user agent process instances that originate SIP requests while the terminating block contains all the user agent process instances that receive these requests. Upon receiving a request, a terminating user agent would reply with the corresponding response messages. It is important to note that only the originating user agent and proxy instances can send SIP requests (including acknowledgements).

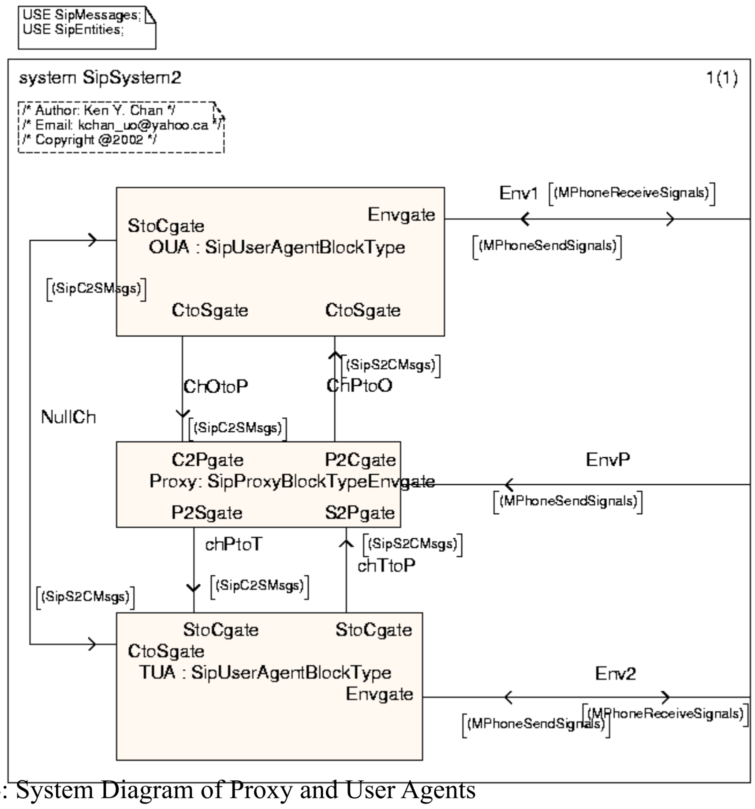

Figure 4: System Diagram of Proxy and User Agents

All blocks are initialized with one process instance. During the simulation, a 'NewInstance' "Abstract User" signal can be sent to a process instance to create a new process instance. Before the first "INVITE" message is sent, the environment must initialize each user agent and proxy instance with a unique Internet address by sending them a message called 'Whoami'. The user agent instances would in turn send an 'Id' message along with its Internet address and process id (PId) to the proxy. Thus, the proxy can establish a routing table for routing signals to the appropriate destinations during simulation. Similar to the user agent, a proxy has a set of permanent and temporary variables for its operations.

SIP messages are defined as SDL signals in the SIPMessage package. The key header fields in a SIP message are represented by the corresponding signal parameters. Since there is no linked list data structure in SDL, the number of variable fields such as 'Via' and 'Contact' must be fixed in our model. Complex data and array types of SDL have been tried for this purpose, but were dropped from the model because they may cause the Tau validation engine to crash. Instead, we used extra parameters to simulate these variable fields.

3.2.4. Behavior Specification

In general, a SIP feature or service is represented by a set of interactions between users and the user agent processes, and possibly the proxy processes. Each process instance plays a role in a feature instance. A process instance contains state transitions which represent the behaviors that the process instance plays in a feature instance. In our model, we capture the behavior of a SIP entity as an SDL process type (e.g. UserAgentType). The first feature we model is the basic SIP signalling functionality, also known as the basic service. Each process type has state transitions that describe the basic service. In the case of a user agent, the process includes the UAC and UAS behavior. We define a feature role as the behavior of a SIP entity that makes up a feature in a distributed system. A feature role is invoked or triggered by trigger events. The entry states of a feature role in a SIP entity are the states for which the trigger events are defined as inputs. For example, the on-hold feature can be triggered in the 'Connect_Completed' (entry) state by an INVITE message with 'ONHOLD' as one of its parameters. After the invite message is sent, a response timer is immediately set. Then, the user agent is waiting for the on-hold request to be accepted. When the other user agent responds with an OK message, the requesting agent would reset the response timer and inform the device to display the on-hold signal on the screen.

In general, trigger events are expressed as incoming signals; pre-condition, post-conditions, constraints are expressed as enabling conditions or decision. Actions are SDL tasks, procedure calls, or output signals. As a triggering event being consumed by the user agent process, the parameters of the event may be examined along with the pre-conditions of the feature. Then, actions such as sending out a message and modifying the internal variables may be executed. Post-conditions and constraints on the action may also be checked. Finally, the process progresses to the next state.

A state transition occurs when (1) an "Abstract User" signal is received from the environment, (2) a request or response message is received, or (3) a continuous signal is enabled. The so-called Continuous Signal in SDL is used to model the situation in which a transition is initiated when a certain condition is fulfilled. For example if the UAS is busy, the boolean 'isBusy' would be true in the 'Server_Ring' state. The UAS would immediately send the BUSY response to the caller. This way, we would not have to worry about the timer expiration because we do not need to send a busy toggling signal to simulate busy during a simulation. An asterisk '*' can be used in a state and signal symbol to denote any state or any signal; its semantics is equivalent to a wildcard. A state symbol with a dash '-' means the current state. Error handling, such as response timer expiration, can easily be modeled with SDL timers and a combination of '*' and '-' state symbols. For example, when the response timer expires, a timeout message would be automatically sent to the input queue of the user agent process. The expiration of the response message is generalized as the situation in which the receiving end does not answer the request in time. Thus, a 'NoAnswer' signal is sent to the environment. Finally, the process can either return to the previous state or go directly to the idle state.

