INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Composing and Personalizing Next-Generation Telecommunication Services While Managing Feature Interactions

Alessandro De Marco

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montréal, Québec, Canada

April 2003

ABSTRACT

Composing and Personalizing Next-Generation Telecommunication Services While Managing Feature Interactions

Alessandro De Marco

An emerging trend in software application design is to provide mechanisms to let end-users customize the *look-and-feel* of their usage experience and even extend *behaviour* in order to satisfy personalized requirements. Telecommunication service providers, now offered *open* access to core networks with enhanced multimedia capabilities, are today in demand of solutions to capitalize on the next-generation infrastructure and the market trend in relation to Internet Telephony service creation. Current proposals to meet the demand have the disadvantage of being inflexible or not feasible for the near-term.

In this thesis, we describe our approach for a flexible framework to enable service composition and personalization. Moreover, we demonstrate how our approach may be applied today. Our framework lets end-users, or third-parties acting on their behalf, create added-value by composing existing services in new ways. As a consequence of *empowering* the end-user with an unprecedented level of control over their services, we must ensure that personalized service configurations can and will behave as expected, and not in detriment to the overall system. Therefore, we have also developed a mechanism to *guarantee* the absence of *conflicting* service behaviour, to a certain degree. In providing the guarantee we have dealt with a fundamental problem in Service Engineering, namely, Feature Interaction.

Our solution is based on our enhancement of SERL, a language and framework for managing the triggering and execution of services. We have defined language extensions to let experts impose service composition constraints. Moreover, we have designed algorithms for validating user-defined service configurations against constraints. Finally, we have designed and implemented a *proof of concept* prototype in a Parlay/OSA context which *virtually* composes services at runtime according to the configurations. In two Case Studies, we demonstrate the approach and the added-value created.

ACKNOWLEDGEMENTS

I would like to acknowledge participation of and funding from the SINTEL Research Group at Ericsson Research Canada in early stages of this work. In particular, I would like to thank Dr. Roch Glitho, André Poulin, and Kindy Sylla.

I also acknowledge funding from Fonds NATEQ (Québec) and the Concordia Research Chair in Telecommunications Software Engineering.

Finally, I take this opportunity to express my sincerest gratitude to my thesis supervisor, Dr. Ferhat Khendek, for his technical direction, creative inspiration, and moral support throughout the duration of my Masters degree programme.

CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In this introduction we present our subject matter, the problems that will be tackled, our rationale for selecting these problems, and a concise summary of our results. Our goal is to give the reader a flavour of the work to be presented in the rest of the thesis.

1. Overview

Next-generation telecommunication networks will provide new and enhanced capabilities and enabling technologies for application-layer service development. This emerging infrastructure is often referred to as a multimedia service network, or simply, a service network. Complementing this service network is a new business model, whereby network operators will "open" their networks to 3rd party service providers or developers through secure gateways offering standardized application programming interfaces (API), such as JAIN [19] and Parlay/OSA [28]. This paradigm allows for new players in the service network architecture, new streams of revenue for network operators, and new business opportunities for 3rd party service providers with innovative services to offer.

A rapidly emerging trend in end-user application design is to provide a mechanism for end-users to personalize or customize the *look-and-feel* of their usage experience and even extend *behaviour* in order to meet the unique requirements of each individual. Examples of this can be seen in applications such as Winamp and Microsoft Windows Media Player, for instance. End-users can download *skins* and *plug-ins* to personalize or extend these applications according to their needs or preferences. Service providers for next-generation telecommunications systems have only begun to capitalize on this trend. Personalized *ring-tones*, and specialized *faceplates* are just two examples of simple ways that end-users may personalize their usage experience today. It is expected that *personalization* will expand in scope within the telecommunications domain. End-users will be provided with means to customize their service behaviour to an unprecedented degree, regardless of where the services may reside in the network. More than just personalizing individual service

behaviour, we believe that users should be given the opportunity to define compositions and inter-workings of services to create added value. Currently, the ability for an end-user to compose services according to individual requirements is not available, due in large part to Feature Interaction.

Feature Interaction [7] exists in current telephony systems, but it is expected that the problem will be severely aggravated in next-generation systems due to the openness and distributed nature of the architectures, and the new types of services or features that will be developed [23]. Feature Interaction is said to occur whenever a service affects the behaviour of another, for better or for worse. Even with the limited number of services that exist today, managing the problem is quite costly for service providers. In next-generation systems, the cost could skyrocket without powerful mediators or control mechanisms designed to avoid, or detect and resolve unwanted, erroneous, or malicious service network usage.

2. Problem Statement

Our vision is that the *killer-app* for next-generation systems will not necessarily be a single application or service, but the capability for an end-user to easily compose and personalize a multitude of services as a whole to meet their specific requirements. With the emergence of high-level standardized interfaces for service creation in next-generation networks (e.g. Parlay/OSA), richer signalling protocols (e.g. SIP), and greater processing power in network and terminal devices, we believe that the capability will be available in the near future. Towards realizing this vision, we identify three fundamental problems to address. We then justify our selection in the next section.

- How to enable end-user personalization and composition of services while avoiding unwanted feature interaction. Next-generation telecommunications services are not necessarily limited to call processing, and therefore a flexible framework is required. At the same time, personalization and composition of services must be easy for end-users to achieve, and there must be a clearly defined relationship between user-requirements and services or capabilities available in the network.

- How to guarantee that an end-user's requirements for service behaviour can and will be met. At some point before or during deployment end-user requirements must be checked to be certain that they can be satisfied. Subsequently feedback must be provided guaranteeing eventual realization with a certain degree of confidence.

- How to build a feature interaction management framework within the context of an available service creation architecture such as Parlay/OSA, but not necessarily limit it to that domain. Parlay/OSA is not currently designed to deal with feature interactions at the API level or otherwise; however provisions have been made for future extensions towards this end.

3. Justification of the Issues

In order to justify the issues that this thesis will tackle, we highlight problems that exist with current approaches and certain benefits that solutions will provide.

3.1. Enabling Personalization and Composition of Services

In traditional networks, the end-user does not play a role in feature interaction management. They simply subscribe to services offered by network operators. End-users typically have no control over what resolutions are determined for their *conflicting* service behaviors, and as a consequence, cannot be blamed for problematic feature interactions. The burden of ensuring highest possible quality of service falls on the shoulders of the service provider and yet currently, service providers provide only *best-efforts* service using whatever techniques they deem suitable.

Several existing approaches centralize control of services to one or more Feature Managers. This could allow for both basic and premium quality of service levels if the approaches would ever be applied. End-users would pay extra for a guarantee of better than *best-efforts* service, and such a service level would be realized by using Feature Managers to monitor and control the execution of services for that user. On the other hand, the basic quality of service level with the potential for unwanted interaction could still be realized with minimal cost. Feature Managers and associated processing overhead would simply not

monitor services for users subscribing to the basic plan. Services would have free reign to operate without Feature Manager intervention.

As opposed to centralized Feature Managers, distributed negotiation-based approaches (e.g. distributed agents) may also allow for different quality of service levels; however we expect such mechanisms to be more complex and to demand excessive processing overhead. This is due to the notion that each service in this context is designed to try to negotiate preferred terms, rather than having an external entity decide for it. Regardless of whether a premium service level agreement (SLA) exists, negotiation would take place due to the nature of the service architecture, hence the excessive processing overhead.

Allowing different quality of service levels is achievable today as explained above and further elaborated upon in the state of the art discussion (see Chapter 3). The problem with most of the solutions though is that end-users still would not have a say in the resolutions of interactions, even with premium subscriptions. Resolutions are left up to the service provider and are generalized for all users. Therefore the viability of Feature Interaction Management solutions in the competitive marketplace is poor. In general, users are more likely to be willing to pay for more functionality, rather than marginally improved quality of service.

Only two approaches, CPL [24] and ACCENT [1], have considered the benefits of allowing services to be composed according to user requirements in attempt to achieve functionality that is more useful than individual service operation. This added-value helps to justify a higher premium service level cost to end-users and is much more marketable.

CPL enables end-user specification of service behavior and guarantees the absence of feature interactions due to the nature of the language. On the other hand, it is not flexible enough to support behavior that is not within the realm of Call Processing. ACCENT is more flexible; however it seems to lack a clear mapping between policies and features or network capabilities. We insist on having a clearly defined relationship to services in the network because powerful billing architectures already deployed today depend on it. The billing architectures do not track rule or policy invocation, but rather service subscription

and usage. Moreover, end-users have grown accustomed to the concept of subscribing to a service and paying for the subscription. ACCENT lacks this mapping, and in certain respects CPL also presents a concern since it is dependent on a user's knowledge of signaling protocol capabilities and not services.

3.2. Guaranteeing Service Behavior

More than enabling personalization and composition of services, we must guarantee that user requirements can and will be met. Justification for this is simple – users must know that *they will get what they pay for* – nothing more, and nothing less. To not be able to guarantee this would disregard the whole concept of providing premium quality of service and thus hinder the whole approach.

3.3. Parlay/OSA and Beyond

Finally, since no previous work has considered feature interaction specifically in the context of Parlay/OSA, we must justify our desire to explore the issue in this context. We stress the importance of Parlay/OSA being the only *technology-agnostic* platform for service creation adopted by 3GPP. This means that any service provider wishing to ensure that the services that they develop will not become obsolete as network technologies (e.g. SIP) or programming languages (e.g. Java) evolve must develop their services for this API. Moreover, since standards defined by 3GPP are implemented by most of the important equipment manufacturers around the world, the potential exists for a service provider to target a global market when developing services.

From a more technical perspective, Parlay/OSA clearly defines points of control in the service architecture which are candidates for the location of Feature Managers. In addition, the richness of the API suite allows for the creation of interesting new services incorporating Call Control, Mobility, Presence and Availability, enhanced Instant Messaging (i.e. more than just text), and more.

Parlay/OSA is not without open issues however. Feature Interactions may occur at different levels in the service network architecture. So for example, interactions may exist between Parlay/OSA and SIP as an underlying signaling protocol. These types of interactions are

not within the scope of this thesis since we focus on services developed for the API only. Many also consider Parlay/OSA to be too complicated for the development community outside of the telecommunications domain or without prior experience programming telephony services. Therefore, higher-level abstractions of Parlay/OSA are being developed such as Parlay-X. In addition, a working group is studying the potential to facilitate compatibility between high-level Web Services and Parlay/OSA.

Hence, we see that as Parlay/OSA becomes more popular, it is probably the best platform to consider for a solution today. For the longer-term however, provision must be made for our solution to be adapted to new technology platforms, which may eventually replace Parlay/OSA.

4. Synopsis of Results

In this thesis, we describe a mediation system, which provides controlled end-user composition and personalization of service network capabilities and high-level services without imposing unwarranted restrictions that would obviate the benefits of the functionality offered. The system we have designed is a significant enhancement of the Service Execution Rule Language and Framework (SERL) [26, 27, 28], and we have designed and implemented it in a Parlay/OSA context as a *proof of concept*.

As will be reported, we have successfully developed a generic framework in a single-network-component, single-user (SUSC) environment. We allow users to define rules for composed service behaviour according to their individual requirements. We then validate a user's configuration to determine if the requirements can be met, and in doing so, we make sure that resolutions exist *a priori* for any potentially disruptive feature interactions. Our validation scheme abstracts a user's rules and checks them against constraints for service composition defined by an expert. If no constraints are violated during offline validation of a configuration, then the user's configuration is deployed and activated. Once activated, an online processing engine, which we have devised, endeavours to control the invocation and execution of services according to the user's personalized configuration.

5. Organization of this Thesis

This thesis is organized as follows. In the next chapter, we briefly review Next-generation IP Multimedia Service Networks, Parlay/OSA and SERL. In Chapter 3, we summarize the state of the art by presenting Feature Interaction approaches in current-generation networks, and by describing technologies for service composition and personalization in next-generation telecommunication networks. Chapters 4, 5 and 6, describe our approach to deal with the problems that this thesis tackles. In Chapter 7, we discuss two Case Studies demonstrating applications of the whole approach. Finally, we conclude in Chapter 8 by summarizing our contributions and hinting at future work.

In this thesis, we use the terms *feature* and *service* interchangeably. Similarly, we often use the term *call* to refer to the concept of a *session*, which is more general that a typical telephony call today. We also consider *Internet Telephony*, *IP Multimedia* and *IP Telephony* to mean the same thing.

CHAPTER 2

BRIEF REVIEW OF NEXT-GENERATION NETWORKS, PARLAY/OSA, AND SERL

In this chapter, we provide a brief introduction to Next-Generation Service Networks, Parlay/OSA, and SERL. Our goal is to define the general context of the work presented in this thesis.

1. Next-Generation Service Networks

Next-generation service networks, often referred to as IP Multimedia or 3G networks [33], combine the advantages of the IP and cellular phone technologies in a converged framework. This convergence will enable a large number of new services combining voice, data, and video, with quality of service assurance. In the overall service architecture for IP Multimedia service networks as shown in Figure 1, we can distinguish three domains: the IP core network, the Camel network and the third party service provider domain. Third party service providers will typically host services on an application server and be provided with access to network capabilities and services through an Open Service Architecture (OSA) gateway. The gateway(s) are referred to as Service Capability Servers (SCS).

The SCS provide a mapping between the standardized OSA interface and the technology of the network. In Figure 1, the SCS maps to the Home Subscriber Server (HSS), essentially a database of user profiles, a Serving Call Session Control Function (S-CSCF, basically a SIP Proxy Server), and indirectly to an IP Multimedia Service Switching Function (IM-SSF, a gateway for Camel services). It is the responsibility of the SCS to inter-work with these systems by implementing ISC or Sh, which are 3GPP protocol identifiers for enhancements to well-known protocols such as SIP.

Within the IP core network, services may be hosted on SIP Application Servers. There is provision for Service Capability Interaction Management for managing feature and service interactions in this context, but no applied solutions that we are aware of yet.

Legacy services exist in a Camel Service Environment and inter-work with the core network using the Camel Application Part (CAP) or Mobile Application Part (MAP) protocols. In passing, the Cx interface is a 3GPP-specific protocol for interrogating or updating the HSS from the S-CSCF.

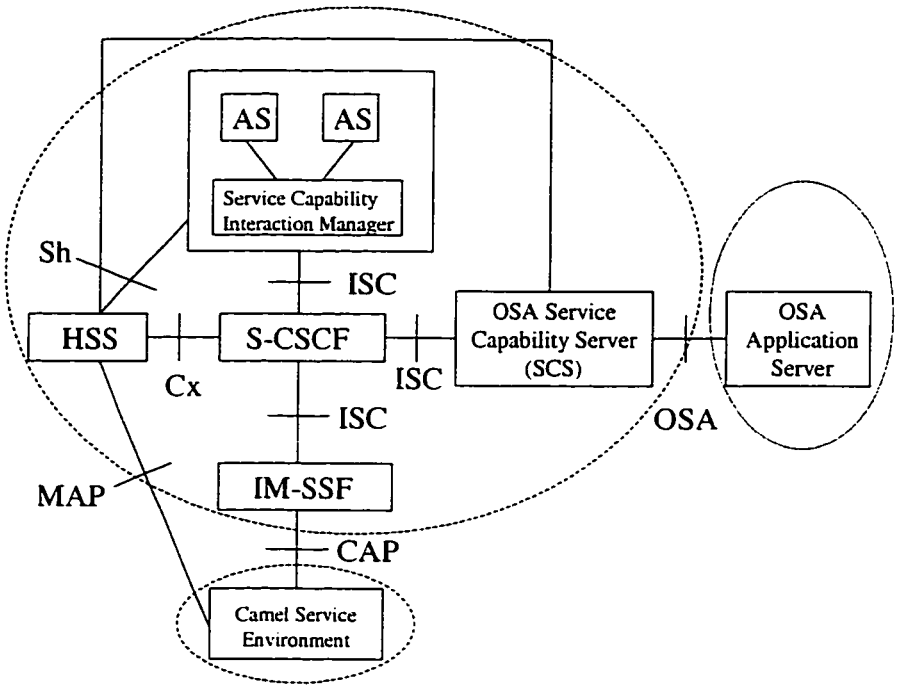

Figure 1: 3G Service Network Architecture

2. Parlay/OSA

3GPP OSA is based on the Parlay suite of APIs [28], which allow for the provisioning of services independently of the underlying network technologies. The goal of Parlay/OSA APIs is to abstract network resources and capabilities into a set of interfaces to allow service providers, including third parties, to use and control network resources in a standard manner. Parlay/OSA API specifications are technology independent, and interface descriptions (i.e. IDL) have been developed for two middleware platforms: CORBA and DCOM. The APIs consist of two categories of interfaces: framework and services. Framework interfaces provide the supporting capabilities for the service interfaces to be discovered, securely accessed, and managed. Service interfaces expose the capabilities of the underlying network such as Call Control, User Interaction, Mobility and Connectivity

Management. API service capabilities can be seen as services provided to any service provider.

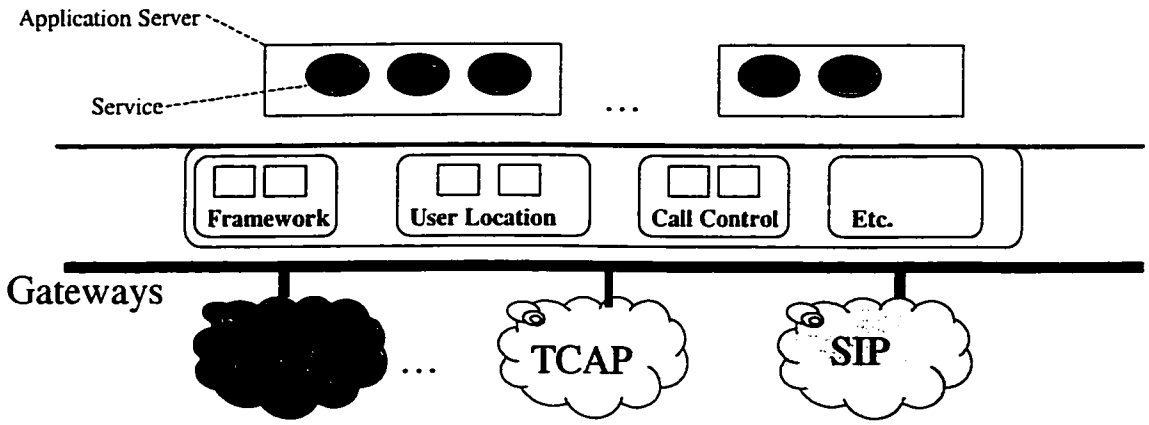

Figure 2: Parlay/OSA APIs

The Call Control interfaces (i.e. Generic and Multi-Party) allow for instantiating and routing calls from the application. They also allow for asynchronous monitoring of call-related events, and the subsequent assumption of control of calls when event properties satisfy certain conditions.

A service can interact with the end-user using the User Interaction interface. For instance, a service can play an announcement to a given call leg, collect a sequence of digits from an end-user, or send a simple instant message. A complete description of the interfaces is given in [28].

2.1. Service Implementation Example in a SIP Network

In Figures 3 and 4, we show sequence diagrams for an *Interactive Call Screening* (ICS) service. We explain each message in Tables 1 and 2. This service will screen all calls made by user A through a third-party, namely user C. Each call attempt made by user A will initially be redirected to user C for approval. Subsequently, an instant message will be sent to user C asking whether user A may have permission to proceed with the original call. If

user C responds positively, then the call established between users A and C is disconnected, and re-established between user A and the original callee. If denied permission, the call is simply rejected. This service may be used by a parent wishing to screen outgoing calls made by their child, or by a manager wishing to screen outgoing calls made by an employee.

