Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 81 (2008) 1640–1662

The Journal of Systems and Software

www.elsevier.com/locate/jss

CVM – A communication virtual machine

Yi Deng, S. Masoud Sadjadi *, Peter J. Clarke, Vagelis Hristidis, Raju Rangaswami, Yingbo Wang

School of Computing and Information Sciences, Florida International University, Miami, FL 33199, USA

Available online 20 February 2008

Abstract

The convergence of data, voice, and multimedia communication over digital networks, coupled with continuous improvement in network capacity and reliability has resulted in a proliferation of communication technologies. Unfortunately, despite these new developments, there is no easy way to build new application-specific communication services. The stovepipe approach used today for building new communication services results in rigid technology, limited utility, lengthy and costly development cycle, and difficulty in integration. In this paper, we introduce communication virtual machine (CVM) that supports rapid conception, specification, and automatic realization of new application-specific communication services through a user-centric, model-driven approach. We present the concept, architecture, modeling language, prototypical design, and implementation of CVM in the context of a healthcare application.
Published by Elsevier Inc.

Keywords: Model-driven architecture; Communication services; Multimedia; Middleware; Telemedicine

1. Introduction

The convergence of data, voice, and multimedia over digital networks coupled with the continuous improvement in network capacity and reliability has enabled a wide range of communication-intensive applications. Examples range from general-purpose communication applications such as VoIP telephony, voice, video or multimedia conferencing to specialized applications such as disaster management and telemedicine. The pace of innovation of new communication-intensive applications will undoubtedly accelerate further as both capacity and demand increase. This trend in communication-intensive applications is reminiscent of the rise of data-intensive applications during the late 1970s through the 1990s.

There are several problems, however, with the current approach for developing communication-intensive applications. First, today's communication tools are developed in stovepipe fashion with limited separation between user-level communication logic, device types, and underlying networks. This has led to the use of multiple specialized communication tools like email, answering machines, fax, and custom-made teleconferencing applications. Unfortunately, these tools cannot serve unanticipated communication needs without first incurring a lengthy development cycle, and consequently, high cost. Second, such vertically developed systems typically have fixed functionality and interface, and do not interoperate with each other because of the differences in design, architecture, API, and network/device assumptions. It is difficult to adapt these systems to fit changing user needs, the dynamics of underlying networks, and new device and network technologies (Krebs, 2005). Users, particularly sophisticated domain specific users, are forced to switch between tools to satisfy their communication needs. Third, the fragmented development approach poses major challenges with respect to integration when providing integrated communication solutions.

[*] Corresponding author. Tel.: +1 305 348 1835; fax: +1 305 348 3549.
E-mail addresses: deng@cis.fiu.edu (Y. Deng), sadjadi@cis.fiu.edu (S. Masoud Sadjadi), clarkep@cis.fiu.edu (P.J. Clarke), vagelis@cis.fiu.edu (V. Hristidis), raju@cis.fiu.edu (R. Rangaswami), ywang002@cis.fiu.edu (Y. Wang).

0164-1212/$ - see front matter Published by Elsevier Inc.
doi:10.1016/j.jss.2008.02.020

Last but not the least, it hinders the development of new communication tools, particularly for domain specific applications (e.g., telemedicine), because of the complexity, cost, and lengthy cycle required for vertical development.

In this paper, we present a fundamentally different approach for engineering communication solutions. This approach, which we call *communication virtual machine* (*CVM*), represents a paradigm shift on how communication applications are conceived and delivered. We argue that the CVM approach provides the basis to effectively address the problems discussed above. The basic premise of the CVM approach is that such a costly development process can be bypassed and largely eliminated by providing a model-driven platform for formulating, synthesizing, and executing new communication services. When a new (communication) service is needed, a model called *communication schema* that specifies the requirements and flow for the service is built and fed as input to the CVM. The CVM synthesizes the communication schema into a *communication control script*, which contains step by step instructions (e.g., initiating a voice call), on how the new service should be executed. The CVM then executes the script and delivers services required by the user. (See Section 4 for more detailed discussion of this process.) Consequently, the CVM approach transforms a full blown development process of delivering a new communication service into a modeling process.