Moreover, we can add additional features or services such as CFB, OCS, and other services, to the system. To add behaviors of additional features to a process type, we can subtype a "basic" process type such as UserAgentType. The derived type has the same interfaces and additional state transitions. We do not want to add new interfaces (signal routes) to the process types because we do not want to change the interfaces of the block types. If a feature requires new SIP methods and response codes, we would not need to change the interfaces because method names and response codes are simply signals parameters in our model. Thus, we avoid the need to add new interfaces whenever a new feature is defined.

3.2.5. Verifying the SDL Model against MSC's

The SDL specification that has been discussed in the previous subsection was constructed using the Telelogic Tau tool version 4.3 and 4.4 [19]. The Tau tool offers many verification or reachability analysis features: bit-state, exhaustive, random walk bit state exploration and verification of MSC. Bit-state exploration is particularly useful and efficient [21] in checking for deadlocking interactions because it checks for various reactive system properties without constructing the whole state space like the exhaustive approach does.

We verified our SDL model of SIP mainly by checking whether the model would be able to realize specific interaction scenarios which were described in the form of Message Sequence Charts (MSCs). In fact, we used the scenarios described informally in [2, 14], and rewrote them in the form of MSCs. Then we used the Tau tool to check that our SDL model was able to generate the given MSC. An MSC is verified if there exists an execution path in the SDL model such that the scenario described by the MSC can be satisfied. This is similar to existential quantification saying that there exists an execution sequence as described by the MSC. When "Verify MSC" is selected, tool may report three types of results: verification of MSC, violation of MSC, and deadlock. Unless 100% state space coverage is selected, partial state space coverage is generally performed and reported in most cases. Verification of an MSC is achieved by using the MSC to guide the simulation of the SDL model. As a result, the state space of the verification has become manageable [22].

3.3. Detecting Feature Interactions based on the SDL model

One of our objectives was to explore the feasibility of using Tau to detect known and new FI's. Although SIP is fundamentally different from traditional PSTN signaling protocols, most of the traditional POTS FI's still exist in SIP, if the SIP services are designed and implemented with the mindset of POTS. (Note: Alternate design approaches to SIP services are discussed in the next section). We do not believe it is either feasible or practical to come up with an automated feature interaction detection scheme using the Tau environment because the tool has limited validation features and the Validator frequently crashes. Instead, we write test cases in the form of either MSC or Observer Process Assertions, a notation provided by the Tau tool, to verify whether traditional FI's that were known to exist in POTS, still exist in the context of SIP [11, 4].

In the previous sections, many FI's were identified as violations of distributed system properties: deadlock (a form of violation of safety) and livelock (a form of violation of liveness). Tau offers various automated reachability analysis tools and reports any deadlocks found during the analysis. Let us revisit the previous deadlock example of CFB and CW (see Section 2.2.2). If we have a SDL system with the corresponding SDL blocks or SIP entities which have user agent processes running with CFB and CW behaviors, we can select the "bit state exploration" option to detect any deadlock. Tau's Validator does not seem to offer reports on the liveness of the analyzed system in any of its reachability analysis functions. Thus, we use Tau's Observer Process features to detect live-locking FI's, as explained in Section 3.3.2.

Furthermore, incoherent interactions are not easy to check with the Tau tool because certain interactions that involve contradictory properties cannot be expressed in a straightforward manner. There are two ways to approach this problem: (1) Use an MSC to specify incoherent properties, or (2) use an observer process offered by Tau to specify assertions. By examining the validation report (which shows the simulation trace in the from of an MSC), we could trace back to the locations where the problem occurs; thus we could identify the interacting features. Both approaches have their advantages and disadvantages. However, the observer approach is the only practical approach in the current Tau release.

3.3.1 To specify incoherencies as MS's

Many service designers like to use MSCs to capture important scenarios of a feature. If the sets of message sequence charts describing two features can be compared/verified against each other, then certain obvious incoherent FI's may be detected in this informal requirement specification stage. However, capturing key behaviors of a feature in MSC is not always possible. For example, the success scenario of OCS cannot be expressed directly as an MSC in the case of OCS and CFB interaction. How can we express in an MSC that user A cannot call user C? The concept of something that can "never happen" is usually expressed using universal quantification. Since verification of MSCs in Tau is based on an existential quantification of the scenario, a property that something should "never" happen can be expressed by negation. If m is a scenario that should never happen, then we could use the Tau tool to check whether the SDL model can satisfy this MSC m. If the result is that m cannot be satisfied by the model, this verifies the property.

If this concept is applied to OCS, "verifying user A can call user C" being false is equivalent to the truth of "user A can never call user C". Thus, the successful verification of the success call scenario from user A to user C concludes the violation of the OCS specification. Since features like OCS apply to all calls including mid-call, the pre-enable condition of the MSC must be specified. The pre-enable condition must include all possible situations in which a call can be made to user C. There are very many possibilities; clearly, detecting incoherent interactions using MSCs is almost impossible.

However, an extension to MSCs, called Live Sequence Chart (LSC) [23], seems to address the above concerns. First of all, an LSC allows the designer to specify messages that cannot be sent in a scenario. This is useful for specifying OCS type of services. Secondly, an LSC can be specified with either the universal or existential quantification of a scenario, which offers great flexibility in verifying scenarios that are contradictory to each other. Last but not least, a universal LSC allows an associated pre-enable condition which defines the scope of the scenario. This is essential to modeling services because such pre-enable conditions can be used as the triggering condition of a feature. Unfortunately, current SDL tools do not support LSCs, but we believe LSCs are promising in this context.