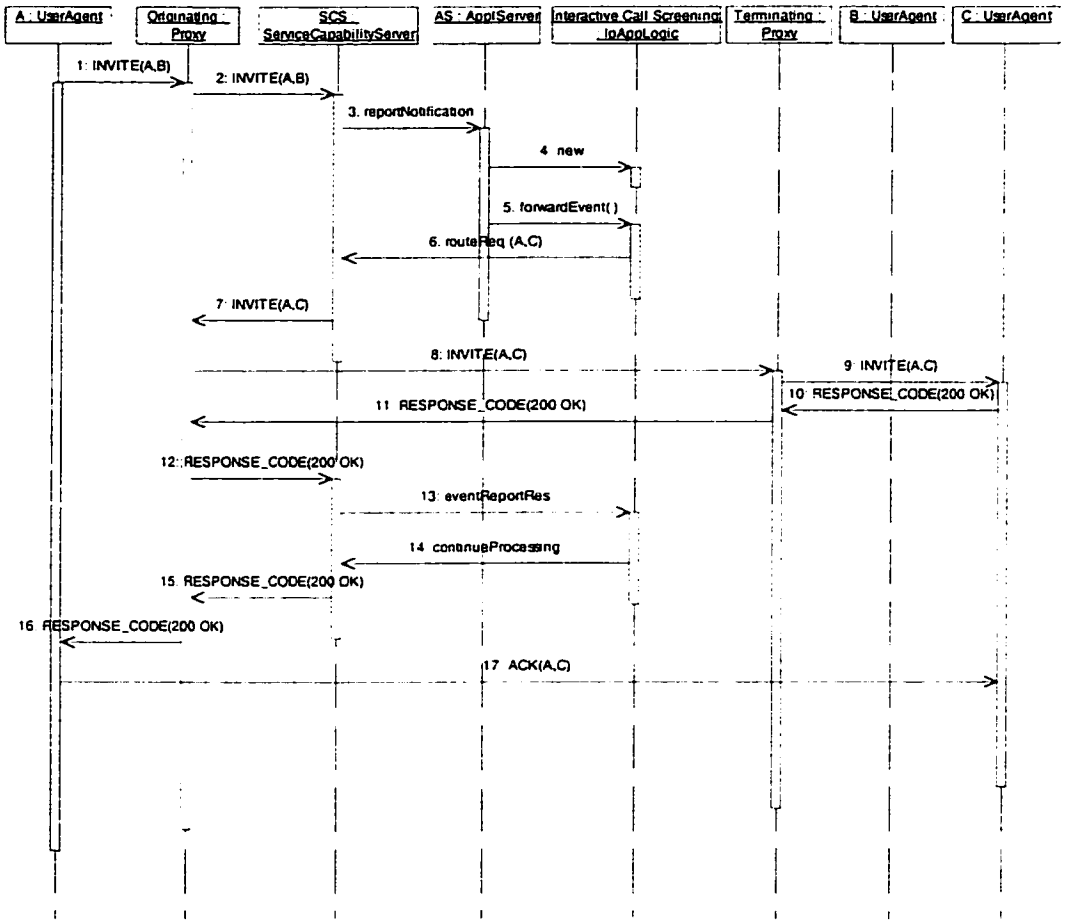

Figure 3: Interactive Call Screening (Part 1)

If we refer back to Figure 2, we see that Parlay/OSA Service Capability Servers functionality may be mapped to a SIP network. In our example, the Multi-Party Call Control SCS and the User Interaction SCS are implemented on the same node. Using SIP messages, they communicate with SIP Proxy Servers (Originating and Terminating with

respect to the call flow), which in turn communicate with SIP User Agents. The ICS service resides in an Application Server and communicates with the SCS using Parlay/OSA API method calls. We have abstracted some of the method invocations to simplify our diagrams and explanation. The important idea to remember is that in principle, the service developed need only be concerned with the interactions between the ICS service and the SCS. All complexities in protocol behavior are hidden. For a detailed analysis of issues related to Parlay/OSA to SIP mapping, the reader may refer to [32] .

Table 1: Interactive Call Screening Messages (Part 1)

1: INVITE(A,B)	User A invites User B to a call session. Since User A does not know User B's IP address, the message is sent to the Originating Proxy.
2: INVITE(A,B)	The Originating Proxy is configured to forward all incoming requests to the SCS.
3: reportNotification	The Multi Party Call Control SCS notifies the IpAppMultiPartyCallControlManager which is the Parlay/OSA interface implemented by the Application Server.
4: new	Since User A has subscribed to the ICS service for all outgoing calls, ICS is created for this call.
5: forwardEvent	Once created, the ICS service is now ready to receive the event information about the call.
6: routeReq(A,C)	ICS determines that outgoing calls from user A need to be screened by user C. A new IpCallLeg is created (not shown) and routed to user C.
7: INVITE(A,C)	Since the request received by the SCS originated from the Originating Proxy, subsequent messages follow the same path.
8: INVITE(A,C)	Originating Proxy does not know about user C, so it forwards the message to the Terminating Proxy.
9: INVITE(A,C)	The Terminating Proxy receives the message, looks up user C's IP address and forwards the request.
10: 200 OK	User C answers the call.
11: 200 OK	Terminating Proxy forwards the response.
12: 200 OK	Originating Proxy forwards the response to the SCS following the same path as the request.
13: eventReportRes	An event report is generated on this leg and forwarded to the service. The service knows that user C has answered.

14: continueProcessing	The service instructs the SCS to continue with any administrative duties involved in setting up the call.
15: 200 OK	User A needs to be told that a call has been established with user C.
16: 200 OK	User A is notified of the call setup.
17: ACK(A,C)	User A acknowledges establishment of the call session with user C.

As of this point in the call, users A and C may communicate with each other. Behavior so far mimics a typical call forwarding service. This purpose of establishing the call between users A and C is to allow user A to explain the nature of the outgoing call to C when asking for permission for the call to proceed.

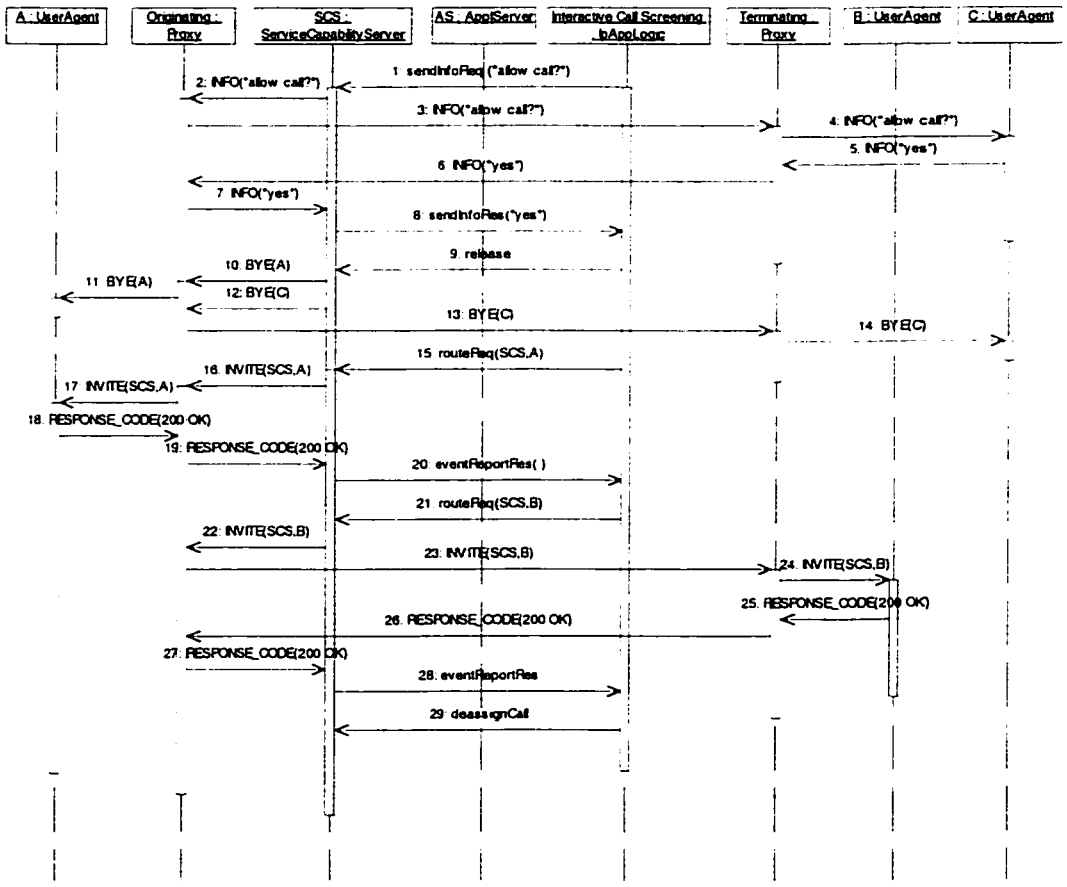

Figure 4: Interactive Call Screening (Part 2)

Figure 4 shows the second part of the service behavior, which queries user C for permission to let user A call user B.

Table 2: Interactive Call Screening Messages (Part 2)

1: sendInfoReq ("allow call?")	Upon establishment of the call between users A and C, the ICS service sends an information request querying user C for permission to allow user A to connect with user B. User A and C may communicate during this time so that user A may explain the nature of the call.
2. INFO("allow call?")	The User Interaction SCS translates the method into a SIP INFO message, and directs it towards the Originating Proxy.
3. INFO("allow call?")	Since the information request is addressed to user C, the Originating Proxy forwards the request to the Terminating Proxy.
4. INFO("allow call?")	The Terminating Proxy forwards the message to user C.
5. INFO("yes")	User C agrees to let the call proceed.
6: INFO("yes")	The Terminating Proxy forwards the message upstream.
7: INFO("yes")	The Originating Proxy forwards the message to the SCS.
8: sendInfoRes("yes")	The User Interaction SCS forwards the event to the service.
9: release	The ICS instructs the Multi Party Call Control SCS to release the call established between users A and C.
10: BYE(A)	This message is sent to the Originating Proxy to notify user A of a disconnection.
11: BYE(A)	Message forwarded from Originating Proxy to user A.
12: BYE(C)	This message is sent to the Originating Proxy to notify user C of a disconnection.
13: BYE(C)	Message forwarded from Originating Proxy to Terminating Proxy.
14: BYE(C)	Terminating Proxy forwards the message to user C.
15: routeReq(SCS,A)	After tearing down the old call, the service instructs the SCS to establish a call leg between itself and A.
16: INVITE(SCS,A)	SCS invites user A to a session via the Originating Proxy.
17: INVITE(SCS,A)	Originating Proxy forwards the request to user A.
18: 200 OK	User A answers.
19: 200 OK	Originating Proxy forwards the response to the SCS.

20: eventReportRes	The SCS forwards the event to the service.
21: routeReq(SCS,B)	The service now attempts to create and route a call leg from itself to user B.
22: INVITE(SCS,B)	SCS invites user B to a session via the Originating Proxy.
23: INVITE(SCS,B)	The Originating Proxy forwards the request to the Terminating Proxy.
24: INVITE(SCS,B)	The Terminating Proxy forwards the request to user B.
25: 200 OK	User B answers.
26: 200 OK	The Terminating Proxy forwards the response to the Originating Proxy.
27: 200 OK	The Originating Proxy forwards the request to the SCS.
28: eventReportRes	The SCS forwards the event to the service.
29: deassignCall	The service is no longer interested in managing the call. It instructs the SCS to handle the call on its own (basic call handling behavior). There is no need for further events to be sent to the service.

2.2. Parlay/OSA versus Predecessors

An important distinction between Parlay/OSA and older platforms for service development (e.g. IN) is Parlay/OSA's standardized interface at a higher level of abstraction. Together, these allow for vendor-independence and improved decoupling of services from the network, thus minimizing maintenance costs as underlying networks evolve, and facilitating more rapid service development. Moreover, the Framework APIs play a prominent role in opening networks to a community of 3rd party service providers that is expected to be quite large in number. This type of open access to the "trusted" network operator's domain has never been offered before. To probe further on the subject of Parlay/OSA and future directions of the technology, the reader may refer to [27] and [14].

2.3. Other Service Creation Technologies

There exist other technologies for service creation in Next-Generation Networks as well. These are out of the scope of our work, but they will be mentioned. Session Initiation Protocol [17], the signaling protocol adopted by 3GPP is complemented by three protocol-specific service creation technologies, namely SIP-Servlets [22], SIP-CGI [25], and CPL

[24]. JAIN [19] is a Java-language specific standard which is very similar to Parlay/OSA. VoiceXML [35] is an XML-based language for describing voice applications, typically Interactive Voice Response (IVR) services. The types of applications that are possible depend on the platform which hosts the voice application. Essentially, all of these technologies are specialized for a particular protocol, language, or execution environment. Parlay/OSA is the only truly *technology-agnostic* standard. For a concise overview of these service creation frameworks, the reader may refer to [15].

3. SERL: Service Execution Rule Language

SERL [26, 27, 28] is a language and a framework for managing the triggering and execution of services. It is based on *condition-action* rules, and a processing model involving interception of events, matching of event conditions to rule triggering properties, and then application of matched rules. SERL was originally developed for SIP, where deployment of rule processing engines implementing the model is more amenable to Proxy Servers, but not unimaginable in User Agents. The XML-based language is flexible enough to support other technologies, such as Parlay/OSA, and even certain heterogeneous environments.

SERL rules are grouped into Rule Modules, each with one owner. The owner of a Rule Module is typically a subscriber to services affected by the rules in the module. Services are classified into groups according to their behavior, and each group is assigned a Processing-Point identifier. Processing Points refer to the points in the call-signaling timeline where services within a certain group may be invoked. Services are triggered in response to events flowing downstream (i.e. requests) or upstream (i.e. responses) through a node. Events usually originate from the network, or from services. When triggering rules are encoded, they take on the same Processing-Point identifiers as the services they relate to. A rule-search algorithm, set to run upon the occurrence of events, takes into consideration Processing-Point identifiers, priority of rules, as well as the relevant event information (a.k.a. event context) when searching for matching rule conditions. Rule actions may involve delaying, overriding, canceling or generating events.

SERL suggests criteria for defining the service groups based on a service's potential actions or behavior in response to events, but the framework allows for enhancement of these criteria, which may lead to alternate groupings. Suggested in [31], services to be invoked before all others are those that may affect routing. Services that may affect the message payload, but not routing information, are invoked next. Finally, a third group exists, where services are invoked, but cannot modify message content at all (e.g. call logging). This ordering of services invocation is partially realized using Processing Point identifiers PP1/PP-1, PP2/PP-2, and PP3/PP-3, respectively, where the sign of the Processing Point identifiers relates to the direction of event-flows through a node, either downstream (+), or upstream (-). Processing Point 0 is used for services that may be invoked both downstream and upstream.

In Figure 5, we see an example of event-flow downstream. An INCOMING_CALL event is intercepted by Bob's Terminating SERL node. The rule-search algorithm implemented within the node determines that S1 and S2 need to be invoked for calls where Bob is the callee (hence our reason for distinguishing the node as Terminating). Since S1 is a service belonging to Feature Group 1 and assigned PP1, it shall be invoked before others. Similarly, S2 is assigned PP3, and will be invoked last.

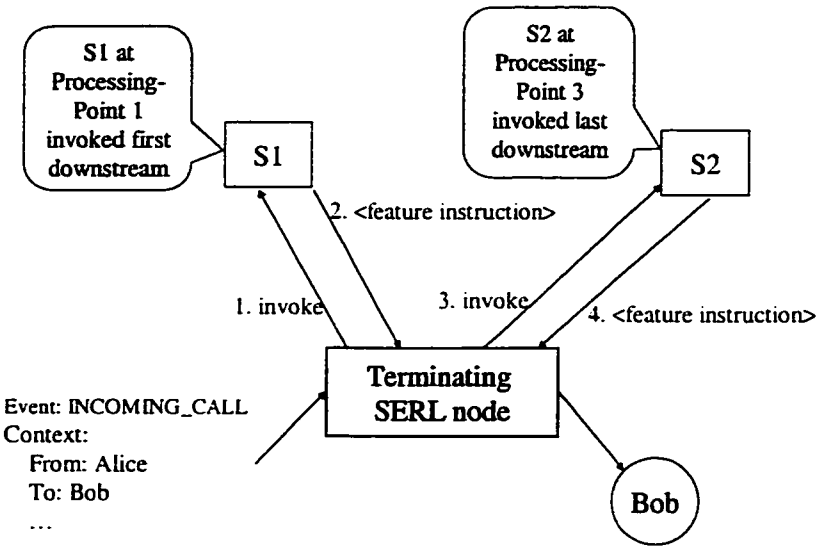

Figure 5: Event-Flow Downstream

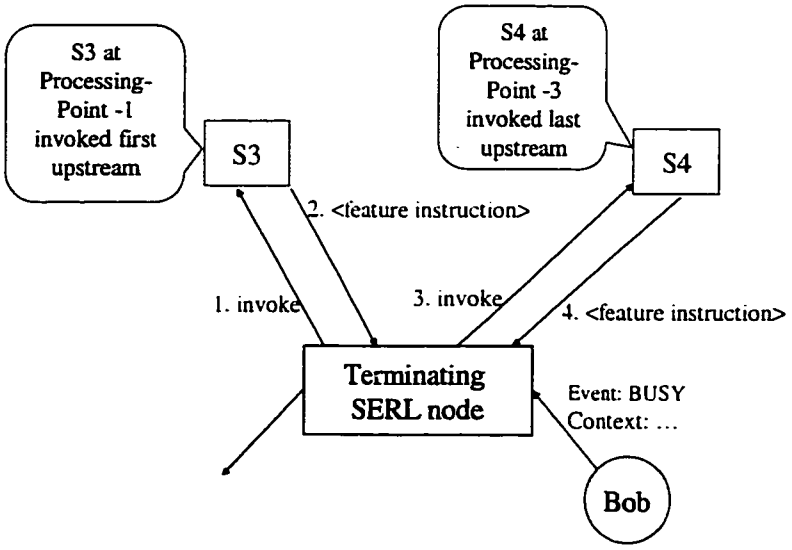

Figure 6: Event-Flow Upstream

Figure 6 demonstrates the event-flow upstream in response to the incoming-call request shown in Figure 5. Here, Bob is busy and cannot accept the connection request from Alice. Upon intercepting the BUSY event from Bob, the SERL node invokes S3(PP-1) and S4(PP-3), according to the Processing Point protocol.

In the example, no services are assigned PP0, PP2, or PP-2, so these processing points are bypassed completely. In addition, the actions taken by the SERL node in response to feature instructions are not specified. We demonstrate one possible service inter-working scenario in the figures above, but in an alternate case, it would be possible for S1, to be invoked and subsequently generate a feature instruction to inhibit further processing downstream. Event-flow would reverse immediately, and therefore processing would skip PP3 and further interaction with Bob, and then transition to PP-1 next. Finally, note that we have shown at most one service invocation per Processing Point, but depending on the services deployed in the system, there may be more, and therefore the potential for feature interactions still exists for services assigned the same processing point.

In addition to managing the execution and triggering of services, SERL may access system capabilities like a database, a Presence server, a Location server, Web services, etc. Examples demonstrating such functionality are not provided in the Internet-Drafts, however such functionality is implied. It is the responsibility of the developer of the SERL engine to build-in adaptors for communication with such network capabilities or distributed services.

SERL is not a language to describe the behavior of services, nor is it intended to be used to detect feature interactions. Rather, it is a mechanism to allow for the application of feature interaction resolution policies in a SUSC [7] context. It is assumed that potential interactions are known *a priori*, and knowledge about how to resolve interactions is encoded in rules. Ordering through Processing-Points and priorities are the only means available to enforce resolutions. For instance, when several rules are matched, SERL does not define a resolution policy other than a simple ordering scheme based on priorities defined *a priori*.