2. To specify incoherencies as assertions in an Observer Process

Observer Processes with powerful access to signals and variables of various processes are provided by Tau, they are useful for feature interaction detection. One or more observer processes can be included in an SDL system to observe the internal state of other processes during validation [19]. When the validation engine is invoked to perform state exploration, the observer processes remain idle until all the observed processes have made their transitions. Then, each observer process would make one transition. All observer processes have access to the internal states of all the observed processes via the Access operators. Therefore, the conditions (e.g. the continuous signals, decisions and enabling condition) in which the observer process makes a transition may be considered for defining assertions for the observed processes. If the observer process finds that the assertion is violated, it will generate a report in the middle of the validation. The validation process would be stopped immediately and the report would be presented to the user.

Furthermore, we have experimented with assertions that verify certain liveness properties of the system. For example, if we have an integer counter for each user agent to keep track of the number of times each user agent has been in 'Ringing' state, we can monitor whether a user agent has been looping at 'Ringing' state or not. More specifically, an assertion, which verifies user agent UA1 and UA2 are not simultaneously in 'Ringing State' for more than three times ensures a certain level of liveness in the system.

In general, we use our intuition to come up with useful assertions for each feature interaction category. Although this is not an automated process, we believe it is a practical approach to a subjective problem such as detecting "undesirable side-effects" between features. We were able to verify whether the well known FI's still exist in SIP or not. Thus, we could apply preventive measures to make SIP services more robust, as explained in the next section.

4. Preventing Feature Interactions

After a feature interaction is detected, the next natural step is to change the design to prevent it. In this section, we discuss corrective measures that can be incorporated in the design and implementation of a system to prevent FI's. These corrective measures map directly to the causes and effects of FI's. Preventing FI's may be done at design-time or at run-time. Run-time prevention is more difficult to exercise because the features may be running on different nodes and it is not easy to coordinate the actions that should be performed when a feature interaction is detected at run-time. Prevention strategies for different feature interaction categories are presented in the following subsections. In addition, the feature interaction examples described in Section 2 will be revisited again. The goal of this discussion is to provide a catalog of FI's to service designers, such that a designer can associate a feature with a set of prevention schemes that would reduce potential interactions with any unknown features. We will also see that it is easier to prevent FI's in SIP than in POTS, because SIP has extra call information (e.g. 'Via', 'Record-Route', 'Contact') in the message headers.

4.1 Preventing resource contention and limitation (example: CW and TWC)

The natural prevention strategy for resource limitation is to increase the number of available resources. IP Telephony services, particularly SIP, should face fewer resource limitation problems than POTS because SIP end-user devices tend to be more powerful:

fast processor, a lot of memory, and flexible user interfaces and displays. For example, the semantic ambiguity of flash hook at the POTS user interface, which is the cause of both resource contention and resource limitation in the case of CW and TWC in the POTS service, should never happen with SIP phones. SIP phones should be able to display the choice of two actions, namely putting the current call on hold and switching to the incoming call, or conferencing in the third party. The phone may assign one or more soft buttons for this purpose. However, if interactive user intervention is not possible, priority schemes (e.g. fuzzy policies [5]) may be used to resolve resource contention. Clearly, these strategies can be applied in both design-time and run-time feature interaction prevention.

4.2 Preventing incoherent interactions (example: OCS and CFB)

Incoherent interactions can either be direct or transitive [4]. Direct incoherencies are present when two features are associated with the same trigger signal but lead to different or contradictory results. A transitive incoherent interaction occurs when one feature triggers another feature, and the latter has results that are contradictory to the former feature. The word 'transitive' refers to the transitivity with respect to features that may lead to loops. Gorse has suggested a scheme to detect this type of interaction [4]. However, no prevention scheme has been presented. Since incoherencies lead to contradictory results, allowing only one of the features to be triggered would prevent the incoherence. However, transitive incoherencies such as OCS and CFB are difficult to prevent at run-time because the interaction may involve indirect triggering of a remote feature. How could we know that the incoming call has triggered CFB running on server Y which contradicts the assumptions of the OCS feature running on server X? Obviously, if the triggering condition of the first feature were made available to the other feature and vice versa, the two features could take advantage of the extra information and negotiate a settlement. We will discuss an example of resolving this type of interactions in Section 7.

4.3 Preventing unexpected non-determinism and unfair interactions (examples: ACD & CP, and CP & AA)

Unexpected non-deterministic interactions must be caused by at least one feature that has some non-deterministic behavior. The non-deterministic action of one feature typically triggers the other feature. Therefore, if we were to remove the triggering dependency of the second feature on the first feature, the problem would be solved. In the case of ACD and CP, an incoming call may trigger the non-deterministic invitation of ACD and the pickup invitation of CP to the same or different destination(s). If concurrent triggering of ACD and CP was detected and only one feature was activated, the problem would be solved.

An unfair interaction represents the violation of a fairness property. An unfair interaction can be avoided by relative priorities between the two features or by introducing randomization in the selection process.

4.4 Preventing deadlocking & livelocking interactions (examples: CW & CFB, and ACB & AR)

Since a deadlock between two features is caused by a mutual dependency on each other's resources, removing the cyclic dependency would typically resolve this problem. The dependency in the case of CW and CFB is on the Busy signal. The busy signal is intercepted by CW, therefore CFB at both ends would continuously ring each other. Loop detection can prevent the problem. Loop detection and prevention is a mandatory feature of the core SIP protocol. Thus, CFB features can examine the {From, To, Contact, Via, Record-Route} headers to determine whether a loop would occur or not. If the answer is yes, the subsequent forward would not be allowed and a response message with a response code of LOOP_DETECTED would be returned.

Livelock interactions, like deadlock interactions, also imply cyclic dependency on each other's resources. However, the result is that the system would fail to proceed. To break the lock or the loop, a randomized timer can be introduced to the triggering of the affected features. For example, ACB and AR may be stuck in a livelock if ACB and AR would initiate the callback and the recall simultaneously on single-line phones. If the triggering of one of these features was delayed, one of the calls would go through. However, this livelocking interaction is unlikely to happen because SIP phones usually have more than one line. Instead, ACB and AR would result in an incoherent interaction because both users would be presented with two calls between the same endpoints. Since two different SIP user agents at both ends handle the calls, loop detection in a user agent would normally not detect this interaction at run-time. However, the user agents that are running in the same terminal may be designed such that they would share their route information with all their local user agents. As a result, this incoherent interaction could be prevented.

5. Run-time feature negotiation

5.1. A feature negotiation framework

Since it is very difficult to detect all possible feature interactions at design time, and it is difficult to foresee new features that may be developed in the future, it appears to be useful to consider run-time methods for avoiding feature interactions. In fact, the process of determining a set of non-conflicting features that correspond to the desire of the users has much similarity with quality of service negotiation for adaptive multimedia applications. In such situations, one has to take into account constraints that come from various system components, administrative policies and user preferences in order to select system features that are technically possible and acceptable to all users involved.

As an example, let us consider incoherent interactions. They can be either direct or transitive [4]. Direct incoherencies are present when two features are associated with the same trigger signal but lead to different or contradictory results. A transitive incoherent interaction occurs when one feature triggers another feature, and the latter has results that are contradictory to the former feature. The word 'transitive' refers to the transitivity with respect to features that may lead to loops. Transitive incoherencies, such as OCS and CFB, are difficult to prevent at run-time because the interaction may involve indirect triggering of a remote feature. How could we know that the incoming call has triggered CFB running on proxy Y which contradicts the assumptions of the OCS feature running on proxy X ? - Obviously, if the triggering condition of the first feature were made available to the other feature and vice versa, the two proxies could take advantage of the extra information and negotiate a settlement.

In the case that the Observer Process (see Section 3.2.2) is used for detecting feature interaction at design-time, all this information would be available to this fictitious observer. For design-time detection, we propose in the following a protocol for exchanging the same information at run-time between the real components of the system.

We therefore propose a run-time feature negotiation framework based on the following principles:

(a) Each feature has a certain number of characteristics which describe assumptions and effects of executing the feature.

(b) When a feature is invoked by a component of the distributed system for a particular user, the characteristics of that feature will be communicated to the other system components through appropriate messages. This information may be appended to control messages that are exchanged anyway, or additional messages may be introduced just for the purpose of exchanging this information.

(c) Each system component may analyze the information about the activated features in the different parts of the system and, when some conflict is detected based on this information, may decide to change the local feature activation and/or exchange messages with the other components to negotiate changes to the features to be used within the given session.

We understand that not all system components are interested in allocating computational resources to resolve feature interactions. Therefore, we propose the exchanged information about active features to be made optional, as well as the analysis of the information received.

In Section 2, we have shown that most feature interactions are related to reactive system properties (e.g. livelock, deadlock, fairness, non-determinism), or incoherence between conflicting user/feature goals. Gorse [4] showed that feature goals can be expressed as first order predicates in the context of automatically filtering feature incoherences. In fact, features have been described [4, 6] as a predicate comprising pre-conditions, triggering events, and post-conditions (i.e. results). We add to this information the identification of the system component where the feature is invoked (in the case of SIP, these are the user agents and proxies), and the identification of the users for whom these features were invoked. This is similar to the negotiating agent approach described in [6]. In summary, we propose the following information to be exchanged for each feature that is proposed to be invoked:

- Name of the feature
- Idenfication of the responsible user
- Identification of system component invoking the feature
- possibly some of the following:
    - trigger event that has lead to the invocation of the feature,
    - preconditions that must be satisfied when the feature is triggered,
    - postcondition that must be satisfied when the feature has completed its task,
    - invariant that must be satisfied throughout the execution of the feature,
    - Proposal of various options for resolving interactions.

As an example, we consider the Originating Call Screening feature and assume that the user "user1@example.com" has instructed his proxy SIP server that outgoing calls to the users "user2@example.com" and "user3@example.com" should not be allowed. If a connect request from user "user1@example.com" goes through the proxy, the latter will forward the following information to the downstream proxies and callee user agent:

- Originating Call Screening feature
- Responsible user: "user1@example.com"
- Responsible system component: <proxy of "user1@example.com">
- Invariant: no connection between "user1@example.com" and "user2@example.com"
- Invariant: no connection between "user1@example.com" and "user3@example.com"

5.2. Feature negotiation protocol and an example

We consider in the following the example of a feature negotiation protocol for Internet telephony that can be used as a complement to SIP. In this work, we have been inspired by the SIP extension draft on caller preferences [24] which allows a caller to express preferences about request handling in servers. The caller preferences are sent as extensions of the SIP header in the SIP INVITE and Response messages. The caller preferences draft leaves much room for the types of values that can be put in the header fields; for example, one can place quality of service parameters like delay, bandwidth, media type capabilities, or request methods in the header. The caller preferences draft also took the SDP Offer/ Answer approach [25] into account for the negotiation of the multimedia session. In this context, one participant offers the other a description of the desired session from their perspective, and the other participant answers with the desired session from their perspective.

However, for the purpose of resolving feature interactions, the information fields for caller preferences are not suitable. Therefore we propose in the following a new protocol for SIP feature interaction negotiation. Instead of adopting the message syntax proposed in [24], we use XML as the coding scheme because of its flexibility for supporting application-dependent data structures that can be defined by appropriate document type definitions (DTD) or schemas. In the case of Internet telephony, the resulting XML messages can be appended to the bodies of the SIP INVITE, Response and Ack messages, as well as to additional INFO messages that may be sent with the sole purpose of feature negotiation. In the case of other application areas, these messages may be appended to the appropriate control messages of those application protocols.