CHAPTER 3

FEATURE INTERACTION, PERSONALIZATION, AND COMPOSITION: STATE OF THE ART

In this chapter we present a summary of the state of the art in Feature Interaction, and Personalization and Composition of IP-based multimedia services. We aim to provide background information for our contribution in the area. Most of our discussion of Feature Interaction stems from two comprehensive review papers, namely [6] by Calder et al., and [20] by Keck and Kuehn. To a great extent, research in the area has it roots documented in the Proceedings of the International Workshop Series on Feature Interactions in Telecommunications and Software Systems [3, 5, 9, 11, 14, 18]. As for Personalization and Composition, relatively new topics with respect to the Telecommunications Service Engineering domain, we select a few examples of approaches which showcase emerging trends in the area.

1. The Feature Interaction Problem

"In software development a *feature* is a component of additional functionality – additional to the core body of software" [6]. An *interaction* is said to be a "behavioural modification" [6] of normal feature operation in the presence of one or more others. A traditional example in the Telecommunications domain is the interaction between Call Forwarding (CF) and Originating Call Screening (OCS). Assume that Alice subscribes to OCS and is prevented from calling Bob because Bob's number is in her screening list. Assume also that Charlie subscribes to CF, such that all calls directed to Charlie shall be forwarded to Bob. If Alice calls Charlie she will be connected to Bob. We see that OCS's goal of preventing a connection between Alice and Bob was compromised due to the presence of CF. This type of interaction involves multiple users, however there exist a category of interactions for a single user as well. An example of this type involves Call Forward on Busy (CFB) and Call Waiting (CW). If a single user subscribes to both services, there is ambiguity as to which one shall trigger when the user is busy and an incoming call request is received. These two services have conflicting actions – forward the call or put the call on hold, and therefore

triggering in parallel does not make sense. In [7], Cameron et al. provide a benchmark for similar feature interactions in traditional telephony.

In next-generation networks, the problem is expected to become more widespread, more apparent to end-users, and more complicated to deal with. This is due to the fact that in emerging service networks, there will be more services, more players in the market due to the openness of the architectures, and the service logic itself will be distributed away from centralized servers under strict control towards the edges of networks and terminal devices in "un-trusted domains" [23].

2. Challenges and Classification Frameworks

Three major challenges may be identified, namely, how to detect or predict interactions, how to determine the best resolution for them once detected, or alternatively, how to avoid or prevent them altogether. All known feature interaction research can be said to address one or more of the three challenges.

In surveying previous feature interaction work, a classification framework is needed. Keck and Kuehn identify two important criteria for comparison of work, namely *Approach*, and *Method* [20]. They define other criteria as well, which we consider somewhat less significant and choose not to discuss. They refine the generally accepted classification of approaches [4, 8] as shown in Table 3. One may view the classification of approaches as a possible set of answers to the questions *what challenge is being addressed, and what is the general approach towards a solution?*

Table 3: Classification of Approaches

Detection
Interaction
Interference (undesired)
Resolution
Restriction
General
Situation-specific
Integration
Cooperation
Conscious
Oblivious
Prevention (avoidance)
Structural (system structure)
Procedural (specific design process)
Management

Keck and Kuehn distinguish between detection of *interactions* and *interference* by defining interference as a class of undesired interactions, as opposed to interactions in a general sense. They state that for realistic systems "it is impossible to prove the absence of mistakes" and that an interaction is not necessarily undesirable [20].

With regards to resolution, the authors identify three strategies, namely, restricting the behaviour of one or more services involved in the interaction, integrating the services into a larger unit of functionality, and co-operation where services co-ordinate with each other *consciously* (i.e. with knowledge of the other) or *obliviously* though a mediation mechanism.

Approaches for preventing interactions have been classified into two groups. The first group encompasses techniques where interactions are guaranteed to not exist due to the nature of the system structure. The second group involves procedures incorporated into the design process to avoid interactions altogether. The authors point out that due to the complexity of the problem, management issues must be added to the list. All approaches related to managing test cases (i.e. for detection), scenario-filtering (i.e. for detection and resolution), and so on, are grouped under this heading. This terminology is somewhat

misleading since in most literature, *feature interaction management* relates to the process of detecting and resolving interactions as a whole, and not a sub-classification of solution approaches.

Methods employed when applying approaches can be classified according to Table 4. Here one may view the Classification of Methods as a solution set to the question *what method will be used in applying the approach?*

Table 4: Classification by Method Applied

Design oriented Methods
Feature Design
Feature Execution Control
Feature Execution Environment
System design and architecture
Analytical methods
Formal techniques (verification)
Informal techniques (heuristics)
Experimental techniques (testing, simulation)

Design-oriented approaches involve design of features in consideration of interaction issues, development of mechanisms for feature execution control, provisioning of specialized execution environments, or the design of system architectures to appropriately deal with certain aspects of the applied approach. A great number of analytical methods involve formal techniques, however informal and experimental techniques also exist. Formal methods, some informal methods, and simulation, by their very nature, apply to models of systems, whereas testing and certain other informal techniques apply to existing systems.

The work of Calder et al. [6] focuses on three major research trends at a higher abstraction level than the scheme used by Keck and Kuehn. The trends that they have identified are *Software Engineering*, *Formal Methods*, and *Online Techniques*, and intuitively, these may be mapped back to Approaches and Methods identified by Keck and Kuehn in [20].

3. Important Concepts

We briefly summarize some of the important concepts introduced by previous work. We align the concepts with the trends identified by Calder et al, and relate them to Approaches and Methods defined by Keck and Kuehn.

3.1. Software Engineering

Software Engineering approaches to deal with feature interaction can be seen as adaptations or specializations of approaches that deal with issues in software development in general. The major focus has been on eliminating feature interactions, either by introducing additional steps in the software development process model or by applying informal techniques at one or more stages in existing process models. Two trends for informal techniques are discernable, namely, imposing "a service architecture that constrains designers to provide 'safe' arrangements of features" [6], and *filtering* to make the problem more manageable by eliminating unlikely service combinations [6].

When trying to relate these approaches to the classification framework proposed by Keck and Kuehn, we see that filtering techniques are considered to be management approaches typically associated with any of the analytical methods. The other software engineering approaches would be associated with feature design or system design methods for detection, resolution, or prevention.

3.2. Formal Methods

Formal Methods are much more rigorous than software engineering approaches, and they require the introduction of specialized notation and models for dealing with issues at a higher level of abstraction. They do have one important characteristic in common with Software Engineering approaches and that is that they are typically applied offline. Published results describe how these methods have been useful in detecting the presence of interactions that were expected in the first place (i.e. validation), but it is evident that few methods have succeeded in the ultimate goal of detecting new or unpredicted interactions.

There are two basic models; the property and behavioural models. Formal methods in the former group check whether system properties which are satisfied when a feature runs

independently, are still satisfied when two or more feature run together. Behavioural models examine the combined operation of two or more services and define interactions in terms of "reachability, termination, deadlock, non-determinism, or consistency" [6]. There also exist methods which combine property and behavioural models.

3.3. Online Techniques

"Online techniques are intended to be applied at service runtime in a network" [6]. Typically, detection is quite difficult to achieve online. Resolution on the other hand may or may not depend on *a priori* knowledge about how to resolve interactions, but obviously, this knowledge helps to improve overall performance by eliminating the need for a resolution negotiation phase. On-line techniques have many advantages over their offline counterparts, however they inherently exhibit processing overhead which degrades performance – a serious concern in such real-time systems.

Online techniques always require methods for monitoring and control, termed either *feature execution control* or *feature execution environment* by Keck and Kuehn [20]. Two classes of online techniques exist, and they are distinguished by the location of control. Feature Managers are centralized controllers which monitor and control the execution of services. Services do not usually have mutual knowledge of each other. Negotiation based approaches on the other hand require services to co-ordinate and communicate with each other directly, through shared public data space, or through their respective agents using distributed artificial intelligence techniques. Hence the control logic is said to be distributed.

For Feature Manager approaches, gathering of information about services is critical in order to detect interactions and subsequently resolve them. Experimental approaches allow services to run independently in test networks, and mechanisms are installed to monitor their execution. Alternatively, other approaches gather information about services in live systems and take action upon entry or attempts to enter into unstable system states. Such resolution actions may either be defined *a priori* in rule tables or based on roll-back mechanisms.

4. Summary of Existing Work and Future Directions

Keck and Kuehn classify most feature interaction research up to 1998 according to their defined criteria. They note that sometimes there is overlap when classifying certain work since it contains aspects of several categories. There are three important conclusions from their work as listed below.

- Formal methods outnumber others by an overwhelming amount for feature interaction detection. However, due to the nature of the techniques, there is little support for real implementations or dealing with legacy systems.

- Some promising approaches for resolution rely on mutual knowledge of other services involved in interactions, which is a problem since such information will not likely be available in a multi-provider, deregulated, and competitive market.

- Prevention approaches depend on supporting infrastructure, which will not exist in the near term.

In their more recent review of the state of the art [6], Calder et al. essentially support the conclusions, and they additionally conclude that much work needs to be done to address feature interaction in emerging telecommunication systems.

Due in large part to deregulation in telecommunication industry, next-generation systems will be open to any 3rd party wishing to offer services. As more and more players enter the market, competition will increase. In order to remain competitive, providers will not be interested in making service specifications public or exposing their services to scrutiny by Feature Mangers in a network operator's domain, for instance. Calder et al. explain that inter-working with such services will be facilitated by new service creation platforms (e.g. Parlay/OSA, JAIN, SIP-Servlets), however the fundamental problem of Feature Interaction will remain, and in fact become more complicated to deal with.

Calder et al. classify feature interaction work for traditional networks according to three areas in their review, but their main goal is to show that feasible solutions for emerging telecommunications systems will need to draw on aspects of all of the areas [6]. In other

words, *hybrid* solutions will be required for next-generation systems. This requirement for hybrid solutions is widely accepted; however there is a current lack of published results actually describing more than elaborations on requirements. Therefore, except for a few notable exceptions as will be explained in the next section, the current state of the art clearly lacks feasible solutions for next-generation networks.

5. Approaches for Personalization of Next-generation Services

In the following section, we discuss ongoing work towards solutions for personalization of services in next-generation networks. We limit our discussion to these two items because we consider these two to be the most closely related antecedents to our work. Both approaches exhibit characteristics of offline and online approaches from the past, and they may be applied in centralized or distributed contexts. The idea is to allow the end-user to play a more prominent role in service creation and provisioning by providing means for them to specify the behaviour of their services according to their own requirements.

5.1. Call Processing Language

Call Processing Language (CPL) [24] allows an end-user to specify their preferred behavior for their call processing services. The language restricts itself from being Turing-complete, and the main intention of restricting its expressiveness is to eliminate the potential for feature interaction problems. At the same time though, it hinders flexibility a great deal.

Users specify condition-action rules for service behavior. Conditions are based on a restricted set of potential events that may occur, and actions, similarly are restricted to one of a predefined set of behaviors. CPL scripts are checked offline, but as mentioned, if scripts are written according to the specifications of the language, then it is impossible for interactions to exist within a single script. For example, only one service may be invoked when the end-user receives an incoming call attempt, but happens to be busy.

Feature interactions between scripts for multiple users possibly executing on multiple servers or terminal devices are an open issue, even though the authors hint at a scheme

whereby an order of execution for scripts is determined according to certain guidelines, and a dependency is required of the underlying signaling protocol.

5.2. ACCENT

One of the main objectives of the ACCENT Project [1] is to define a Policy Description Language (PDL) to allow individual end-users to express their own policies for call processing, which happen to be similar to rules. ACCENT requires such policies to be defined at a very high-level of abstraction, not necessarily in terms of a particular call-model or condition-action service triggering. It attempts to be much more flexible than CPL. Intuitively the advantages of this include a potentially simplified language for non-experts to use, dynamic-binding to services offering certain quality of service or price advantages, and more. In defining policies at a high-level of abstraction, interpretation is required to relate policies to actual services available in the network or participants in the call. The actual interpretation procedure is still an open issue.

One of the requirements of PDL is that it be amenable to static (offline) policy conflict analysis. Interactions between policies are expressed using policy composition operators, and techniques for analysis are adapted from ANISE [2]. In a multi-user context, run-time detection and resolution schemes are envisioned, but not yet completely defined.

6. Service Composition

The development of component-based software that can be reused to create added-value while incurring only minimal integration costs is considered by many to be one of the fundamental goals of Software Engineering. In light of legacy software that may not conform to componentization standards, incompatible standards, and the competitive nature of the industry in general, the path towards achieving the goal in the large has been quite challenging to date. Thankfully, a single standard is beginning to dominate the horizon, namely Web Services [37]. In fact, with the convergence of distributed services over an IP-core platform, even traditional Telecommunication Service Providers have begun to adopt the paradigm. For example, the Parlay Group [28], where the majority of members have their roots in the Telecom world, is working on a Web Services compatibility strategy. The point is that distributed service composition approaches today are evolving towards

standardization and are motivated by the emergence of the World Wide Web as the ubiquitous distributed computing platform and XML as the *lingua franca* for information exchange. In the next few paragraphs, we select a few examples of projects that demonstrate concepts in service composition with respect to IP-based Multimedia services. We also briefly discuss Web Services and service composition in that context.

6.1. Distributed Feature Composition

Distributed Feature Composition (DFC) [18] is based on a pipe-and-filter architectural design pattern, where "customer calls are processed by dynamically assembled configurations of filter-like components: each component implements an applicable feature, and communicates with its neighbors by featureless internal calls that are connected by the underlying architectural substrate" [18]. By employing a pipe-and-filter architecture, DFC benefits from the following advantages: "feature components are independent, they do not share state, they do not know or depend on which other feature components are at the other ends of their calls (pipes), they behave compositionally, and the set of them is easily enhanced" [13]. DFC is a virtual architecture which can be mapped to an implementation architecture for call processing, as was done in the ECLIPSE project at AT&T [12]. The order of the feature arrangements in a configuration is determined by the type of the feature and whether it is owned by the callee or caller. For each *usage* or customer call, all eligible features are instantiated in an arrangement. Features observe events that are piped through them, acting transparently until a triggering event is intercepted causing the feature to take action. DFC was novel at the time of its introduction for its use of the pipe-and-filter architecture for service composition; however its applicability in next-generation networks with multiple providers and a wide range of different types of multimedia services is unclear.

6.2. CPL and ACCENT as Service Composition Approaches

As we have seen previously, Call Processing Language and ACCENT/Policy Description Language let users define service behaviour according to their personalized requirements. Both languages implicitly depend on the composition of services in order to realize the overall service behaviour required. For example, a CPL script may indicate that a call must be forwarded if a certain event occurs, or an email sent if another event occurs. The call

processing node responsible for interpreting the script would map the CPL script to services available, either features (e.g. Email client) or protocol capabilities (e.g. SIP forwarding). Basically, in mapping the script to services, a composition of services is realized. ACCENT/PDL employs a similar approach to service composition; however the mapping between policies and services requires an interpretation step which is not as straightforward as CPL. Here again, we can say that the services are *virtually* composed from the end-user point of view. Moreover, the realization of the composition is *transparent*, which is to say, not apparent to the services themselves.

6.3. Web Services

Web Services enable program to program interaction over the Web. By definition, Web Services are compositions of services using XML-based protocols for information exchange and the Web infrastructure for transport. Web Services offered by a service provider are described in Web Services Description Language (WSDL) [37], and these descriptions are published in a Universal Description, Discovery, and Integration (UDDI) database [34]. The UDDI database may be queried automatically by Web Services interested in binding to other Web Services for interaction. "The Web Services Choreography Working Group is chartered to design a language to compose and describe the relationships between Web Services. This composition is known as *choreography* of Web Services" [37]. Complex interactions between Web Services are envisioned and must be choreographed. For instance, such interactions may involve cascading Web Services, where service providers actually aggregate services provided by others. The language for describing such interactions will likely be at a high level of abstraction such as workflows or processes, rather than rules or policies. The Working Group has yet to publish a working draft. Once again, we point out that services have no *a priori* notion of how they will be composed with others. Interactions will need to be managed externally from the service logic.

7. Conclusion

As shown by Calder et al. [6] and Keck and Kuehn [20], much work has been done to address feature interaction in plain old telephony systems (POTS), however many open issues exist for emerging telecommunication networks.

A growing trend in next-generation networks is to let end-users play a more prominent role in service creation and provisioning. Two simple to use, yet powerful mechanisms have been proposed to let users define services or policies for service behaviour, but many open issues prevent them from being adopted in industrial-scale, *real-world* applications.

The goal of composing reusable services is to create added-value at minimal cost. Approaches to reuse existing IP-based multimedia services as building blocks within frameworks to facilitate composition have shown promise. In general, services do not have prior knowledge of how they will be composed. The compositions must be managed externally in order to provide enhanced overall behaviour. Web Services are becoming the dominant platform for service composition in the future, however languages to express complex compositions of Web Services are yet to be defined.

CHAPTER 4

ENHANCED SERVICE EXECUTION RULE LANGUAGE AND FRAMEWORK

In this chapter, we describe our contributions with respect to the first problem identified in Chapter 1. We have recently published a large part of this chapter and the next in [10]. We discuss our enhancements to SERL to allow for personalized customization of services by end-users who do not have expert knowledge of the services and the environment, while guaranteeing, to a certain degree, the absence of unwanted feature interactions. We assume that any user may attempt to configure any service that they subscribe to, or compose and inter-work several of them for added-value. SERL, as it is defined, allows customization of individual, inter-worked, and composed service behavior according to individual requirements, however it does not define a mechanism to protect network operators or service providers from malicious, erroneous, or otherwise unwanted service usage, nor does it provide feedback to end-users indicating whether their requirements for service behavior will be met. By validating user-defined configurations against constraints imposed by experts, we provide a partial solution to these problems.

1. Language Extensions

As a primary enhancement to SERL, we define two types of SERL Rule Modules: Composition Constraint Rule Modules, and Configuration Rule Modules. Composition Constraints represent expert knowledge about how services may inter-work or be composed. Configuration Rules relate to instances of acceptable compositions of services, and configure the instances with personalized service data. Configuration Rules are expressed using a subset of the language already defined for general SERL rules.

Typically, Composition Constraints are generated by experts with knowledge of the environment, the services deployed in the system, and potential feature interactions. Non-expert end-users will define Configuration Rules. Composition Constraints are modified when new services are added to or removed from the system. They may also be updated if new feature interactions are detected between existing services. On the other hand, each user will manage his or her own Configuration Rules.

In Figure 7, we provide a graphical representation of the relationship between rule types and services for a system with three end-users, Alice, Bob and Charlie. The Composition Constraints are written by an expert. A multitude of compositions that are not in violation of these constraints may exist, hence referred to as *acceptable compositions*. End-users will define service configurations, where each should define instances of one or more acceptable composition along with personalized service data. Configuration of individual services is allowed since each service on its own is actually considered to be an acceptable composition. In Figure 7, Alice and Bob each have their own configuration of the same acceptable composition. A subscriber, like Alice for instance, may have two or more configurations, but only one configuration may be active at a time, and this active configuration must be specified.