For the example of the Originating Call Screening feature considered above, we could have the following XML message:

```
<feature    name = "OCS"   subscriber = "sip:user1@example.com"    component = " proxy.example.com">
    <predicate  name ="connected"    required-value = "false"    scope = "invariant" >
         <param   name = "caller"   value = "sip:user1@example.com" />
         <param   name = "callee"   value = "sip:user2@example.com" />
      </predicate>
    <predicate   name ="connected"   required-value = "false"    scope = "invariant" >
         <param   name = "caller"   value = "sip:user1@example.com" />
         <param   name = "callee"   value = "sip:user3@example.com" />
      </predicate>
    <alternative name=”cancelCall” required-value=”true” priority=”1”>
         <param   name = "caller"   value = "sip:user1@example.com" />
         <param   name = "callee"   value = "sip:user3@example.com" />
      </alternative>
</feature>
```

The meaning of this message is as follows: The first line indicates the *name* of the feature and the system *component* where it is invoked. Then there are two invariant assertions in the form of two *predicate* elements: the *name* defines a Boolean function, in this case the function "connected" (which corresponds to "call(u,v)" in [6]), the *scope* and *required-*

value indicate that this function must be false during the entire time that this feature is invoked. Finally, the *alternative* element represents a standard proposal for resolving a possible feature interaction by proposing the cancel the call from the *caller* to the *callee*. A *priority* is also indicate for this choice. This value may be considered during feature negotiation.

Similarly, the information representing the invocation of the Automatic Recall feature for the same caller and callee at the same proxy could be represented as:

```
<feature    name = "AR"    subscriber= "sip:user1@example.com"    component = " proxy1.example.com">
    <predicate name ="ConReq" required-value = "true"   scope = "trigger">
                <param   name = "caller"   value = "sip:user1@example.com" />
                <param   name = "callee"   value = "sip:user2@example.com" />
          </predicate>
    <predicate name ="busy" required-value = "true"   scope = "precondition">
                <param   name = "callee"   value = "sip:user2@example.com" />
          </predicate>
    <predicate name ="connected" required-value = "true"   scope = "postcondition">
               <param   name = "caller"   value = "sip:user1@example.com" />
               <param   name = "callee"   value = "sip:user2@example.com" />
         </predicate>
</feature>
```

The information representing the invocation of Call Forward on Busy for a call from caller "user1@example.com" to the callee "user2@example.com" to the callee "user3@example.com" could be written as:

```
<feature    name = "CFB"    subscriber = "sip:user2@example.com"    component = "proxy2.example.com">
     <predicate name ="ConReq" required-value = "true"   scope = "trigger">
                <param   name = "caller"   value = "sip:user1@example.com" />
                <param   name = "callee"   value = "sip:user2@example.com" />
          </predicate>
    <predicate name ="busy" required-value = "true"   scope = "trigger">
                <param   name = "callee"   value = "sip:user2@example.com" />
          </predicate>
    <predicate name ="connected" required-value = "true"   scope = "postcondition">
               <param   name = "caller"   value = "sip:user1@example.com" />
               <param   name = "callee"   value = "sip:user3@example.com" />
          </predicate>
    <alternative name="busyTone" required-value="true" priority="3">
               <param   name = "caller"   value = "sip:user1@example.com" />
         </alternative>
</feature>
```

Figure 5 shows a SIP message exchange involving a call from User A to User B which is forwarded to User C. Let us assume that the call was forwarded because of a Call Forward on Busy feature invoked on Proxy X where the user preferences of User B are registered. According to the proposed feature negotiation protocol, the last XML message above, represented in the figure as "CFB(A,B,C)", should be included in the INVITE messages sent by Proxy X to UserAgent C. This means that the receiving User C can be informed that the call was forwarded. The same information will also be included in the response message which may then may inform the caller that the call was forwarded and for what reason.

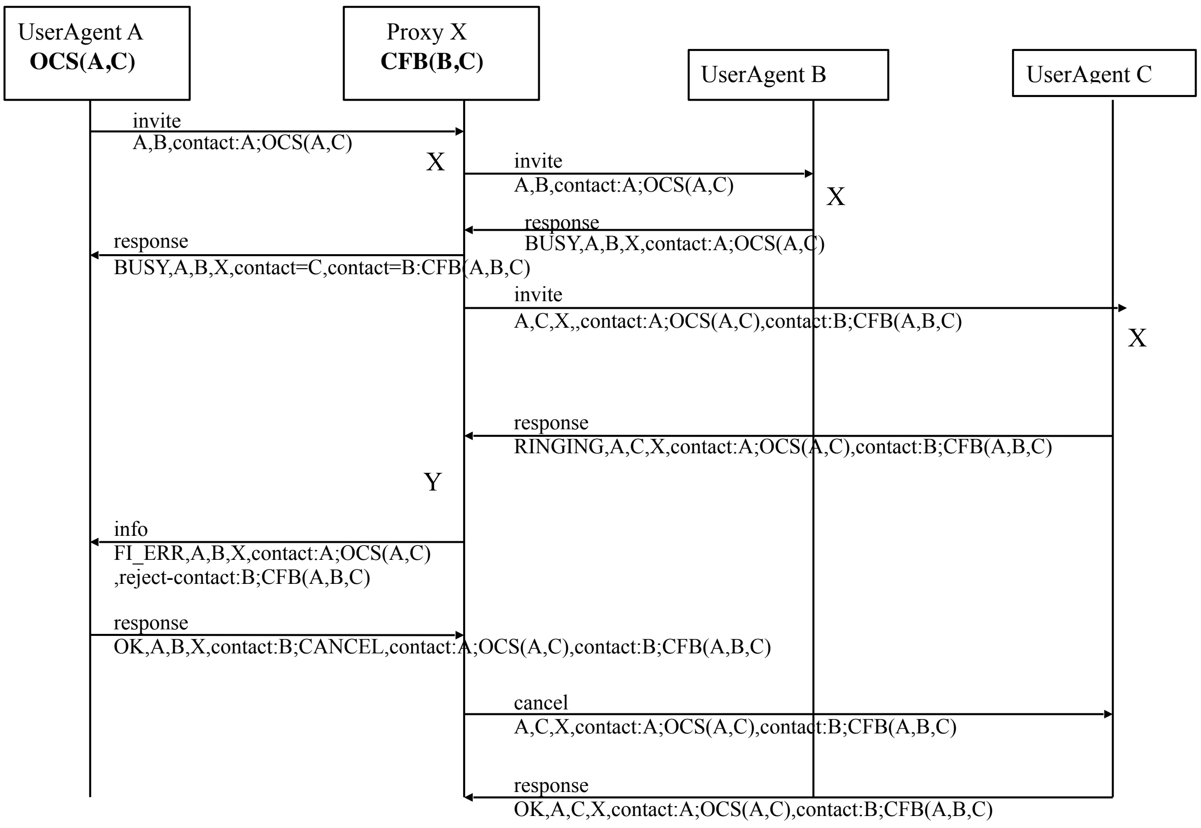
UserAgent A
OCS(A,C)
Proxy X
CFB(B,C)
UserAgent B
UserAgent C
invite
A,B,contact:A;OCS(A,C)
X
invite
A,B,contact:A;OCS(A,C)
X
response
BUSY,A,B,X,contact:A;OCS(A,C)
response
BUSY,A,B,X,contact=C,contact=B:CFB(A,B,C)
invite
A,C,X,,contact:A;OCS(A,C),contact:B;CFB(A,B,C)
X
response
RINGING,A,C,X,contact:A;OCS(A,C),contact:B;CFB(A,B,C)
Y
info
FI_ERR,A,B,X,contact:A;OCS(A,C)
,reject-contact:B;CFB(A,B,C)
response
OK,A,B,X,contact:B;CANCEL,contact:A;OCS(A,C),contact:B;CFB(A,B,C)
cancel
A,C,X,contact:A;OCS(A,C),contact:B;CFB(A,B,C)
response
OK,A,C,X,contact:A;OCS(A,C),contact:B;CFB(A,B,C)

Figure 5: Message Sequence Chart showing OCS and CFB feature interaction detection

Let us additionally assume that User A has subscribed to Originating Call Screening for Users A and C, and that this subscription is handled by the UserAgent A. Then the user agent could include the first XML message above, represented in the figure as "OCS(A,C)", in the initial INVITE messages. This information will then be forwarded by Proxy X to the other parties involved, as shown in the figure. This means that the call screening information would become known to the Proxy X and Users B and C. This is not necessarily desirable from the point of view of privacy protection; therefore Proxy X may decide not to include this information in the outgoing SIP messages.

The "X" shown in Figure 5 indicate the time instant when each of the system components obtains the information from both of the XML messages. At this point, the component may analyze these two messages and determine whether there are any feature interactions. In the case we have considered here, there is a contradiction concerning the value of the "connected" predicate which is supposed to be "false" according to the OCS feature and "true" according to the CFB feature. If the OCS feature information is not communicated to the other users, then this contradiction can only be detected at Proxy X or by the user agent of User A. Otherwise, the contradiction could also be detected by the other parties. The scenario shown in Figure 5 assumes that this feature interaction is detected by Proxy X, however not at the earliest possibility (indicated by an "X"), but at a later point (indicated by a "Y"). At this point, Proxy X sends an INFO message to the component handling the other feature (that is, UserAgent A), proposing to reject the call forwarding. This proposal is accepted by the UserAgent in the response and the Proxy X cancels the call to User C.

We note that a different scenario could be considered here where the UserAgent A gives a different response to the INFO message indicating the feature interaction. If for some reason the forwarding of the call to User C is the right thing to do (this may depend on respective priorities for call screening and forwarding which may depend on the particular destination user, or the user agent may consult with the user), the Agent may indicate in its response that call should be forwarded in this case. In this case the INVITE to User C will not be cancelled and the final accepting response from UserAgent C will be forwarded to UserAgent A.

It is important to note that for dealing with run-time feature interactions, as described above, each system component has to have an implementation only of those features that it supports itself, not of features only supported by other components. However, for detecting feature interactions, and for dealing with the associated negotiations, each component has to understand the information messages concerning the features implemented by the other components. The high-level characterization of features through predicates, as suggested in [4,6] and in our feature negotiation framework, has the advantage that a few predicates may be sufficient to characterize a larger number of features. For example, the two predicates "connected" and "busy" considered in our examples are sufficient to characterize the three features OCS, AR and CFB. It could therefore be expected that some new feature to be defined some time in the future may be characterized only by already known predicates. In this case, some older existing system implementations not supporting this new feature may still be able to detect feature interactions with this new feature and participate in a meaningful negotiation with the new component implementations supporting this new feature.

We believe that feature characterization message, such as those presented above, provide enough information for the session participants (user agents and proxies in the context of Internet telephony) to detect feature interactions at run-time and to engage in a meaningful negotiation for resolving the problem encountered. If a system component needs more information about a feature implemented in another component, it may possibly consult the registry for the complete feature description, for instance, SIP telephony features may be described as CPL scripts [27] which are typically stored in proxies. However, comparing scripts during a session invitation is generally too computationally intensive, thus impractical for our consideration.

5.3. Other examples: e-mail filtering and quality of service negotiation

One of the emerging Internet applications are implemented as personal agents. The concept of agent has been adopted in many e-commerce sites, such as Amazon.com. They developed these shopping agents for their portal where shoppers can specify their shopping preferences and the agents would look for the best-match items for the shoppers. We believe that the personal agent concept can be further extended; for instance, they may act like the shopper's secretary. They could crawl through the most popular Internet shopping web sites and look for a good match or bargain merchandise according to the shopper's priorities. For example, user Alice specifies that she would like to buy a Sony headphone for the price of $x dollars or lower. She shall be notified via e-mail including a photo of the suggested article. Then she would confirm the purchase with her personal agent.