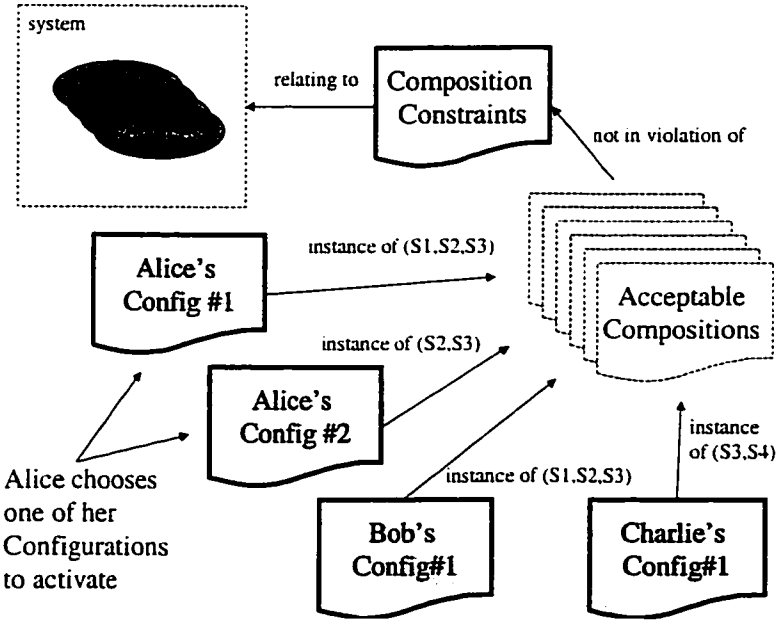

Figure 7: Relationship between Rule Types

2. Composition Constraints

As we have mentioned, Composition Constraints must not be violated when services are composed in user-defined configurations. Since we are considering a SUSC context, it is reasonable to suggest that all possible compositions of services in the system are known *a priori*, which allows offline analysis of services to detect interactions. Moreover, an expert

may use his or her knowledge, based on experience, to facilitate detection. The analysis procedure used by experts to detect interactions is out of the scope of our current work because we believe that it would not be difficult to adapt several possible Feature Interaction detection mechanisms for application here. Following analysis, the expert will have the required information to define constraints for service composition, where the objective of these rules is to force end-users to define configurations that will not cause feature interactions.

We highlight the fact that the set of possible compositions of services in the system may be reduced simply because we are dealing with SERL, involving service invocation according to Processing Points and priorities. Intuitively, this may eliminate some potentially disruptive feature interactions. We define such constraints imposed by SERL as *implicit constraints* because they are intrinsic to the framework.

Composition Constraint Rule Modules express *explicit constraints*. To write these rules, an expert must consider the behavior of services, their input/output data, and implicit constraints of SERL. Explicit constraints extend the service inter-working and composition protocol defined by SERL on a per system basis (i.e. single network component). We require that when services are deployed in the system, service providers will include *deployment descriptors* for each service. Contained within deployment descriptors are items of information about the service that will be needed for feature interaction detection, and subsequently, for writing Composition Constraints. If source code for services being deployed is available, then deployment descriptors may not be necessary.

Explicit constraints are classified into three types: *order-preserving*, *data selection*, and *mutual exclusion*. In the absence of a constraint, services may be invoked in parallel. We need to extend the SERL language to be able to express these types of constraints in Composition Constraint Rule Modules. In addition, we need to be able to describe the service objects that will be invoked by eSERL rules. Such descriptions include the names of the services, as well as their input and output parameters. In [30], the authors express the SERL language using an XML-Document Type Definition. We have taken that original

definition and added to it as shown in Appendix A. In Appendix B, we show the example of a Composition Constraint Rule Module used for the two Case Studies in Chapter 7.

Composition Constraints usually express constraints for service composition and inter-working pair-wise, but triples, quads, etc., are allowed. This does not imply that services (assumed to be atomic) cannot be interleaved, so long as all constraints are satisfied for the duration of an event-flow leading to service invocations at the node, referred to a Cascaded-Chain in [31]. For example, let S1, S2, S3, and S4 be assigned PP1, PP1, PP2, and PP2, respectively, and we ignore priorities. An expert may derive Composition Constraints stating that S1 and S4 are mutually exclusive, and that S1 must precede S2 if they ever run in succession. In addition, they may determine that S3, when invoked after S1, must select one of S1's defined data outputs as input. Many compositions are acceptable, meaning not in violation of implicit or explicit constraints, but it is important to note that a very large number of user-defined configurations may exist due to the personalized service data. Some acceptable compositions are: (S1, S2, S3), (S1, S3), (S2, S3, S4), (S2, S4, S3). Unacceptable compositions include: (S1, S2, S4), (S2, S1), and (S4, S2), among others.

3. Configuration Rules

Configuration Rule Modules contain Configuration Rules, which specify conditions, and actions to carry out when conditions are satisfied. Configuration Rule Modules extend the run-time behavior of the call processing system. As such, a Configuration Rule Module may be seen as a meta-service or more abstract service layer. There is no difference in the syntactic representation of these rules and those already defined by SERL. Existing language constructs allow end-users to write Configuration Rules, and as such, these rule sets would be compatible with SERL processing nodes not implementing the enhancements that we propose for Feature Interaction Management. On the other hand, we must constrain the SERL language slightly in order to allow for validation. Hence, generic SERL Rules Modules would not be compatible with a SERL processing node expecting more specialized Configuration Rule Modules.

A user may have more than one Configuration Rule Module defined; however for simplicity, we consider only one *activated* Configuration Rule Module at any given time. An alternate Configuration Rule Module may replace the active one upon request. Advanced rules to automate activation or deactivation of Configuration Rule Modules depending on user-context (i.e. location, role, membership, task, etc.) are a topic for future work. Also, we assume that the services considered in a Configuration Rule Module are actually in service while the Configuration Rule Module is active. Otherwise, a Configuration Rule Module would not be eligible for activation.

For an example of how a Configuration Rule Module can define a personalized composition of services, let us refer back to *the Interactive Call Screening* service used as an example Parlay/OSA service in Chapter 2. In the example, we saw how the service could be broken into two parts, essentially a Call Forwarding part to forward outgoing calls to a manager or parent who will screen the call, and a second part where the service establishes a call between the original callee and caller. In between the two parts, there was an instant message sent to query the screening party, and a feature instruction to tear down the forwarded call to the screener. Even though we considered ICS to be a single service in the example, if that single unit of functionality were not available as a service in the network, then an end-user could emulate it by composing *Call Forwarding*, *Instant Messaging* and *Application-Initiated Call* services, for instance. They would define conditions and actions in a configuration relating the three services to each other, and assuming that there were no violated composition constraints, the configuration would be deployed into the system. The call processing system would then be have the knowledge required to provide overall behavior similar to ICS. This is an example of low-level service composition. Examples of higher-level service composition using more abstract building blocks to deliver overall behavior with increased complexity are discussed in the Case Studies of Chapter 7.

4. Modified Feature Grouping Criteria

The final enhancement to SERL that we have devised is a modification to the criteria for determining the Feature Group/Processing Point identifiers for services. As defined, SERL provisions for this type of enhancement. Our modifications are summarized as follows:

- For services that may add, delete, or modify any part of the event context (e.g. SIP message, Parlay/OSA event), and are mostly related to routing, we assign Processing Point 1/-1. Such services may modify source, destination, or intermediary nodes identified in the event context, for instance.

- For services that may add, delete, or modify any part of the event context except routing information, and are mostly related to screening, we assign Processing Point 2/-2. Such services require read-only access to routing information, and are only allowed to block a call based on certain screening criteria. They may not re-route calls.

- For services that may add, delete, or modify any part of the event context except routing information, and are mostly related to the event context payload, we assign Processing Point 3/-3. Such services require read-only access to routing information, and may only modify event context payload (e.g. SIP message body).

- For services that may not add, delete, or modify any part of the event context, we assign Processing Point 4/-4. Such services require read-only access to event context.

CHAPTER 5

AUTOMATED DETECTION OF CONSTRAINT VIOLATIONS IN USER-DEFINED RULES

Our approach for Feature Interaction Management hinges on the concept of validating Configuration Rules in order to guarantee the absence of feature interactions. This chapter is concerned with the second of the problems identified in Chapter 1. As explained, providing the guarantee is of utmost importance in order to protect the interests of all stakeholders involved, namely, network operators, service providers, and end-users.

1. Determining Acceptable Compositions

The first requirement for validation is that the set of *acceptable compositions* for services deployed in the system be determined from Composition Constraints. Computing acceptable compositions need only occur once, after having deployed a new service into the system along with Composition Constraints defined by an expert relating the new service to the others. Once acceptable compositions are known, they are stored in the system for future reference.

As we have mentioned, the simplest way to determine acceptable compositions is to enumerate all possible combinations of services deployed in the system, and then eliminate those combinations, which violate constraints. This method exhibits combinatorial expansion problems. An optimization may be obtained by considering a full-mesh graph, where nodes represent services, and links represent combinations of services. For each node, if we enumerate all paths from it to each other one, avoiding the traversal of the same node twice, then we obtain the set of all possible compositions. Now, if we mark links in order to represent invalid compositions (obtained from composition constraints), and only then enumerate compositions while abiding by semantics of the marked links, we obtain the set of acceptable compositions directly while using much less memory and in much less time than the previous *brute-force* method.

1.1. Completeness Assumption

We assume Composition Constraints to be complete. However, in theory, this assumption cannot be guaranteed. We rely on experts to know about possible problematic interactions between services and consider them when defining constraints. As more problematic interactions are discovered though service usage over an extended period of time, Composition Constraints will be updated. Seeing as this is a continuous process, and given that we view the problem from a less formal, more pragmatic standpoint, we require Composition Constraints to approach a *complete set*, but probably never fully satisfy the requirement. Moreover, the process of updating constraints may invalidate previously valid configurations, and solutions for this type of non-monotonic extension of the system should be explored in future work.

The degree of confidence with which one may guarantee the behavior of a service configuration is dependent on the level of completeness of the set of constraints. We have not been able to quantify the degree of completeness of these sets, and therefore, we cannot accurately provide a measure of confidence. We expect that as more empirical results are gathered from more case studies, methods to provide a measure of confidence will be developed.

1.2. Consistency of Composition Constraints

Consistency of Composition Constraints is another issue that we have to take into account. Indeed the set of Composition Constraints may be inconsistent. In the worst case, this will lead to a set of acceptable compositions, where each composition in the set is a service on its own. In other words, these services cannot be composed. In such a case, an expert would intuitively try to detect inconsistencies and relax constraints if possible. More formal approaches to detect inconsistencies have not yet been investigated.

2. Validation of Configuration Rules

Whenever a Configuration Rule Module is created or modified by a user, the rules within the module need to be validated. Validation is possible as long as a set of acceptable compositions is known. Once validated, Configuration Rule Modules are eligible for deployment and activation in the call processing system to be detailed in the next chapter.

The algorithm to validate a user's Configuration Rule Module essentially tries to determine all potential service compositions from the user's set of rules, and then makes sure that each composition is acceptable, or in other words, whether the composition exists in the set of acceptable compositions. The issue requires us to examine the possibility of having separate rules in a configuration which may have triggering conditional expressions satisfied simultaneously, thus causing their actions to be composed. If it is possible for a single event occurrence to satisfy conditions of two or more rules, we say that the rules *overlap* and assume that the services affected by the actions constitute a composition.

2.1. Constraint Violations by Actions of a Single Rule

All Configuration Rules can be expressed in the form *R: if C then A*; where for rule *R*, if conditional expression *C* is satisfied, then action *A* occurs. For a single rule, it is easy to determine whether it violates constraints or not. For example, let us define the following constraints and rules:

```
Constraints:
K1: mutex(S1,S2);
K2: order(S1,S3);
K3: select(S3.in1 := (S1.out1 | S1.out2))

User-Defined Rules:
R1: If C1 then {          R2: If C2 then {
      Invoke (S1)                 Invoke (S1)
      Invoke (S2)                 EventContext.setVal(S3.in1,S1.out2)
}                                 Invoke (S3)
                          }
```

R1 is invalid since no composition (S1, S2) exists in the set of acceptable compositions due to the constraint K1 which states that S1 and S2 must be mutually exclusive for the duration of a call. R2 on the other hand is valid because S1 and S3 are invoked in the appropriate order (K2), and prior to invoking S3, we specify that input S3.in1 be set to S1.out2 in the event context (K3).

2.2. Constraint Violations by Composed Actions

A more complex problem exists when trying to find constraint violations given a set of rules. The main issue here is the possibility of conflicts when combining the actions of several rules. A set of rules, defined as R_i: *if* C_i *then* A_i; where $1 \leq i \leq N$, *may* have actions

A_j and A_k in violation of constraints if $C_j \wedge C_k$ is satisfied. Hence, whenever conditional expressions from different rules overlap, we need to build composite-actions from actions within those rules, and then validate as explained for a single rule. The difficulty, then is to determine whether rules overlap.

This problem has been studied in a different context in [36]. In this paper, the authors were interested in detecting conflicts between rules by determining overlapping conditional expressions and then checking for conflicting actions with well-known semantics. The authors explored what they call "Fault Detection", and use *firewalls* as an example. A typical IP packet filter or firewall will have a set of rules defining what to do with incoming or outgoing packets. Conditions for rules are usually specified as a range of IP addresses and ports, and actions state that packets must either be forwarded or discarded. In this case, the authors provide an algorithm for determining when IP address ranges overlap, and hence determine whether rules overlap. When overlap is detected, actions are checked to determine whether they are conflicting. Since there is no ambiguity as to whether *discard* and *forward* constitute conflicting actions, they have no problem detecting rules which are in conflict with each other.

In our case the semantics of actions are unknown, but constrained by the Composition Constraints. If a set of actions is not forbidden by the Composition Constraints, we say that these actions are *non-conflicting*.

2.3. Determining Whether Rules Overlap

Determining whether conditional expressions overlap has a solution in polynomial time if the set of possible values for variables is discrete, finite, and ordered as shown in [36]. Due to the usage of SERL language constructs in a Parlay/OSA context, this holds, but we also define a general principle to be applied when comparing rules pair-wise.

General Principle

Two rules are said to be overlapping, unless

(a) conditional expressions have at least one common dimension, AND

(b) at least one common variable in the common dimensions, AND

(c) non-overlapping values for the common variables.

Dimensions can be seen as Parlay/OSA APIs, or other discrete, finite, and ordered quantities. We illustrate our approach using a few examples. Note that for each example we assume that we have two rules being compared to determine whether they overlap or not. Conditions for each rule can be encoded in terms of Parlay/OSA API methods or events, but we have abstracted them to simplify our discussion.

Example 1

```
C1 := {"my location is Montreal"}
C2 := {"my location is office"}
```

In this example, let us assume that locations "Montreal" and "office" have World Geodetic System 1984 (WGS84) codes predefined for them. WGS84 is the encoding format used by the Parlay/OSA Mobility API. Using the predefined semantic interpretations for locations "Montreal" and "office" we are able to determine if the locations overlap. If my "office" is in "Montreal" then the actions for these rules are considered to be composed. If this composition is in the set of acceptable compositions, then it is allowed, and the validation process continues for the remainder of the rule pairs in the configuration.

Example 2

```
C1 := {"my location is home"}
C2 := {"caller is bob@school.com"}
```

The dimensions of conditions C1 and C2 in this second example are not equivalent. If Bob calls me while I am at home, then both rules are satisfied simultaneously. Therefore, we automatically consider the rules to be overlapping and validate the composition of their actions.

Example 3

```
C1 := {"my location is school" AND "caller is alice@home.com"}
C2 := {"my location is office" AND "caller is sales@company.com"}
```

In this example, conditional expressions piece together the Mobility API with the Multi-Party Call Control API, but the dimensions of C1 and C2 are equivalent. Assuming that no semantic interpretation for callers is done (i.e. syntactic comparison only, see [38]), and WGS84 codes for locations "school" and "office" do not results an intersection of geographic areas, we can determine that C1 and C2 cannot be simultaneously satisfied. Hence, the actions for these rules do not overlap, and therefore there actions will never occur simultaneously.

Example 4

```
C1 := {"time is (11:00 to 14:00)" OR "caller is alice@home.com"}
C2 := {"time is (09:00 to 10:00)" OR "caller is sales@company.com"}
```

Here, conditional expressions are based on time and the Multi-Party Call Control API, but we use OR operators to compose dimensions. In this case, it is possible for Alice to call me at 09:30, thus satisfying C1 and C2 simultaneously. Therefore, we must consider C1 and C2 as overlapping, and compose the rule actions.

Example 5

```
// Assume that rSet holds result of the database query:
//      "SELECT * FROM Contacts WHERE SupportsVideo=true"
C1 := {"caller terminal capabilities support video" OR  "caller in rSet"}
C2 := not C1
```

The only way to ensure non-overlapping OR composition of conditional expressions, or dynamically defined conditions (e.g. database query) is to use binary conditional expressions. Here C1 is based on Terminal Capabilities API, and a record set obtained by querying a database at runtime. Even though we do not know what the records in the record set will be at runtime (not statically definable), we are sure that C1 and C2 will never be satisfied simultaneously.

Example 6

```
C1:= {"outgoing call"} // priority 10, action triggers S1
C2:= {"callee is busy" AND
       NOT "Session.isTriggered(S1)"} // priority 5, action triggers S2
```

In this example, one would normally expect "outgoing call" and "callee is busy" to be separate events. In traditional networks, this is likely true. However, in next-generation

networks, it is possible for the event context to have been populated with information about the callee status at some point along the signalling path. When the eSERL node eventually receives the event, the event context would contain both items of information simultaneously. Our approach still applies regardless of whether single or multiple events occur. In this sixth example, assume that first rule would trigger S1 and the second would trigger S2. Also, the first rule has higher priority than the second. Neglecting the *Session* object in C2 for a moment, the two rules would overlap because C1 and C2 refer to different event context variables in the same dimension (i.e. Multi-Party Call Control API). If a mutual exclusion constraint existed for S1 and S2, then these rules together would violate it. To remedy the situation, we introduce the *Session* object which is part of the eSERL processing environment. It keeps track of triggered services, among other properties. By using it as shown in the example along with priorities defining an order of invocation, we can guarantee that the two rules can never trigger together in violation of the constraint.

As a final note, we point out that nesting of rules is allowed. Such nested rules need to be normalized as will be explained later. Moreover, we also allow for *if (C_i) { ... } else { ... }* structures to enable a more convenient way of expressing conditions like those in *Example 5*. The examples discussed above are quite simple. In our Case Studies of Chapter 7, we present more complex examples and the results of running our rule validation algorithms on them.

3. Validation Algorithm

All eSERL rule modules are encoded in XML according to the eSERL Document Type Definition (DTD) in Appendix A. In this section we explain the algorithms for Configuration Rule Module Validation

Step 1: Normalize all Configuration rules.

Each rule is re-written in the form *if (C) then A*, where *A* is one, and *only one* action, and *C* is an aggregated conditional expression made up from each conditional expression encountered in a depth-first search through the XML document tree, starting from the document root node and ending at the action leaf node. By normalizing, we are able to

handle nested rules, as we would any other. It is important to keep track of Processing Points and priorities of rules in the data structures used to store rules after normalization. Processing Points and priorities are used for determining whether order constraints are violated in Step 2.

Step 2: By default, assume that a Configuration Rule Module is valid. Then do:

```
1 For rule1, where rule1 is a Configuration Rule
2    For rule2, where rule2 is a Configuration Rule and not rule1
3*       If rule1.condition and rule2.condition overlap then
4            If rule1.action composed with rule2.action is
                                not in set of acceptable compositions then
5                Configuration Rule Module is invalid.
6            End if
7        End if
8    End for
9 End for
```

In the above algorithm, we compare rules pair-wise. We check conditional expressions for overlap as explained in the previous section, and if overlap is found, then we compose the actions and check if the composition exists in the set of acceptable compositions.