Let us assume that Alice is on a trip and her e-mail is forwarded to a foreign message transfer agent (MTA), and that this MTA filters the incoming mail for spam and during high-load conditions, may also filter messages of larger size. If at a given time, the load of this MTA is high and the size of the message from the personal agent to Alice is very large (because it includes a large photo), the personal agent may receive negative response to the sending of the message including the following characterization of the mail filtering feature which was invoked at the overloaded MTA in response to the arrival of Alice's message:

```
<feature   name = "mail-discarded"   subscriber = "admin@example.com"   component =
"mta.example.com">
     <predicate name ="message-size" maximum-exceeded = "100000"  scope = "trigger"/>
    <alternative name=”try-later”  suggested-delay = "2h" />
</feature>
```

We note that the *name* "mail-discarded" of the feature implies that the message was dropped, there is no alternative to be negotiated for this instance of message transmission. The *predicate* element indicates that this feature was triggered because the *message-size* exceeded the maximum value which was set to 100000 octets. The *alternative* element indicates the "try-later" alternative, which means that the same feature my possibly not be invoked if the same e-mail message will be sent again at a later time (because the overload condition of the MTA may have disappeared).

When the personal agent receives this message it has therefore the choice of either immediately sending the message without the photo, or trying to send the complete message at a later time.

We may notice from this example that our feature negotiation framework is also easily applicable to quality of service negotiation for distributed multimedia applications. In this context, each system component, such as the user's workstation, the server or the network, may impose some constraints on the quality of service parameters, as for instance the video resolution, frame rate or bandwidth. For instance, a constraint on the bandwidth of 200Kbps could we expressed as a predicate of the form

```
      <predicate   name ="bandwidth"  maximum-value = "200 kbps"  scope = "invariant"/> .
```

The issues of quality of service management for mobile applications is discussed in [28], and the possibilities of scheduling multimedia sessions in the future, if currently the server or network are overloaded, is discussed in [29]. If the video server is overloaded, a new request for viewing a given video may be refused including, however, a proposal for viewing the document at some future time. Such a proposal may look like the following:

```
      <OR>
         <alternative name=”try-later”  exact-delay = "20 min" >
                  <predicate   name ="resolution"  required-value = "600x500"  scope = "invariant"/>
                  <predicate   name ="frame-rate"  required-value = "30"  scope = "invariant"/>
               </alternative>
         <alternative name=”try-later”  min-delay = "10 min" >
                  <predicate   name ="resolution"  required-value = "300x250"  scope = "invariant"/>
                  <predicate   name ="frame-rate"  required-value = "15"  scope = "invariant"/>
               </alternative>
      </OR>
```

6. Conclusions

We have introduced a new classification scheme for feature interactions based on the effect (or symptom) of the interactions. These effects are the undesirable results of the interactions, such as livelock or deadlock, contradicting objectives (incoherence), unfairness or unexpected non-determinism. This categorization gives hints about methods for detecting such feature interactions and for avoiding them through a redesign of the features involved. In Section 4, we have provided a detailed discussion of strategies that can be used for avoiding the different classes of feature interactions.

Since the considered symptoms of interactions are well-known concepts in the area of distributed system design, it appears to be reasonable to use, for the detection of feature interactions at design time, the common methods and tools that have been developed for the validation and verification of distributed system designs. We describe in Section 3 a case study where the SIP protocol for IP telephony was modeled in the formal description technique SDL in order to identify interactions between the various call handling features that are defined within this context. The contribution of this case study, besides an encompassing SDL specification of SIP and many additional features which is available on the Web [26], is a discussion of the experience with the use of the Telelogic Tau tool for the detection of feature interactions. We used not only the standard method of checking questionable behavior scenarios (in the form of Message Sequence Charts, MSC's) against the SIP specification in SDL, but also used the Observer facility provided by the Tau tool to detect inconsistencies during the simulation of the SIP specification.

Finally, we propose a framework and a protocol for feature interaction detection and negotiation during run-time. This framework is based on the premise that each party provides information to the other parties within the distributed system about the features that are locally invoked. Given that information, each of the parties within the system may analyze the information obtained from various sources and engage a negotiation with the other parties if inconsistencies between the invoked features are detected. Several examples are given to demonstrate the usefulness of this approach.

While the specification of SIP in SDL is an important step in making the description of this protocol more precise, thus providing a formal basis for the investigation of feature interactions among the SIP features currently defined, the detection of feature interactions for protocols of this complexity is an arduous task, even with the SDL tools available to date. In fact, most of the feature interactions we detected are closely related to well-known features in the context of traditional telephony.

We believe that the run-time detection of features, as proposed in this paper, is an promising approach. However, we note that each application domain needs to come up with information structures that represent the features involved in that application area. In the context of IP telephony, we have given three examples of information structures representing three SIP features. Further work is needed to consolidate these information structures and demonstrate that they contain enough information to allow the detection of the typical feature interactions that can be expected in IP telephony. Further work is also needed in order to elaborate more sophisticated negotiation algorithms that go beyond the automatic cancellation of one of the features.

Acknowledgements
We would like to thank our colleagues Luigi Logrippo (Université du Québec en Outaouais, Gatineau, Québec) and Daniel Amyot (University of Ottawa) for many discussions providing insight into the problems of feature interactions. We also would like to thank Telelogic for providing us with the Tau tool and for their support for using it. This

work was partially supported by funds from Communications and Information Technology Ontario (CITO) and the Natural Science and Engineering Research Council (NSERC), Canada.

References

[1] M. Calder, M. Kolberg, E. Magill, and S.Reiff-Marganiec, “Feature Interaction: A Critical Review and Considered Forecast”, Feature Interaction: A Critical Review and Considered Forecast. M. Calder, M. Kolberg, E. H. Magill and S. Reiff-Marganiec. Computer Networks, vol 41/1 pp 115-141, January 2003. Elsevier Science. http://www.dcs.gla.ac.uk/~muffy/papers/calder-kolberg-magill-reiff.pdf, accessed on February 06, 2002.

[2] M. Handley, H. Shulzrinne, E. Schooler, and J. Rosenberg, “SIP: Session Initiation Protocol”, Request For Comments (Proposed Standard) 2543, Internet Engineering Task Force, Mar. 1999.

[3] J. Lennox, and H. Schulzrinne, “Feature Interaction in Internet Telephony”, Sixth Feature Interaction Workshop, IOS Press, May. 2000.

[4] N. Gorse, “The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage” (Master Thesis), University of Ottawa, Canada, 2001.

[5] M. Amer, A. Karmouch, T. Gray, and S. Mankovskii, “Feature-Interaction Resolution Using Fuzzy Policies”, Feature Interactions in Telecommunications and Software Systems VI, pp. 94-111, IOS Press, 2000.

[6] N.D. Griffeth and H. Velthuijsen, “The Negotiating Agents Approach to Runtime Feature Interaction Resolution”, in: Feature Interactions in Telecommunications Systems, L.G. Bouma and H. Velthuijsen (eds.), IOS Press, Amsterdam, pp. 217-235, 1994.

[7] Pansy Au, and Joanne Atlee, “Evaluation of a State-Based Model of Feature Interactions”, Fourth International Workshop on Feature Interactions in Telecommunication and Software Systems, pp. 153-167, June 1997.

[8] L. Blair, and J. Pang, “Feature Interactions – Life Beyond Traditional Telephony”, Feature Interactions in Telecommunications and Software Systems VI, IOS Press, pp.83-93, 2000.

[9] Amy P. Felty and Kedar S. Namjoshi, “Feature Specification and Automatic Conflict Detection”, In M. Calder and E. Magill, editors, Feature Interactions in Telecommunications and Software Systems VI, IOS Press, May 2000

[10] J. Kamoun and L. Logrippo, “Goal-Oriented Feature Interaction Detection in the Intelligent Network Model”, School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, Canada, 1998.

[11] E.J.Cameron, N.D.Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Shure, and H. Velthuijsen, “A feature interaction benchmark for IN and beyond,” Feature Interactions in Telecommunications Systems, IOS Press, pp. 1-23, 1994.

[12] Courtiat, J.-P., Dembinski, P., Holzmann, G.J., Logrippo, L., Rudin, H. and Zave, P. (1996) “Formal methods after 15 years: Status and trends — A paper based on contributions of the panelists at the FORmal TEchnique ’95 Conference, Montreal, October 1995”. In: Computer Networks and ISDN Systems, 28, Elsevier Science B.V., 1845-1855.

[13] International Telecommunication Union, “ITU-T Recommendation Z.120: Message Sequence Chart (MSC)”, ITU-T, Geneva, Switzerland, 1999.

[14] A. Johnston, R. Sparks, C. Cunningham, S. Donovan, and K. Summers, “SIP Service Examples”, Internet Draft, Internet Engineering Task Force, June 2001, Work in progress.

[15] K. Y. Chan and G. v. Bochmann, Modeling the IETF Session Initiation Protocol and its services in SDL, Proc. of SDL Forum 2003, Springer Verlag (LNCS).

[16] K. Y. Chan and G. v. Bochmann, Methods for designing SIP services in SDL with fewer feature Interactions, Proc. Intern. Workshop on Feature Interactions in Telecommunications and Software Systems, Ottawa, Canada, June, 2003, IOS Press.

[17] M. Handley, and v. Jacobson, “SDP: Session Description Protocol”, Request For Comments (Proposed Standard) 2327, Internet Engineering Task Force, April 1998.

[18] International Telecommunication Union, “ITU-TS Recommendation Z.100: Specification and Description Language (SDL)”, ITU-TS, Geneva, Switzerland, 1999.

[19] Telelogic Inc., “Telelogic Tau SDL & TTCN Suite”, version 4.3 and 4.4, http://www.telelogic.com

[20] K. Chan, Methods for designing Internet telephony services with fewer feature interactions, Master Thesis, SITE, Univesity of Ottawa, June 2003.

[21] Holzmann, G.J. (1988) ‘An improved protocol reachability analysis technique’, Software Practice and Experience, Vol. 18, No. 2, pp. 137-161.

[22] O. Hargen, “MSC Methodology”, http://www.informatics.sintef.no/projects/sisu/sluttrapp/publicen.htm, accessed on Dec. 23, 2002, SISU, DES 94, Oslo, Norway, 1994.

[23] W. Damm, and D. Harel, “LSCs: Breathing Life into Message Sequence Charts*”, Formal Methods in System Design, Kluwer Academic Publishers, pp. 19,45-80, 2001.

[24] J. Rosenberg, H, Schulzrinne, and P.Kyzivat, “Caller Preferences and Callee Capabilities for the Session Initiation Protocol (SIP)”, draft-ietf-sip-callerprefs-08.txt, Internet Draft, Internet Engineering Task Force, Expires Aug 2003, Work in progress.
[25] J. Rosenberg, and H, Schulzrinne, “An Offer/Answer Model with the Session Description Protocol (SDP)”, Request For Comments (Standards Track) 3264, Internet Engineering Task Force, June 2002.
[26] http://www.site.uottawa.ca/~kchan/
[27] J. Lennox, and H. Schulzrinne, "Call processing language framework and requirements," RFC 2824, Internet Engineering Task Force, May 2000.
[28] K. El-Khatib, N. Hadibi and G. v. Bochmann, Support for personal and service mobility in ubiquitous computing environments, Accepted to EuroPar conference, Klagenfurt, Aug. 2003.
[29] A. Hafid, G. v. Bochmann and R. Dssouli, Quality of service negotiation with present and future reservations: A detailed study, Computer Networks and ISDN Systems, volume 30, issue 8, 1998, pp. 777-794.