Step 2.1: Service Ordering

When composing the rule actions, order needs to be considered. If two rules have processing-points with different event-flow directions then they do not form a composition and each, individually, is an acceptable composition. Ordering of actions when forming a composition follows this scheme:

```
1 If ((processing point of rule2 < processing point of rule1) OR
      ((processing point of rule2 == processing point of rule1) AND
       (priority of rule2 > priority of rule1))) THEN
2    rule2's action before rule1's action
3 Else
     rule1's action before rule2's action
4 End if
```

By default, rule1 is ordered before rule2. Now if the processing point of rule 2 is less than the processing point of rule 1, the order is reversed. If they have the same processing point, then the higher priority rule has precedence.

Further Considerations for Service Ordering

If the two rule actions invoke the same service (possibly based on completely different conditional expressions), we consider the composition of the service with itself as acceptable. In other words, we accept invocation of the same service twice as long as there is an order specified (due to possibly of different data for each invocation).

Step 3: Check Data Constraints

In a final step, we check for data constraint violations. This simply requires us to make sure that the most recent action statements existing before service invocation action statements in the XML document structure set service variables to appropriate values, as required by constraints. No further checks are done. For a simple example of this, refer back to the action statements of the second rule, R2, in the example of Section 2.1 of this chapter.

CHAPTER 6

DESIGN AND IMPLEMENTATION OF OUR FRAMEWORK IN PARLAY/OSA

In order to actually realize the approach discussed in Chapters 4 and 5, we require an implementation context. We have selected Parlay/OSA and justified the selection in Chapter 1. In this chapter we address the last of the three problems considered in our thesis, namely our design and implementation of a *proof of concept* prototype.

1. Overall Architecture

Our approach requires an architecture employing one or more Feature Interaction Managers (FIM), which act as mediators controlling the triggering and execution of features. The behavior of a FIM in our context differs from the commonly understood behavior(s) documented in [6]. Each of our FIMs *virtually* composes services according to user-defined requirements (i.e. rules). In other words, each FIM manages the events that affect the behavior of deployed services which may lead to the overall appearance of composed service behavior from the user point of view. SERL is a technology that essentially implements a SUSC subset of the generalized model. Moreover, since Parlay/OSA defines an architecture with clearly specified points of control (i.e. application servers, service capability servers), we intuitively position the FIM(s) in a Parlay/OSA framework as shown in Figure 8.

FIMs must be able to intercept events in order to apply resolution rules. In a distributed system with multiple application servers, it is unrealistic to have a single, centralized FIM, since a global view is difficult to obtain and the FIM would most definitely become the system bottleneck. A distributed approach, where multiple FIMs communicate, is a much better solution, albeit more complex to design. We have initially considered the positioning of one FIM in the Parlay/OSA architecture, and have provisioned for application in a multi-FIM environment in future work.

Our positioning of the FIM takes advantage of the standardized Parlay/OSA APIs, and the Half-Object Plus Protocol design pattern [26]. The FIM is inserted between services and the actual Parlay/OSA client side interface implementation.

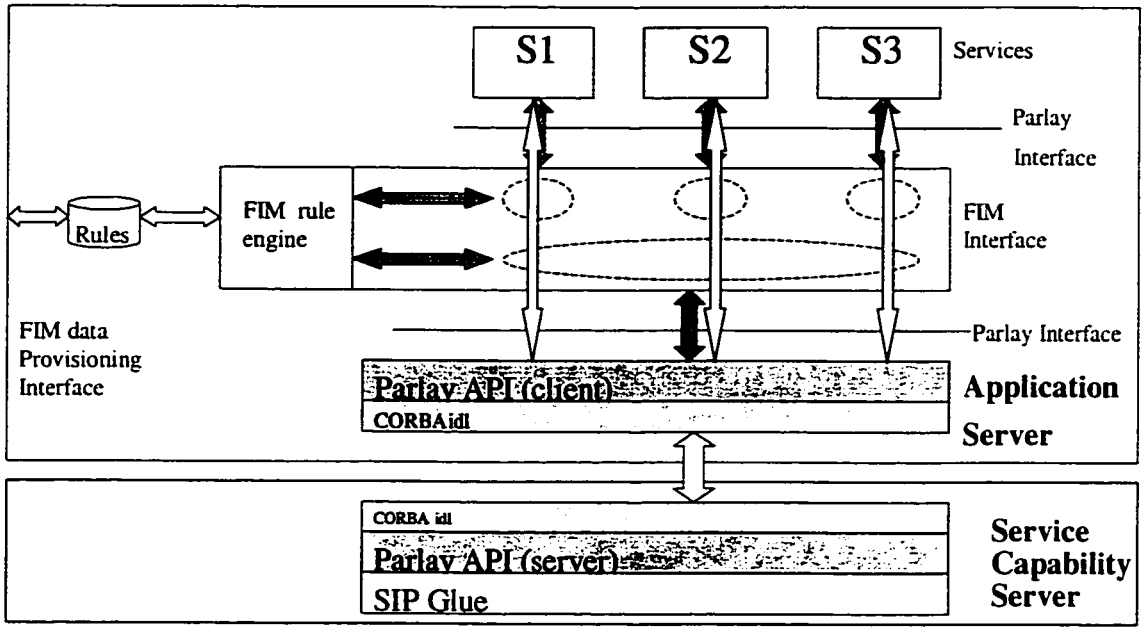

Figure 8. A FIM for one Application Server

The FIM offers a Parlay/OSA API interface to services and it uses the actual APIs provided by the Application Server and the Service Capability Server to implement the interface offered to services. The idea is to enable transparency of FIM behavior from the service point-of-view. A service developer should be able to develop a Parlay/OSA service and run it on an Application Server, whether a FIM is present or not. With the FIM positioned as such, it is able to intercept all events to or from services that are deployed on the Application Server. We require each FIM in the architecture to implement eSERL, as we have defined it in Chapters 4 and 5.

In Figure 9, we show an example of how a FIM virtually composes services according to Configuration rules. The FIM intercepts, and may modify, duplicate, filter, or simply forward any message to or from services. In the example, the FIM composes a Call Forwarding (CF) service and an Information Delivery (ID) service. All calls directed to

Bob are forwarded to Charlie. Whenever a call is forwarded, an instant message is delivered to Bob to notify him.

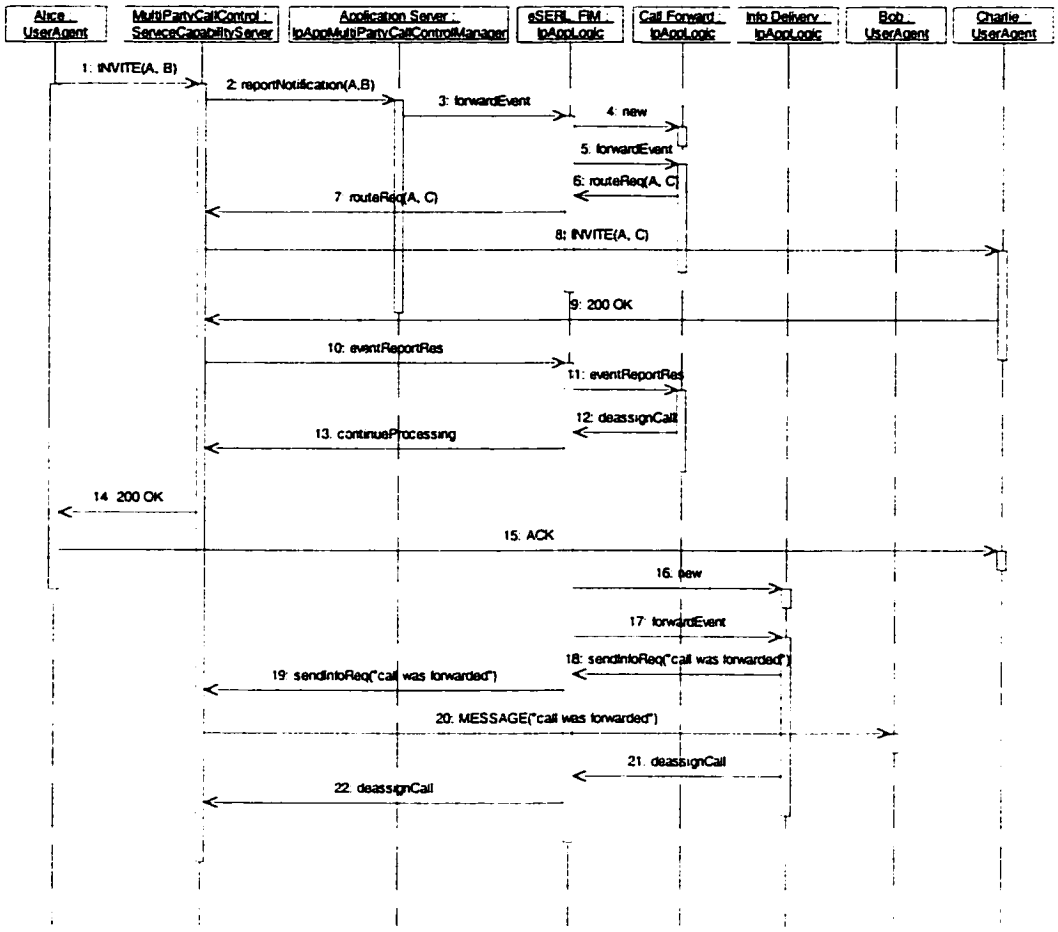

Figure 9: Composed Services Example

Notice that for every event received by the FIM, a lookup for matching rules is performed. Also, recognize that the FIM has overridden the deassignCall() (interaction #12) event generated by the CF service, and also generated a new event to forward to the ID service (interaction #17). All interactions are explained in the Table 5.

Table 5: Composed Service Example: Messages Exchanged

1: INVITE(A,B)	Alice invites Bob to a call session. The message is sent directly to the Service Capability Server.
2: reportNotification(A,B)	The SCS notifies the IpAppMultiPartyCallControlManager interface which is implemented by the Application server.
3: forwardEvent()	Only the eSERL-FIM has registered with the Application Server. Other services register with the eSERL-FIM (registrations not shown). Therefore, the Application Server calls back the FIM.
4: new	The FIM looks up rules for Alice. None are found. Subsequently, it looks up rules for Bob. Bob has defined a rule to trigger the Call Forwarding service.
5: forwardEvent()	The call notification data is forwarded to the CF service.
6: routeReq(A,C)	The service changes the destination address to Charlie.
7: routeReq(A,C)	The FIM looks up rules for Alice and Charlie. None are found. It forwards the message to the SCS.
8: INVITE(A, C)	The SCS maps the API method to a SIP INVITE message and sends it to Charlie.
9: 200 OK	Charlie answers.
10: eventReportRes	This call leg event is sent to the eSERL-FIM.
11: eventReportRes	No rules match. The FIM forwards the event to the CF service.
12: deassignCall	CF has completed and no longer needs to monitor the call.
13: continueProcessing	The FIM recognizes that CF has no longer needs to monitor the call. Upon discovering the change in state of the CF service, or in other words, that the call has successfully been forwarded for Bob, the FIM takes the first of two actions, namely, to let any administrative processing continue.
14: 200 OK	The SCS lets Alice know that the call has been answered by Charlie.
15: ACK	Alice acknowledges the call session with Charlie, and a full-duplex RTP media stream is established between endpoints (not shown).
16: new	After having instructed the SCS to continue processing, and knowing that the call has been forwarded for Bob (CF has run to completion without error), the FIM looks up rules matching this event. It determines that it needs to trigger ID.
17: forwardEvent()	It encapsulates the event into a message and forwards it to ID.
18: sendInfoReq("call was	The ID service sends a message to the inform Bob that the call has

forwarded")	been forwarded.
19: sendInfoReq("call was forwarded")	The FIM looks for matching rules for this event and finds none. It forwards the event to the SCS.
20: MESSAGE("call was forwarded")	A SIP MESSAGE is sent to Bob. INFO is only used for *in-band* signaling once a session has been established between endpoints.
21: deassignCall	The ID service is no longer interested in controlling the call.
22: deassignCall	The FIM recognizes that the ID service has run to completion. No subsequent services need to be triggered. It forwards this message to the SCS.

In the remainder of this chapter we discuss some of the salient features of the Feature Interaction Manager implementation.

2. Relationship between SCS, AS, and eSERL-FIM

The Application Server contains the IpAppMultiPartyCallControlManagerImpl object, but requires all services to provide their own implementations of the remainder of the API callback objects and, of course, their respective service logic (i.e. IpAppLogic). Our eSERL FIM is actually implemented as a service which runs as a daemon service on the Application Server. The Application Server "knows about" the FIM because it is the only service that has registered with it, and therefore delegates all service invocation and execution management to the FIM service. All other services register with the FIM.

3. Absence of Matching Rules for Events Received

If events are received from the network which do not have rules defined for their handling, these events are broadcast to all running services for the users identified in the event context. In the example of Figure 9, only one service was interested in receiving events at any point in time, and therefore no broadcasting is shown. Similarly, for events originating from services, if no rules are matched, they just pass through the FIM uninterrupted on their way to the network.

If an event is received from the network and no services exist for the users involved in the event, then a Basic Call Handler service is invoked. A typical example of this is a user

without any service subscriptions who just wishes to make a call. This Basic Call Handler service is actually the base class for all services we have implemented.

4. Session and Proxy Objects

Of fundamental importance is the mechanism to allow events generated from the network or from services to be dispatched to the FIM engine for processing. In order to do this while ensuring *transparency* of FIM operation from the point of view of the services, we had to introduce *proxy* objects and a *call session container* for these objects. According to the Parlay/OSA specifications, services need to register for events with SCS and subsequently take action when notified of their occurrence though callbacks.

If services were allowed to register directly with the SCS in our architecture, the callbacks would bypass the FIM completely. Similarly, if services were allowed to perform actions on call objects in the SCS directly, they would again bypass the FIM.

We introduce proxy objects within the FIM which implement appropriate Parlay/OSA interfaces. The implementation of these objects simply forwards events received towards the FIM processing engine. The FIM then calls back *real* objects when applying resolutions. Each API object has a corresponding pair of proxy objects in the FIM in order to enable event dispatching to services. Figure 10 shows the objects in question for our implementation of the Parlay/OSA Multi-Party Call Control API. We have similar object implementations for User Interaction and Mobility APIs.

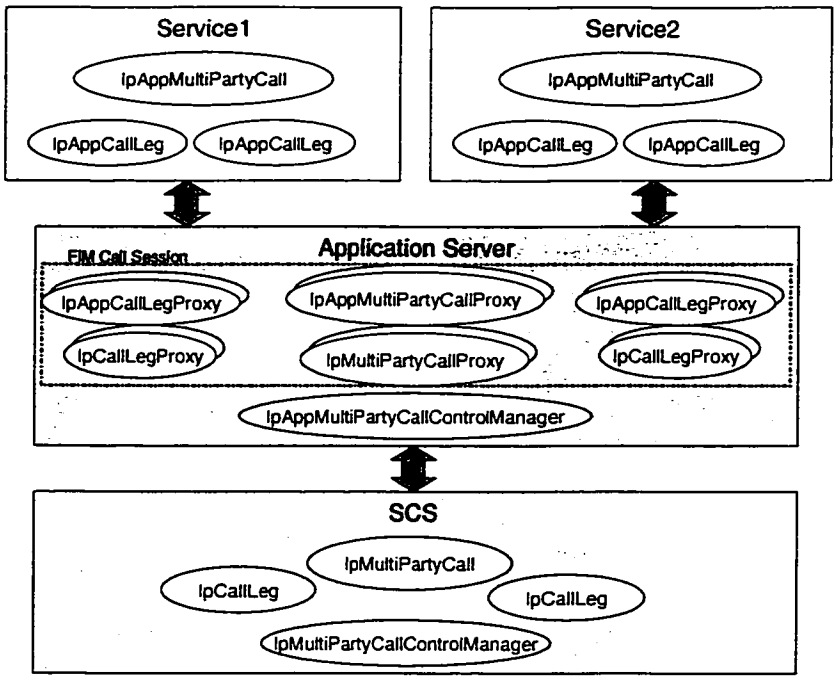

Figure 10: Session and Proxy Objects for Call Control

The function of the session object is to facilitate management of all the proxy objects in the FIM and to keep track of the call session state and the state of services. A new session is created for every new call notification generated by the SCS. Note that there is a difference between a new call notification and a typical call leg event.

5. Event Translation and Synchronous Method Simulation

As we mentioned in the previous section, the function of the proxy objects is to forward events that are received from the network to the FIM engine for processing. The same must be done for events received from services, but the complication is that events from services are actually *method invocations* on proxy objects rather than events per se. We are required to translate method invocations to event objects and subsequently forward them to the engine.

In certain cases, the method invocations are *synchronous* meaning that a response value must immediately be returned to the service which invoked the method. We therefore simulate synchronous behavior with our asynchronous event dispatching mechanism by

setting a flag in the event object. Upon receiving an event with this flag set, the engine will treat this event with higher priority. In addition, the engine calls back the proxy object which set the flag. This is not the normal channel for event flow as shown in Figure 11.

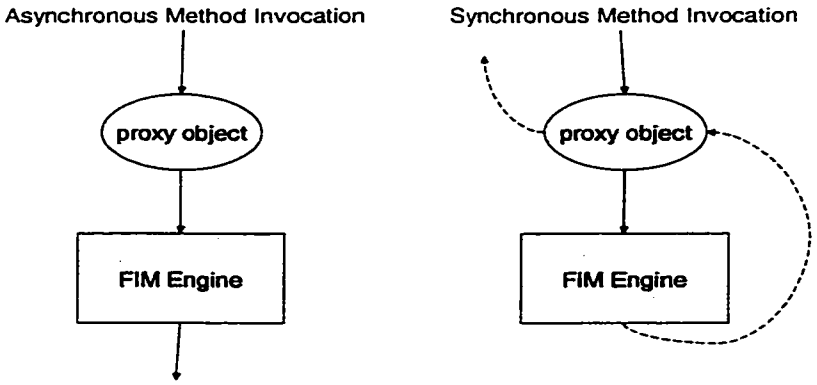

Figure 11: Asynchronous vs. Synchronous Method Invocation

6. Service Discovery and Callback Registration

Parlay/OSA depends on either CORBA or DCOM to allow the SCS and Application Servers to communicate is a distributed environment. The SCS is assumed to have minimal downtime, while the same cannot be said for all Application Servers. Upon initialization, Application Servers need to discover Service Capability Servers. This may be achieved using the service discovery mechanism provided by CORBA for example.

Services, not having any notion of a FIM in the architecture, are invoked in response to events. Our FIM is actually a service invoked at Application Server initialization time (as a daemon), and we delegate all service invocation and event processing to the FIM in order to ensure that the FIM remains part of the signaling path throughout the service lifetime.

If we wanted to offer basic service rather than premium service as provided by a FIM, all we would have to do is let the Application Server be responsible for invoking services for

basic service subscribers. We have not implemented this, and have rather focused on delegation to the FIM for premium quality of service.

When invoking a new service, in order to force that service to eventually callback proxy objects rather than SCS objects, references to the proxy objects are inserted in place of references to SCS objects in the event messages forwarded to the new service. SCS object references are then stored in the session. In this way, the services are *fooled* into believing that they will communicate with the SCS directly when they actually communicate with the FIM through proxies. This is a critical operation to ensure transparency.

7. FIM Management

We have constructed an Application Server Manager which communicates with the Application Server over CORBA. Using this manager we are able to deploy new services into the Application Server and update Composition Rules. We also create end-user subscriptions to services and deploy Configuration Rules for users. Typically the deployment of Configuration rules should be done by users through a Web Application, but as a proof of concept this interface is adequate.

8. Rule Matching Performance

The most time consuming part of event processing at the FIM is rule matching. For every event that is received from the network, and for every feature instruction generated by a service, the rule matching operation must run. Improving the rule matching algorithms, which are basically string comparisons, would greatly improve overall performance. We have not been able to improve the algorithms for rule matching, however we do expect that if users minimize conditional expressions in rules by ignoring those conditions which are superfluous we will see important performance gains.

CHAPTER 7

CASE STUDIES

In this chapter, we present two Case Studies as *proof of concept*. The first scenario involves service composition and personalization for a sales agent at a company, while the second is for Julie Jones when driving the family car. In both cases the same set of services and composition constraints are used, but the user-defined service configurations are very different. We begin our discussion by describing the test architecture and the services to be used.

1. Test Architecture and Service Benchmark

Our test architecture is developed for a local-area IP network that supports SIP as a signalling protocol. Ericsson Research Canada provided us with a Parlay/OSA gateway which maps the Call Control and User Interaction APIs to SIP and vice versa. We have built a Parlay/OSA Application Server which hosts services and communicates with the gateway over CORBA. Additionally we have build a Mobility Server and a tool to simulate user mobility. Ericsson also provided us with SIP clients, which we were able to extend with additional functionality, such as Instant Messaging. We show our test architecture in Figure 12.

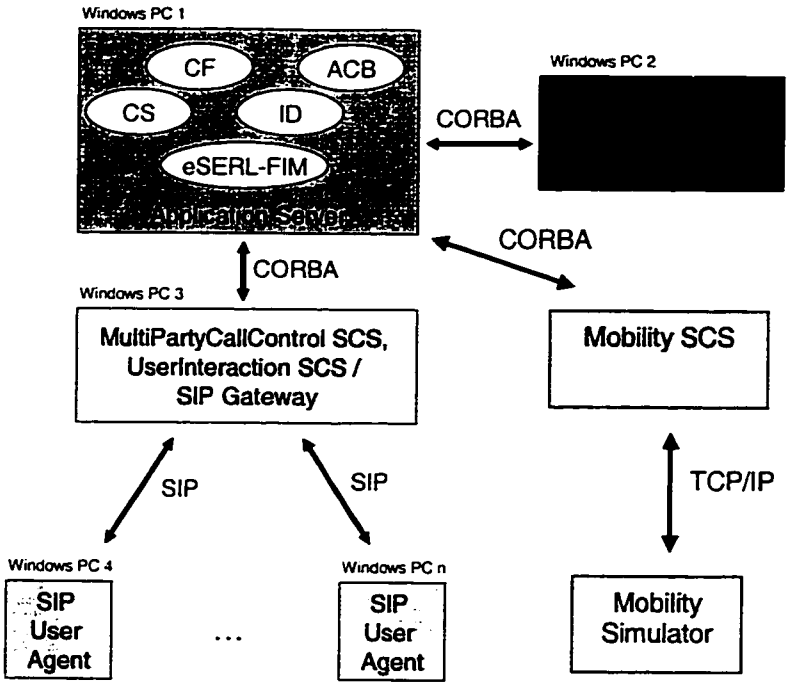

Figure 12: Test Architecture

1.1. Enhancements Required for Our Case Studies

As we have mentioned, the SCS, and the SIP User Agents were provided by Ericsson. All of the other components were implemented from scratch. We take this opportunity to mention some of the enhancements that were required to the Ericsson software in order for our Case Studies to be conducted.

The SIP User Agents consisted of a user interface over a SIP stack. The user interface did not initially support Instant Messaging though. The SIP stack did however support INFO message in-band session signalling. Rather than redesigning the whole User Interface to support Instant Messaging, we simply implemented a secondary User Interface module for Instant Messaging that could be activated along with the primary. We did this by installing *callbacks* between the stack and the primary User Interface. We used INFO messages for Instant Messaging, and so any INFO messages received would be sent to the secondary interface rather than being discarded altogether. This was an *ugly* way to proceed, but it worked adequately.

The SCS was initially designed as a prototype for *service initiated* call control. This needed to be enhanced for call session monitoring and control of session instances which were initiated by end-points (i.e. SIP User Agents). The SCS needed to be able to intercept events and subsequently trigger services which would be allowed to control the call. In implementing the enhancement, we were careful to not break existing functionality for service initiated call control. The resulting behaviour of the SCS is to create a IpCallLeg and IpMultiPartyCall for incoming SIP-INVITE messages. After creating the objects, reportNotification() calls back registered services. Services then have control of the incoming leg, and may create one or more outgoing legs if necessary. Sharing control of call objects in the SCS is achieved through the FIM.

One final consideration for the enhancement was to force SIP User Agents to forward all outgoing messages through the SCS rather than directly to end-points. This was accomplished by setting a parameter in the User Agent configuration instructing it to forward messages through a proxy, which was actually not a proxy, but the SCS.

2. Service Benchmark

We have implemented four services to run in our Parlay/OSA Application Server. These services are quite simple when considered individually, however, when composed, they can provide a great deal of value for end-users, as we will show. Some of the services provide functionality already available in SIP, but at the Parlay-level, the underlying protocols should not be considered. The four services that we developed are Call Forwarding, Call Screening, Information Delivery, and Auto-Callback. We have already introduced some of the services previously in this thesis, but we explain each in more detail below.

- *Call Forwarding* (CF), simply changes the destination address of a call. Triggering of this service may occur for any event the user requires. Such triggering rules shall be stored in Configuration Rules for the user. This service is a terminating service only, while all services that follow can be considered terminating or originating depending on their configuration.

- *Call Screening* (CS) blocks a call if screening criteria is not met. Triggering conditions and screening criteria for the service are specified in Configuration Rules. Screening criteria may be the SIP address of a 3rd party, signifying that an Instant Message should be sent to the 3rd party asking for permission for the call to proceed.

- *Information Delivery* (ID) uses Instant Messaging capabilities of the network to deliver information provided by the participants, or about the environment, to specified end-users. It is also used for text-based request/response interactions between the system and the user(s). Information may be Caller-ID, menu-of-the-day when calling a restaurant, or more. Triggering conditions and information to deliver are stored in Configuration Rules.

- *Auto-Callback* (ACB) allows a participant to save a call that he or she missed or was unable to establish (as caller or callee), and retry at a later time automatically. Two types of triggering rules must be specified: when to save the call, and when to call back.

3. Composition Constraints for the System

After analysis of the deployment descriptors that accompany the four services deployed in the system, CF, CS, ID, and ACB are assigned Processing-Points +/-1, +/-2, +/-2, and +/-1 respectively. They are assigned positive and negative numbers because they may be invoked downstream or upstream depending on the user-defined configurations. The assignment of processing points as such constrains ID or CS to never be allowed to run before ACB or CF for a single-direction event flow through a node. An expert will also determine the following constraints.

- ACB and CF are mutually exclusive, and cannot be triggered on the same event.

- To run ID after ACB or CF, the name of the party who's information must be delivered must be specified before invocation.

- To run CS after ACB or CF, the name of the party to screen must be specified before invocation, along with screening criteria.

- Finally, even though CS and ID have the same processing point, ID must only run after the call has been screened.

All of the constraints defined above are encoded in Composition Constraints provided in Appendix B.

4. Sales Agent at a Call Centre

In this Case Study, a sales agent at a company subscribes to all four services. He wants to configure the services so that clients waste no time on hold when he is busy. He also wants calls from familiar callers to be handled differently.

4.1. Requirements

If the sales agent receives an incoming call, and the call is from a familiar person (i.e. address found in a personal contact list), then call handling exhibits different behaviour than for a stranger, possibly a new customer.

Handling Calls from Familiar Persons

If the sales agent is ever busy when a familiar caller tries to call, then ACB should be invoked. The call should be saved and the caller notified that he or she will be called back later with ID.

When the agent eventually becomes available, ACB should attempt to setup a connection between the sales agent and the original caller as expected. No further requirements for incoming call handling for familiar callers are specified.

Handling Calls from Potential New Clients

As for strangers who are possibly new clients, the sales agent should deliver information about himself and perhaps his products to the caller before the connection is established. Therefore ID should be invoked.

The caller may cancel the call after reading information about the sales agent, but if not then processing continues.

If the sales agent was originally busy when the call was received, then CF should be invoked to forward the call to another sales agent before delivering any callee information with ID.

If the second agent is available, information about her should be delivered instead.

Once again the caller should have the option to cancel the call before connection establishment.

If the second sales agent is also happens to be busy, ACB should be invoked, and the caller notified with ID that the he or she will be called back when the first sales agent become available again. There is no reason why the customer should ever have to be on hold for an indefinite amount of time.

Handling Outgoing Calls

Finally, for all outgoing calls made by the sales agent, or on his behalf (i.e. ACB call retry), the call must be approved by his manager if it is long-distance. In order to request approval automatically, CS should be invoked.

4.2. Composed Service Behaviour

For this case study, we provide a series of sequence diagrams showing the overall composed service behaviour. Our objective is to show how added-value is created by composing very simple, some may say *traditional* services. In the next Case Study we instead focus on Configuration Rules and the validation process.

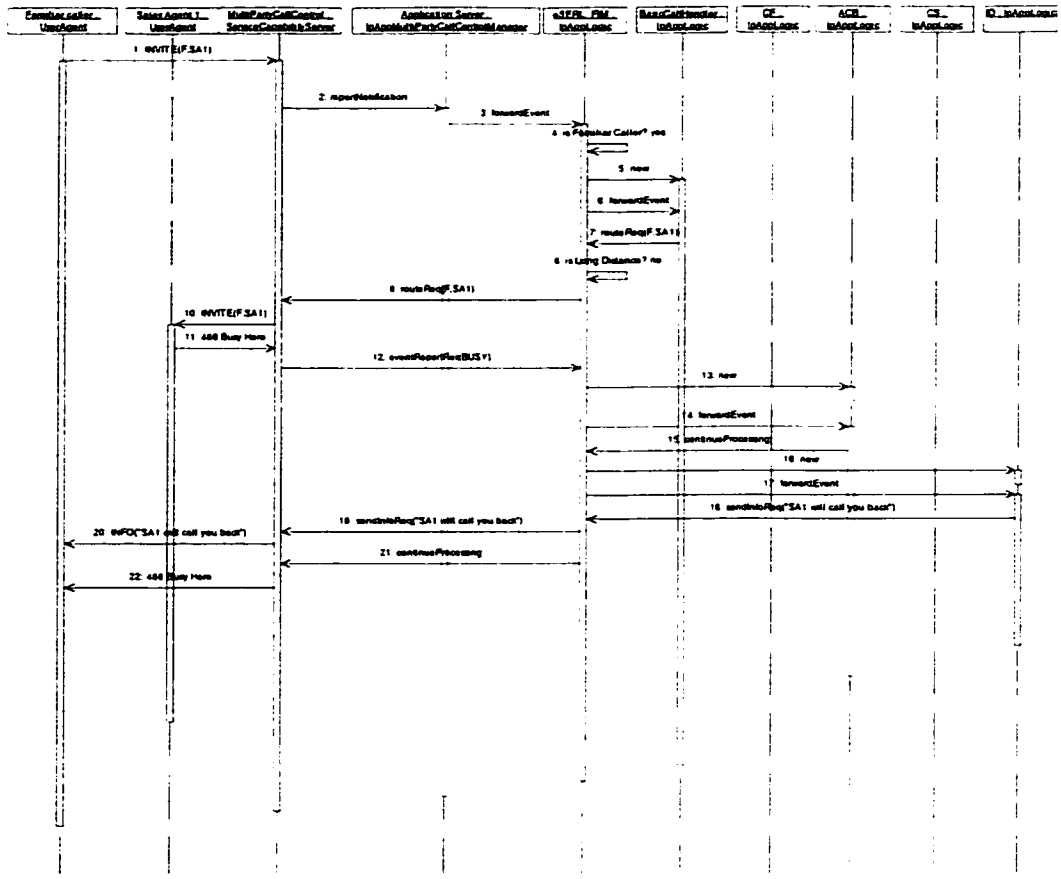

Figure 13: Familiar Caller Calls Busy Sales Agent

In Figure 13, a familiar caller calls the sales agent. The sales agent happens to be busy, so ACB and ID are invoked.

Table 6: Familiar Caller Calls Busy Sales Agent

1: INVITE (F,SA1)	Familiar caller calls sales agent, through the SCS.
2: reportNotification	Call request notification is sent to App Server.
3: forwardEvent	Event forwarded to FIM.
4: is familiar caller? Yes	Up until this point we are not sure if caller is in contact list or not. We search our database of contacts to determine that it is indeed a familiar caller. Since it is a familiar caller, we handle the call differently than for potential new clients.
5: new	Basic Call Handler invoked to process call using default behavior.

6: forwardEvent	Original event forwarded to Call handler.
7: routeReq(F,SA1)	Routing the call from familiar caller to sales agent 1
8: is Long Distance? no	Not a long distance call... no need to invoke CS.
9: routeReq(F, SA1)	Routing the call from familiar caller to sales agent 1
10: INVITE(F,SA1)	Translation from Parlay method to SIP message,
11: 486 Busy Here	Sales agent is busy.
12: eventReportReq	Sales agent status reported to the FIM
13: new	Invoke ACB
14: forwardEvent	Forward data to ACB about the call and event. ACB proceeds to save the caller, callee info for future call reestablishment.
15: continueProcessing	Once all info is saved, ACB lets call proceed.
16: new	FIM invokes ID
17: forwardEvent	Forwards data to ID about the call and event along with the info message to send, as defined in the Configuration Rule.
18: sendInfoReq("SA1 will call you back")	ID informs the caller that sales agent will call back later.
19: sendInfoReq("SA1 will call you back")	FIM simply passes this message on.
20: INFO("SA1 will call you back")	Translation to SIP.
21: continueProcessing	This event was delayed from when ACB invoked it before. Since no services need to run, it may pass it onward.
22: 486 Busy Here	The SCS determines that to continue processing, it simply needs to pass on the last SIP message received. It informs the caller that the sales agent is busy.

In Figure 14 below, the sales agent eventually becomes available again. On this event occurrence, ACB calls the familiar caller back on the sales agent's behalf.

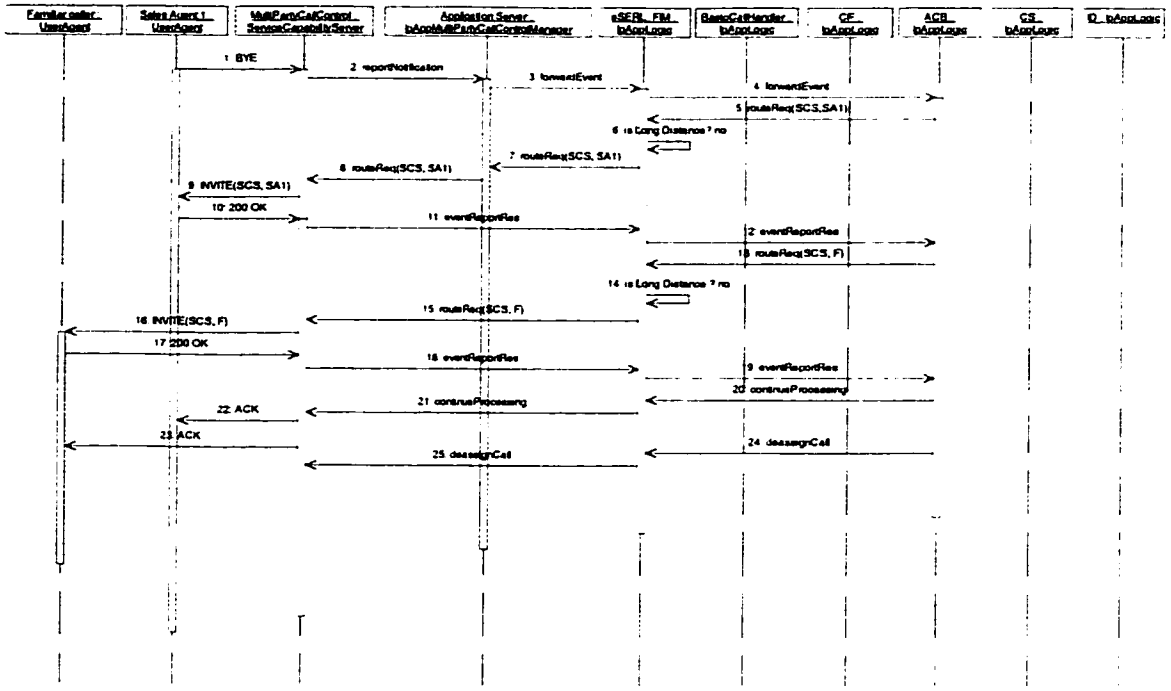

Figure 14: ACB calls Familiar Caller Back

We aggregate some of the explanations of interactions in the table below.

Table 7: ACB Calls Familiar Caller Back

1: BYE	The sales agent terminates his previous call.
2: reportNotification	Since BYE is a SIP request, if no call session exists for it, it may be transalated to a new reportNotification
3: forwardEvent	Event forwarded to the FIM.
4: forwardEvent	The FIM calls back ACB.
5 to 25:	ACB calls back both parties on the sales agent's behalf. The behaviour is the same as for the App Initiated call control part of the Parlay/OSA service example of Chapter 2. The only difference is that when the FIM receives an outgoing routeReq() on the sales agent's behalf, it check to see whether a long distance call is being made due to a rule for invoking CS to screen long distance calls through manager.

Now if the original caller was not a familiar caller, as determined by step 4 in Figure 13, then ID would have had to have been invoked after the Basic Call Handler had called the sales agent and determined that he was not busy. This is shown in Figure 15.

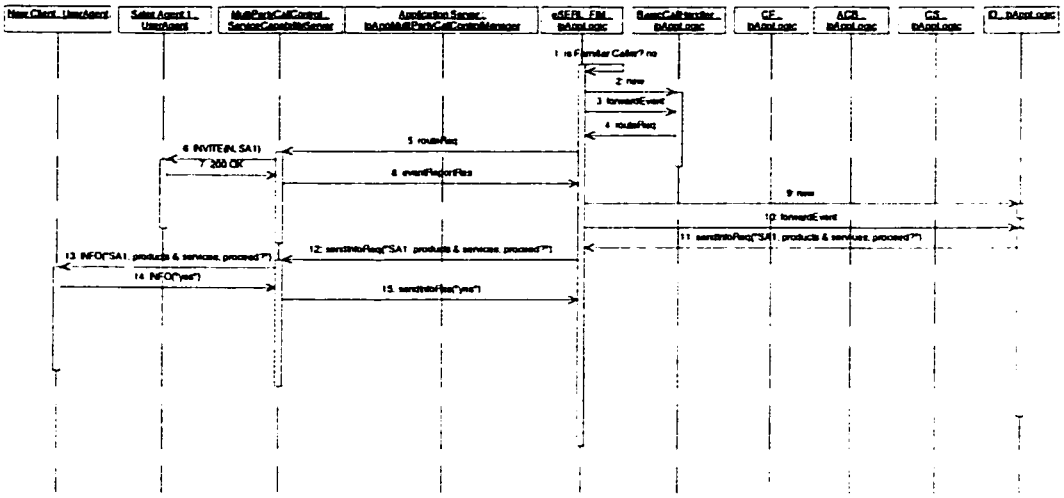

Figure 15: Info Delivery to New Client

We do not provide a table explaining interactions. It is clear that information about sales agent 1 is delivered to the caller and the call proceeded normally after that. If the caller would have rejected the call, it would have been disconnected.

In Figure 16, the call from the new client was made while the first sales agent was busy. Therefore, as a slight variation to what is shown above, the call is forwarded to sales agent 2, and her information is delivered instead.

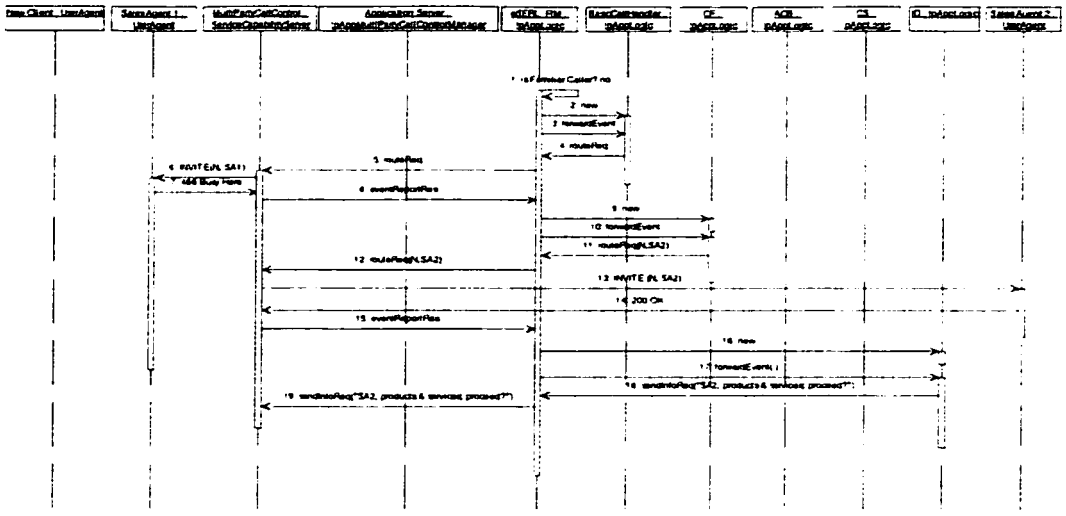

Figure 16: New Client Forwarded to Sales Agent 2

If the second sales agent had been busy, then ACB and ID would have been invoked and further call handling would mimic the behavior for calling back familiar callers as in Figure 14. Even though CF had run before during the session, since it had run to completion, there would be no mutual exclusion constraint violation. To guarantee this during validation, a condition is to be specified in the rule for triggering ACB.

For all outgoing calls made by the sales agent or on the sales agent's behalf by ACB, the calls need to the screened by his manager. This behavior is exactly what is show in Figures 3 and 4, except that triggering of CS would depend on the result of event analysis as performed by the eSERL FIM (see steps 6 and 14 in Figure 14, for instance) rather than unconditionally.

4.3. Result

When the sales agent uploads his service configuration into the system, the Configuration Rule validation algorithm checks the rules. After passing the validation step, the rules are deployed into the FIM. Henceforth, calls to or from the sales agent are processed according to his personalized rules, providing overall behaviour as shown in the previous sequence diagrams.

5. The Jones Family Car

This scenario involves personalized service management for Julie Jones when driving the family car. The Jones Family consists of Julie, Mom, Dad, and of course, the family car. Julie has just earned her driver's license, and is a probationary driver. Each family member owns a mobile phone, and carries it with them everywhere they go. The Jones Family Car is no ordinary car; it has advanced communication functionality built-in – certain ones due to governmental regulations. Requirements for this scenario require an analysis step to figure out which services may be composed and configured to meet requirements.

5.1. Requirements

Three entities have requirements that must be satisfied by Julie's Configuration Rules – the government, the Jones Family, and Julie.

Governmental Requirements

Governmental safety regulations require drivers to "plug-in" their mobile phones, thus identifying themselves as drivers when behind the wheel, and providing a mobile communication terminal for the car's onboard computer.

Furthermore, they require that all calls be disconnected or blocked when made to or from the driver of car when the car speed is greater than 60 km/h and traffic density is medium, or when the car speed is greater than 40 km/h and traffic density is high.

Jones Family Requirements

Mom requires certain functionality for all members of the Jones Family; namely, if the driver is a target or destination in a call attempt, then other participants should be informed that communication may be cut-off due to government regulations as previously stated.

Upon receiving this notification, the other participants should be given the chance to reject the call before establishment.

If the call is approved and established after all the other participants receive the notification about potential disconnection, then if ever the call is cut-off later on, the system will

automatically attempt to reestablish the call when the driver becomes available again (for a sustained period).

Julie's Requirements

Mom also defines requirements for Julie in particular. If Julie drives more than 100 km away from home, an Instant Message must be sent to Mom to let her know.

5.2. Assumptions

We assume that Julie's configuration will extend the configuration defined for her family, which in turn, will extend the configuration defined by the government for all drivers in the land. The online tool used by Mom when writing configuration rules for Julie could enforce this hierarchy. This may be accomplished using Configuration Rule Module templates, where certain rules are defined, but cannot be deleted or neglected when the templates are filled-in and extended. Mom uses a template for drivers, defined with governmentally imposed rules. She extends it with new rules for drivers in the Jones Family. Finally, she extends this one with rules specifically for Julie, and fills-in fields for identifying Julie Jones as the driver.

5.3. Analysis

To meet governmental requirements we use the Call Screening service, and configure it to handle incoming and outgoing calls. In addition, we expect the car's onboard computer to take control of the mobile phone to inform the network of its speed, location, and traffic density. The car computer, using an advanced highway GPS receiver and perhaps a WAP service, may obtain the latter two items of information. We do not consider this data to constitute a service on its own. This data need only be transmitted to the eSERL node periodically.

As for sending information about the potential for disconnection to participants other than the driver, the Information Delivery service shall be used.

For reestablishing disconnected calls, Auto-callback is required. When ACB tries to reestablish calls, such an action will be considered an as an outgoing call on Julie's behalf.

Finally, for determining whether Julie has driven more than 100km from home, the eSERL engine can calculate this and notify Mom using the Information Delivery service.

5.4. Encoding and Validation of Configuration Rules

A high-level abstraction of Julie's configuration may be defined as in Figure 17. A valid encoding of these rules using actual Parlay/OSA methods and events is given in Appendix C. We shall explain why the rule module is valid according to the constraints imposed for the system.

```
Rule 1(PP2/-2): If (INCOMING_CALL OR OUTGOING_CALL) {
                    Invoke CS(screening party: car)
Rule 2(PP2/-2):     If (response from car: Julie = AVAILABLE){
                        Invoke ID("call may be disconnected")
                    }
                }

Rule 3(PP1/-1): If (Session.CallExists(Julie)) {
Rule 4(PP1/-1):     If (INCOMING_CALL from car &&
                        Julie = BUSY) {
                        Invoke ACB
                         // which also terminates existing calls
                    }
                }
Rule 5(PP1/-1): Else {
Rule 6(PP1/-1):     If (INCOMING_CALL from car &&
                        Julie = AVAILABLE) {
Rule 7(PP1/-1):         If (Session.isWaiting(ACB)) {
                            ForwardEvent ACB
                        }
                    }
                }

Rule 8(PP2/-2): If (Julie's location is more than 100km from home){
                    Invoke ID
                }
```

Figure 17: Julie's Configuration Rules

Notice in Figure 17 that Rule 1 and Rule 2 and Rule 8, all at Processing Point 2/-2, may be satisfied simultaneously (i.e. overlapping) because their conditions refer to different variables, regardless of their dimension. Now between Rule 1 and Rule 2, an order must be specified for actions otherwise the constraint on invocation of CS before ID would be violated. Similarly an order between CS and ID in Rule 8 must be defined. Our validation

tool would give a number of errors, but we only show the following error related to the outgoing call possibility.

```
UNACCEPTABLE COMPOSITION:
line 57  (at Processing Points  2 -2)
IF
(CS.approved) matches (yes) AND
(EventContext.TriggerType) matches (Terminating) AND
(TpCallNotificationInfo.CallEventInfo.CallEventType) matches
(P_CALL_EVENT_TERMINATING_CALL_ATTEMPT)
THEN
INVOKE(priority 0) ID
-- WITH --
line 51  (at Processing Points  2 -2)
IF
(EventContext.TriggerType) matches (Terminating) AND
(TpCallNotificationInfo.CallEventInfo.CallEventType) matches
(P_CALL_EVENT_TERMINATING_CALL_ATTEMPT)
THEN
INVOKE(priority 0) CS
```

Figure 18: CS and ID Constraint Violation

To remedy this problem we may define an ordering between CS and ID by setting the priority of Rule 1 to 10, Rule 2 to 5, and Rule 8 to 5.

Now, an ordering must be defined between the two instances of ID, which may be invoked by Rules 2 and 8. Our initial plan to set both rule priorities to 5 is faulty. Our validation tool gives the following error.

```
UNACCEPTABLE COMPOSITION:
line 117  (at Processing Points  2 -2)
IF
(EventContext.TriggerType) matches (Originating) AND
(TpCallNotificationInfo.CallEventInfo.CallEventType) matches
(P_CALL_EVENT_ORIGINATING_CALL_ATTEMPT) AND
(CalculateLocationOverlap(Session.Participant("Julie").location,
new Location(HOME,100km))) matches (no)
THEN
INVOKE(priority 5) ID
-- WITH --
line 57  (at Processing Points  2 -2)
IF
(CS.approved) matches (yes) AND
(EventContext.TriggerType) matches (Originating) AND
(TpCallNotificationInfo.CallEventInfo.CallEventType) matches
(P_CALL_EVENT_ORIGINATING_CALL_ATTEMPT)
THEN
INVOKE(priority 5) ID
```

Figure 19: ID and ID Constraint Violation

Our solution is to assign a priority of 0 to Rule 8 in order to define the ordering.

In this example, Processing Points ensure that ACB will be invoked before CS and ID. Also, since CF is not considered in the configuration, there were no violations of mutual exclusion constraints to worry about.

We do not show data exchanged through the event context. As we have explained in Chapter 5, checking whether data parameters are set appropriately before service invocation is quite simple to do. We do not think that it merits an elaborate discussion here.

5.5. Result

The guarantee provided to the user through validation simply implies that the services will behave as required and specified by rules. Even if the configuration rules are valid, we cannot guarantee whether a user's *true intentions* for service behavior will be realized. 3rd party service providers with expertise in the area may develop tools (e.g. templates and wizards) that will help users express rules. It is the responsibility of these developers to ensure that the rule templates deliver what they promise in this respect. In this Case Study, we can show that the overall behavior of services achieves what is required by examining

an elaborate series of sequence diagrams. We have chosen not to present them herein because we believe such a lengthy discussion is unnecessary. We trust that the presentation of our proof of concept implementation in Chapter 6 and our rule validation results will suffice.

CHAPTER 8

CONCLUSION

In this chapter we highlight the contributions of our work in order of importance. Finally we discuss potential future work.

1. Summary of Contributions

We have designed a validation scheme to guarantee, to a certain degree, that end-user rules for composed service behavior can and will be met. Our scheme is flexible because we separate the service composition constraints imposed by experts from user-defined configuration rules and the execution environment (e.g. Parlay/OSA). In theory, our scheme may be applied in any domain where the set of possible variables for rule conditions are discrete, finite, and ordered. Even though our Case Studies revolve around Call Control, our approach is not limited to that domain.

We have designed and implemented a novel solution for allowing end-user personalization and composition of services based on SERL. Our solution enables a distinction between different quality of service levels in the application-layer service domain by allowing services to be used as stand-alone components, or in composed configurations with enhanced, personalized behavior.

We have designed and implemented our proof of concept prototype in the context of Parlay/OSA. Parlay/OSA was not originally designed to facilitate service composition and personalization, nor does it currently deal with feature interaction issues. We have managed to address both needs by constructing a Feature Interaction Management Service, which may be ported to any Parlay/OSA host with a minimal amount of work.

2. Future Research

Our current solution applies for a single user, with services hosted on one node. Our future work focuses on application of our concepts in a multi-user context. This involves new architectures, enhanced algorithms, and accompanying tools.

2.1. Distributed Architecture

In a distributed environment, FIMs will be located on several nodes and at different layers in the architecture. Such an architecture may require an extension to the Parlay/OSA APIs to enable communication between FIMs. In this scenario, multiple Application Servers with local FIMs exchange data (i.e. Composition Constraints, Configuration Rules, and more) through a lower-layer FIM located in the Service Capability Server (Parlay/OSA server side).

2.2. eSERL with Multiple Users

In a multi-user, single-component (MUSC) context all acceptable compositions of services in the system are still known a-priori. This context requires the merging of Configuration Rules at runtime for all call participants, and validating the merged configuration against the constraints. A mechanism would need to be defined for merging configurations and resolving conflicts due to merging at runtime.

In a multi-user, multi-component (MUMC) context, not only would Configuration Rules need to be merged, but also, so would Composition Constraints from all eSERL nodes along the signaling path. Here, the conflicts arising from the merging Composition Constraints would need to be resolved by an "expert". With the potential for thousands of calls through a system at a time, interrogating human experts is unrealistic. Other techniques, possibly involving negotiating software agent experts would be required.

2.3. Activation Rules

Activation Rules can be seen as meta-rules relating user-context to the appropriate Configuration Rule Module to activate. User-context has a very broad and somewhat ambiguous meaning, which may encompass a user's location, activity, membership, role, or more. For instance, when Anna is in her car, Configuration Rule Module 1 applies, but when she is in the office, Configuration Rule Module 2 applies. Handling of a 'busy' event related to a call request directed towards Anna would depend on the Configuration Rules defined within the currently active Configuration Rule Module. Open issues here include: where to position activation rules in the architecture, and how to validate them, or even implement them. A language to describe user-context will be required.

2.4. Service Life-Cycle Management Process

We expect that users will manage services through a Web-based interface. Service providers will deploy new services into the system by uploading service executables and deployment descriptors to a Web server. Experts will analyze services and update Composition Constraints for the system before deploying the new services onto eSERL-FIMs. Users will subscribe to services, and then compose and configure their subscribed services by accessing a Web-based configuration tool. The framework to allow such a process must be developed. In addition, as we have mentioned earlier on in this thesis, considerations must be made for non-monotonic extension of the system, where, for instance, and expert adds a new service or constraint which invalidates previously valid Configuration Rule Modules.

2.5. Theme-based Rule Templates and Wizards

Enabling end-user personalization and composition of services is probably not going to stimulate service network usage on its own since the task of expressing requirements in relation to services and capabilities in the network will still be too complicated for the average user. On the other hand, we believe that 3rd parties with intermediate-level expertise may develop tools to facilitate user definition of requirements, namely *rule templates* and *wizards*. In doing so, they may additionally *bundle-in* their own services in order to maximize revenue opportunities.

Rule templates may be written according to certain themes, for example, a *Family Theme*, *Driver's Theme*, *Real-Estate Agent's Theme*, or *World Traveller's Theme*. Each template would define a set of rules geared towards satisfying the needs of family members, automobile drivers, real-estate agents, or travellers respectively. With a template defined, a wizard would then query the end-user for data in order to personalize individual rules, for example, contacts and their addresses, geographic locations, interesting land properties, points of interest, and so on. The important thing to remember is that the end-user has no notion of how their rules will be realized in the network. In is the responsibility of the 3rd party developer to understand the behaviour of services, and generate rule templates that can actually be realized with the services and network capabilities available. When services or capabilities cannot meet the demands of rules, the 3rd party would develop their own

services and then bundle them in. This model significantly enhances the potential product offering for service providers. The same services could be used in many different themes, and indeed, themes can be developed and sold without incurring the costs of developing any new services at all.

A framework to facilitate the creation of such rule templates and wizards by 3rd parties should definitely be investigated in future work.

REFERENCES

[1] ACCENT Project at http://www.cs.stir.ac.uk/~kjt/research/accent.html

[2] ANISE Project at http://www.cs.stir.ac.uk/~kjt/research/anise.html

[3] L. G. Bouma and H. Velthuijsen, editors. Feature Interactions in Telecommunications Systems. IOS Press (Amsterdam), 1994.

[4] L. G. Bouma and H. Velthuijsen, Introduction of [14]

[5] M. Calder and E. Magill, editors. Feature Interactions in Telecommunications and Software Systems VI. IOS Press (Amsterdam), 2000.

[6] M. Calder, E. Magill, M. Kolberg, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast", *Computer Networks*, Volume 41/1, pp. 115-141, North-Holland. January 2003.

[7] E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W.K. Schnure and H. Velthuijsen, "A Feature Interaction Benchmark for IN and Beyond", in *Feature Interactions in Telecommunications Systems*, IOS Press, Amsterdam, pp. 1-23, 1994.

[8] E.J. Cameron and H. Velthuijsen, "Feature Interactions in Telecommunications Systems," IEEE Communications, vol. 31, no. 8, pp. 18-23, 1993.

[9] K. E. Cheng and T. Ohta, editors. Feature Interactions in Telecommunications Systems III. IOS Press (Amsterdam), 1995.

[10] Alessandro De Marco, Ferhat Khendek, "Feature Interaction Management using Composition Constraints and Configuration Rules", to appear in Feature Interactions in Telecommunications and Software Systems VII. IOS Press (Amsterdam), 2003.

[11] P. Dini, R. Boutaba, and L. Logrippo, editors. Feature Interactions in Telecommunication Networks IV. IOS Press (Amsterdam), 1997.

[12] ECLIPSE Project, http://www.research.att.com/projects/eclipse/

[13] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, 1-39, World Scientific Publishing Company, 1993. (Cited in [18])

[14] Roch H. Glitho and Kindy Sylla, "Developing Portable Applications for Internet Telephony: An Overview of Parlay and a Case Study on its Use in SIP Networks", submitted to IEEE Network Magazine, 2002.

[15] Roch H. Glitho, Ferhat Khendek, Alessandro De Marco, "Creating Value Added Services in Internet Telephony: An Overview and A Case Study on a High-Level Service Creation Environment", to appear in IEEE – Transactions on Systems, Man and Cybernetics, 2003.

[16] N. Griffeth, Y.-J. Lin, editors. Feature Interactions in Telecommunications Systems. IOS Press (Amsterdam), 1992.

[17] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, "SIP: Session Initiation Protocol", IETF RFC 2543, http://www.ietf.org/rfc/rfc2543.txt?number=2543

[18] Michael Jackson and Pamela Zave. "Distributed feature composition: A virtual architecture for telecommunications services". *IEEE Transactions on Software Engineering*, XXIV(10):831-847, October 1998.

[19] JAIN API Specifications at http://java.sun.com/products/jain/api_specs.html

[20] D. O. Keck and P. J. Kuehn, "The feature and service interaction problem in telecommunications systems: A survey", IEEE Transactions on Software Engineering, 24(10):779–796, October 1998. IEEE.

[21] K. Kimbler and L. G. Bouma, editors. Feature Interactions in Telecommunications and Software Systems V. IOS Press (Amsterdam), 1998.

[22] A. Kristensen and A. Byttner, "The SIP Servlet API", (work in progress), IETF Internet-Draft: draft-kristensen-sip-servlet-00.txt.

[23] J. Lennox and H. Schulzrinne, "Feature Interaction in Internet Telephony", 6th Workshop on Feature Interactions in Telecom and Software Systems, Scotland, June 2000.

[24] J. Lennox, and H. Schulzrinne, "Call Processing Language Framework and Requirements", IETF RFC, http://www.ietf.org/rfc/rfc2824.txt.

[25] J. Lennox, H. Schulzrinne, and J. Rosenberg, "Common Gateway Interface for SIP", IETF RFC, http://www.ietf.org/rfc/rfc3050.txt.

[26] Gerard Meszaros, "Half Object Plus Protocol", in *Pattern Languages of Program Design*, Vol. 1, James O. Coplien, Douglas C. Schmidt, eds. Addison-Wesley, 1995.

[27] Ard-Jan Moerdijk and Lucas Klostermann, "Opening The Networks With Parlay / OSA APIs: Standards And Aspects Behind The APIs", (work in progress), to be submitted to IEEE Communications Magazine, 2003. (http://www.parlay.org/specs/library).

[28] Parlay API Specifications at http://www.parlay.org

[29] H. Smith and R.W. Steenfeldt, "SERL Examples with SIP and SDP", (work in progress), IETF Internet-Draft: draft-smith-serl-ex-00.txt, May 7, 2001.

[30] R.W. Steenfeldt and H. Smith, "Service Execution Rule Language (SERL 1.0) for SIP", (work in progress), IETF Internet-Draft: draft-steenfeldt-sip-serl-00.txt, May 21, 2001.

[31] R.W. Steenfeldt and H. Smith, "SIP Service Execution Rule Language: Framework and Requirements", (work in progress), IETF Internet-Draft: draft-steenfeldt-sip-serl-fwr-00.txt, May 7, 2001.

[32] Kindy Sylla, *Parlay APIs and SIP protocol based Multipoint Control Unit*, Masters thesis. Université du Québec à Montréal, 2002.

[33] Third Generation Partnership Project (3GPP) at http://www.3gpp.org.

[34] Universal Description, Discovery, and Integration of Web Services, http://www.uddi.org

[35] VoiceXML Forum at http://www.voicexml.org/

[36] Dong Wang, Ruibing Hao, David Lee. "Fault Detection in Rule-Based Software Systems", Concordia Prestigious Workshop on Communication Software

Engineering, Montréal, Canada, Sept. 2001. Extended version to appear in the International Journal of Information and Software Technology, Elsevier, 2003.

[37] Web Services, http://www.w3.org/2002/ws/Activity

[38] P. Zave. Lecture on Address Translation Feature Interactions. Concordia Summer School on Communications Software Engineering, August 2002.

APPENDIX A: ESERL DTD

```
<?xml version='1.0' encoding='us-ascii'?>
<!--       Draft DTD for SERL, corresponding to
             draft-steenfeldt-sip-SERL-00.txt

           and extended by Alex De Marco for eSERL requirements
-->

      <!ELEMENT rulemodule    (owner, protocol, rmacl, rmid, serule*)>

      <!ELEMENT owner         (name, hostname+, aliashostname,
                               ipv4address, ipv6address, userinfo,
                               company, ownerid, serlversion)>

      <!ELEMENT serlversion   (#PCDATA)>
      <!ELEMENT name          (#PCDATA)>
      <!ELEMENT ownerid       (#PCDATA)>
      <!ELEMENT hostname      (#PCDATA)>
      <!ELEMENT aliashostname (#PCDATA)>
      <!ELEMENT ipv4address   (#PCDATA)>
      <!ELEMENT ipv6address   (#PCDATA)>
      <!ELEMENT userinfo      (user,password)>
      <!ELEMENT company       (#PCDATA)>
      <!ELEMENT user          (#PCDATA)>
      <!ELEMENT password      (#PCDATA)>
      <!ELEMENT protocol      (subprotocol*)>
      <!ELEMENT rmid          (#PCDATA)>
      <!ELEMENT serule        (property | action)+>
      <!ELEMENT property      (property | action)+>
      <!ELEMENT action        ((invoke |break), lock*, unlock*,
                              timeactive?)>
      <!ELEMENT invoke        (objname, rmid?, objowner?, success?,
                              failure?,notfound?)>
      <!ELEMENT break         EMPTY>
      <!ELEMENT lock          EMPTY>
      <!ELEMENT unlock        EMPTY>
      <!ELEMENT timeactive    EMPTY>
      <!ELEMENT objname       (#PCDATA)>
      <!ELEMENT objowner      (#PCDATA)>
      <!ELEMENT success       (log |charging | rmid | break)+>
      <!ELEMENT failure       (alarm | log | mail | rmid | break)+>
      <!ELEMENT notfound      (alarm | log | mail | rmid | break)+>
      <!ELEMENT log           (#PCDATA)>
      <!ELEMENT charging      (#PCDATA)>
      <!ELEMENT alarm         (#PCDATA)>
      <!ELEMENT mail          (#PCDATA)>
      <!ELEMENT rmacl         (acrule+)>
      <!ELEMENT acrule        (rmuser, privileges)>
      <!ELEMENT rmuser        (#PCDATA)>
      <!ELEMENT privileges    (read?, write?, execute?)>
      <!ELEMENT read          EMPTY>
      <!ELEMENT write         EMPTY>
      <!ELEMENT execute       EMPTY>
```

```
<!ATTLIST rulemodule
    priority          CDATA                          #REQUIRED
>


 <!ATTLIST owner
    class             ( network_operator |
                        service_provider |
                        account |
                        subscriber )          #REQUIRED
>


<!ATTLIST hostname
    port              CDATA                    #IMPLIED
>

<!ATTLIST aliashostname
    port              CDATA                    #IMPLIED
>

<!ATTLIST ipv4address
    port              CDATA                    #IMPLIED
>

<!ATTLIST ipv6address
    port              CDATA                    #IMPLIED
>

 <!ATTLIST protocol
    protocolname      CDATA                    #REQUIRED
    protocolversion   CDATA                    #REQUIRED
>

 <!ATTLIST subprotocol
    protocolname      CDATA                    #REQUIRED
    protocolversion   CDATA                    #REQUIRED
>

 <!ATTLIST serule
    processing-point  CDATA                    #REQUIRED
>

<!-- dimension attribute added by Alex, for eSERL -->
 <!ATTLIST property
    name              CDATA                    #REQUIRED
    matches           CDATA                    #REQUIRED
    dimension         (
                       unknown       |
                       time          |
                       system        |
                       event_context |
                       parlay_mpcc   |
                       parlay_gcc    |
                       parlay_mmcc   |
                       parlay_cccs   |
                       parlay_ui     |
```

```
                        parlay_mobi
                       )                                #REQUIRED
    not                (yes|no) "no"
    case-sensitive     (yes|no) "no"
>

 <!ATTLIST invoke
    key                CDATA                            #IMPLIED
    type               (serl|
                        cpl|
                        sip-cgi|
                        sip-servlet|
                        osa| 3GPPSc |
                        system|
                        sip|
                        http|
                        ipv4| ipv6|
                        hostname)                        #REQUIRED
    typeversion        CDATA                            #IMPLIED
    subscriber         (from|
                        Request-URI|
                        forwardedby)                     #IMPLIED

   ownerclass         ( network_operator |
                        service_provider |
                        account |
                        subscriber)                      #IMPLIED

    onresponse         (yes|no) "no"
    reponselist        CDATA                            #IMPLIED
    continue           (yes|no) "no"
>

 <!ATTLIST log
    name               CDATA                            #IMPLIED
    comment            CDATA                            #IMPLIED
>

 <!ATTLIST mail
    url                CDATA                            #IMPLIED
    subject            CDATA                            #IMPLIED
>

 <!ATTLIST failure
    error               CDATA                           #IMPLIED
    timeout             CDATA                           #IMPLIED
    default            (proxy|redirect)  "proxy"
>

 <!ATTLIST notfound
    error               CDATA                           #IMPLIED
    default            (proxy|redirect)  "proxy"
>

 <!ATTLIST timeactive
    tzid               CDATA                            #IMPLIED
    turl               CDATA                            #IMPLIED
```

```
        dtstart             CDATA                       #IMPLIED
        dtend               CDATA                       #IMPLIED
        duration            CDATA                       #IMPLIED
    >

     <!ATTLIST lock
        object              (message|property)          #REQUIRED
        name                CDATA                       #IMPLIED
    >

     <!ATTLIST unlock
        object              (message|property|keep)     #REQUIRED
        name                CDATA                       #IMPLIED>

     <!ATTLIST rmuser
        class               (network_operator |
                             service_provider |
                             account |
                             subscriber)                #REQUIRED
    >

     <!ATTLIST read
        switch              (yes|no)                    #REQUIRED
    >

     <!ATTLIST write
        switch              (yes|no)                    #REQUIRED
    >

     <!ATTLIST execute
        switch              (yes|no)                    #REQUIRED
    >


<!-- added by Alex, for eSERL -->
<!ELEMENT composition_rulemodule (owner, protocol, rmacl, rmid,
service_object*, constraint*)>
<!ELEMENT constraint     (property | action)+>
<!ELEMENT service_object (objname, objowner, objproperty*)>
<!ELEMENT objproperty    (#PCDATA)>
<!ATTLIST service_object
        processing-point   CDATA                       #REQUIRED
>
<!ATTLIST constraint
        type           (mutex | order | select)        #REQUIRED
>
<!ATTLIST objproperty
        name   CDATA                                   #REQUIRED
        type           (in | out | inout)                          #REQUIRED
>
<!ATTLIST action
        priority      CDATA                      #IMPLIED
        mode               (once|permanent) "permanent"
                                    >
```

APPENDIX B: COMPOSITION CONSTRAINTS FOR CASE STUDIES

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE composition_rulemodule SYSTEM "dtd/serl.dtd" >
<composition_rulemodule>


  <owner class="network_operator">
    <name>ECE - Concordia University Telesoft Labs</name>
    <hostname>chirp.win.ece.concordia.ca</hostname>
    <aliashostname>chirp</aliashostname>
    <ipv4address>132.205.3.31</ipv4address>
    <ipv6address>132.205.3.31</ipv6address>
    <userinfo>
      <user>Alex De Marco</user>
      <password>mypassword</password>
    </userinfo>
    <company>Concordia University</company>
    <ownerid>ownerID1234</ownerid>
    <serlversion>2.0 (eSERL)</serlversion>
  </owner>
  <protocol protocolname="Parlay/OSA" protocolversion="3.0"/>
  <rmacl>
    <acrule>
      <rmuser class="account">Alex De Marco</rmuser>
      <privileges>
        <read switch="yes"/>
        <write switch="yes"/>
        <execute switch="yes"/>
      </privileges>
    </acrule>
  </rmacl>
  <rmid>rulemoduleID1234</rmid>


  <service_object processing-point="1,-1">
    <objname>ca.concordia.ece.telesoft.org.csapi.services.CF
    <!--Call Forward, where the event to trigger the forwarding is
specified by the user, for example, BUSY, NO_ANSWER, etc...--></objname>
    <objowner>ECE - Concordia University Telesoft Labs</objowner>
    <objproperty name="callee" type="inout"/>
    <objproperty name="caller" type="inout"/>
    <objproperty name="forwardTo" type="in"/>
  </service_object>


  <service_object processing-point="1,-1">
    <objname>ca.concordia.ece.telesoft.org.csapi.services.ACB
    <!--ACB = Auto-CallBack. This service attempts to re-establish a
connection at a later time. It needs two trigger points, one to save the
old number, and another trigger to attempt the callback to the old
number.--></objname>
    <objowner>ECE - Concordia University Telesoft Labs</objowner>
```

```
    <objproperty name="callee" type="inout"/>
    <objproperty name="caller" type="inout"/>
  </service_object>
  <service_object processing-point="2,-2">
    <objname>ca.concordia.ece.telesoft.org.csapi.services.ID
    <!--ID = Information Delivery using instant messaging.--></objname>
    <objowner>ECE - Concordia University Telesoft Labs</objowner>
    <objproperty name="callee" type="in"/>
    <objproperty name="caller" type="in"/>
    <objproperty name="deliverTo" type="in"/>
    <objproperty name="approved" type="out"/>
  </service_object>


  <service_object processing-point="2,-2">
    <objname>ca.concordia.ece.telesoft.org.csapi.services.CS
    <!--Call Screening - screens a call (before establishment) based on
certain criteria.--></objname>
    <objowner>ECE - Concordia University Telesoft Labs</objowner>
    <objproperty name="caller" type="in"/>
    <objproperty name="callee" type="in"/>
    <objproperty name="criteria" type="in"/>
    <objproperty name="approved" type="out"/>
  </service_object>


<constraint type="mutex">
    <action>
      <invoke type="osa">
       <objname>ca.concordia.ece.telesoft.org.csapi.services.CF</objname>
      </invoke>
    </action>
    <action>
      <invoke type="osa">
      <objname>ca.concordia.ece.telesoft.org.csapi.services.ACB</objname>
      </invoke>
    </action>
    <!--ACB and CF cannot be invoked for the same event. They are
therefore MUTEX.-->
  </constraint>


  <constraint type="order">
    <action mode="permanent">
      <invoke type="osa">
       <objname>ca.concordia.ece.telesoft.org.csapi.services.CS</objname>
      </invoke>
    </action>
    <action mode="permanent">
      <invoke type="osa">
       <objname>ca.concordia.ece.telesoft.org.csapi.services.ID</objname>
      </invoke>
    </action>
  </constraint>
```

```
<constraint type="select">
    <property dimension="system"
matches="ACB" name="InvokedServices">
      <property dimension="event_context"
matches="ACB.caller | ACB.callee" name="CS.caller">
        <property dimension="event_context"
matches="ACB.caller | ACB.callee" name="CS.callee">
          <property dimension="event_context"
matches="CS.caller" name="CS.callee" not="yes">
            <property dimension="event_context"
matches="null" name="CS.criteria" not="yes">
              <action mode="permanent">
                <invoke type="osa">
<objname>ca.concordia.ece.telesoft.org.csapi.services.CS</objname>
                </invoke>
              </action>
            </property>
          </property>
        </property>
      </property>
    </property>
  </constraint>


  <constraint type="select">
    <property dimension="system"
matches="ACB" name="InvokedServices">
      <property dimension="event_context"
matches="ACB.caller | ACB.callee" name="ID.caller">
        <property dimension="event_context"
matches="ACB.caller | ACB.callee" name="ID.callee">
          <property dimension="event_context"
matches="ID.caller" name="ID.callee" not="yes">
            <property dimension="event_context"
matches="ID.caller | ID.callee" name="ID.deliverTo">
              <action mode="permanent">
                <invoke type="osa">
<objname>ca.concordia.ece.telesoft.org.csapi.services.ID</objname>
                </invoke>
              </action>
            </property>
          </property>
        </property>
      </property>
    </property>
  </constraint>


<constraint type="select">
    <property dimension="system"
matches="CF" name="InvokedServices">
      <property dimension="event_context"
matches="CF.caller | CF.callee" name="CS.caller">
        <property dimension="event_context"
matches="CF.caller | CF.callee" name="CS.callee">
          <property dimension="event_context"
```

```
matches="CS.caller" name="CS.callee" not="yes">
            <property dimension="event_context"
matches="null" name="CS.criteria" not="yes">
              <action mode="permanent">
                <invoke type="osa">
<objname>ca.concordia.ece.telesoft.org.csapi.services.CS</objname>
                </invoke>
              </action>
            </property>
          </property>
        </property>
      </property>
    </property>
  </constraint>


  <constraint type="select">
    <property dimension="system"
matches="CF" name="InvokedServices">
      <property dimension="event_context"
matches="CF.caller | CF.callee | CF.forwardTo" name="ID.caller">
        <property dimension="event_context"
matches="CF.caller | CF.callee | CF.forwardTo" name="ID.callee">
          <property dimension="event_context"
matches="ID.caller" name="ID.callee" not="yes">
            <property dimension="event_context"
matches="ID.caller | ID.callee" name="ID.deliverTo">
              <action mode="permanent">
                <invoke type="osa">
<objname>ca.concordia.ece.telesoft.org.csapi.services.ID</objname>
                </invoke>
              </action>
            </property>
          </property>
        </property>
      </property>
    </property>
  </constraint>

</composition_rulemodule>
```

APPENDIX C: CONFIGURATION RULES FOR JULIE JONES

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rulemodule SYSTEM "dtd/serl.dtd" >
<rulemodule priority="1">
  <owner class="subscriber">
    <name>Julie</name>
    <hostname>chirp.win.ece.concordia.ca</hostname>
    <aliashostname>chirp</aliashostname>
    <ipv4address>132.205.3.31</ipv4address>
    <ipv6address>132.205.3.31</ipv6address>
    <userinfo>
      <user>Julie</user>
      <password>pass</password>
    </userinfo>
    <company>Jones Family</company>
    <ownerid>ownerID1234</ownerid>
    <serlversion>2.0 (eSERL)</serlversion>
  </owner>
  <protocol protocolname="Parlay/OSA" protocolversion="3.0"/>
  <rmacl>
    <acrule>
      <rmuser class="subscriber">Bob</rmuser>
      <privileges>
        <read switch="yes"/>
        <write switch="yes"/>
        <execute switch="yes"/>
      </privileges>
    </acrule>
  </rmacl>
  <rmid>rmid2</rmid>


  <serule processing-point="2,-2">
    <property dimension="event_context"
matches="Originating"
name="EventContext.TriggerType">
      <property dimension="parlay_gcc"
matches="P_CALL_EVENT_ORIGINATING_CALL_ATTEMPT"
name="TpCallNotificationInfo.CallEventInfo.CallEventType">
        <action priority="10">
          <invoke continue="no" type="osa">
            <objname>CS</objname>
          </invoke>
        </action>
        <property dimension="event_context"
matches="yes"
name="CS.approved">
          <action priority="5">
            <invoke type="osa">
              <objname>ID</objname>
            </invoke>
          </action>
        </property>
```

```
      </property>
    </property>
    <property dimension="event_context"
matches="Terminating"
name="EventContext.TriggerType">
      <property dimension="parlay_gcc"
matches="P_CALL_EVENT_TERMINATING_CALL_ATTEMPT"
name="TpCallNotificationInfo.CallEventInfo.CallEventType">
        <action priority="10">
          <invoke continue="no" type="osa">
            <objname>CS</objname>
          </invoke>
        </action>
        <property dimension="event_context"
matches="yes"
name="CS.approved">
          <action priority="5">
            <invoke type="osa">
              <objname>ID</objname>
            </invoke>
          </action>
        </property>
      </property>
    </property>
  </serule>


  <serule processing-point="1,-1">
    <property dimension="event_context"
matches="Originating"
name="EventContext.TriggerType">
      <!--Car notifies network about Julie by "calling the network" on
Julie's behalf. This is seen as an "originating call attempt".-->
      <property dimension="parlay_gcc"
matches="P_CALL_EVENT_ORIGINATING_CALL_ATTEMPT"
name="TpCallNotificationInfo.CallEventInfo.CallEventType">
        <!--If a call exists (i.e. is in session currently) when the car
notifies that Julie becomes BUSY.-->
        <property dimension="system"
matches="yes"
name="Session.CallExists">
          <property dimension="system"
matches="BUSY"
name="Session.Participant(&amp;#34;Julie&amp;#34;).state">
            <action>
              <invoke continue="yes" type="osa">
                <!--Invoke ACB, but at this trigger point, we only save
the call info so we can attempt reconnection later.-->
                <objname>ACB</objname>
              </invoke>
            </action>
          </property>
        </property>
        <property dimension="system"
matches="yes"
name="Session.CallExists" not="yes">
          <property dimension="system"
```

```
matches="AVAILABLE"
name="Session.Participant(&#34;Julie&#34;).state">
            <property dimension="system"
matches="yes"
name="Session.WaitingServicesList.contains(&#34;ACB&#34;)">
              <action>
                <invoke continue="no" type="osa">
                  <!--Invoke ACB, but at this trigger point, we attempt
the reconnection.-->
                  <objname>ACB</objname>
                </invoke>
              </action>
            </property>
          </property>
        </property>
      </property>
    </property>
  </serule>


  <serule processing-point="2,-2">
    <property dimension="event_context"
matches="Originating"
name="EventContext.TriggerType">
      <!--Car notifies network about Julie by "calling the network" on
Julie's behalf. This is seen as an "originating call attempt".-->
      <property dimension="parlay_gcc"
matches="P_CALL_EVENT_ORIGINATING_CALL_ATTEMPT"
name="TpCallNotificationInfo.CallEventInfo.CallEventType">
        <!--If location is > 100 km from home-->
        <property dimension="system"
matches="no"
name="CalculateLocationOverlap(Session.Participant(&#34;Julie&#34;).locat
ion, new Location(HOME,100km))">
          <action>
            <invoke type="osa">
              <objname>ID</objname>
            </invoke>
          </action>
        </property>
      </property>
    </property>
  </serule>
</rulemodule>
```