We argue that this approach is feasible for *most* conceivable communication services. This is because the basic functions of communication (e.g., voice/video call, conferencing, file/data exchange, and messaging) are common across different applications; therefore, the interface to data sources, underlying networks, and different device types can be normalized. The logic and workflow for communication services are fairly simple (compared to general software systems), and thus can be synthesized from higher-level models. We make no claim that this approach can handle every possible application, but we do argue that it is general enough to have far reaching utility and impact.

Furthermore, this approach offers a number of advantages over the vertical development-based approach in terms of flexibility, adaptability, and interoperability, because it separates the logic and control of user communication from communication networks, devices, data sources, and add-on functions. For instance, security and privacy functions are not addressed in our current prototypical implementation of CVM. However, the extensible design of CVM allows existing tools for password protection, authentication, access control, and auditing to be readily added to the CVM platform and made available to all communication services. A change of device type will amount to adding a new device interface in CVM. A change in user-communication requirements only leads to change of a communication schema. A new mediator can be added for access new data sources, etc. Since even a sophisticated communication schema can be built in terms of hours or days, rather than months or years needed for designing and implementing a major communication application (e.g., telemedicine), CVM provides an effective way to support complex application-specific communication needs.

A layered CVM architecture will be presented. These layers are common to and shared by different communication applications. This architecture separates and encapsulates major concerns of communication modeling, synthesis, coordination, and the actual delivery of the communication by the underlying network and devices, into self-contained compartments with clear interface and responsibility. We will show that this architectural principle of separation of concerns employed in CVM is the basis for its automation and flexibility. This is because system components and communication protocols that are common to different applications can be identified and shared without having to be hard coded into a stovepipe system. This will also enable the CVM architecture to be independent of the underlying networking infrastructure and communication devices.

Coupled with the CVM architecture, there are several major components that together form the CVM system: *A communication modeling language* that provides an intuitive graphic form for users (or user organizations) to declaratively model their communication requirements (in terms of communication schema); a *synthesis engine* that negotiates and synthesizes user-communication sessions; a *communication engine* that executes user-communication logic; and *network communication broker* that interfaces with the underlying network infrastructure.

The design of CVM draws from the concepts of model-driven engineering (Bettin, 2004; Schmidt, 2006) communication middleware (Schmidt, 1997) and middleware-based architecture (Schmidt, 2002). However, by focusing on the communication aspects of the application, CVM achieves a high degree of automation and effectiveness, and avoids the pitfalls of many general purpose methods and techniques for model-driven engineering that are overreaching and consequently ineffective. Furthermore, the CVM approach goes far beyond the goals of communication middleware towards end-to-end communication solutions.

A prototypical design of the CVM is implemented and is fully functional. In addition to supporting general purpose communication functions (e.g., multimedia conferencing), we have worked with physicians and technical staff at the Miami Children's Hospital (MCH) and the Teges Corporation, which supplies MCH with its patient medical information system called *i-Rounds*™ (TegesTM Corporation, 2005; Burke and White, 2004). We have conducted case studies of using the CVM to support communication between doctors involved in cardiovascular operations based on scenarios and criteria formulated by the doctors. We have demonstrated that it takes less than a day to discuss, formulate and structure the telemedicine scenarios into CVM communication schemas; and it takes two graduate students a week to integrate the CVM system with the i-Rounds patient information systems, which transforms CVM into a communication platform capable of supporting a variety of telemedicine communications with structured exchange of patient information.

In the rest of this paper, we present the concept, architecture, modeling language, prototypical design, and implementation of CVM. Without compromising the applicability of CVM in other application domains, we use a healthcare communication application as the case study.

2. Motivating example

The authors have been collaborating with members of the cardiology division of Miami Children's Hospital (MCH) over the last 2 years to study the applications of CVM in healthcare. One such scenario is post-surgery consultation among the heart surgeon, attending physician, and referring physician.

Scenario: Baby Jane, who appears to have a heart condition, has been referred to MCH by Dr. Sanchez for observation and additional tests. Dr. Monteiro, the attending physician for baby Jane at MCH, performs additional tests on baby Jane and determines that her heart condition is severe. Dr. Monteiro consults with Dr. Burke, the chief cardiovascular surgeon at MCH, and a decision is made to perform surgery the following day. After Dr. Burke performs the surgery, he returns to his office and contacts both Dr. Sanchez and Dr. Monteiro to let them know how the surgery went and to share several aspects of Jane's medical record with them, including the post-surgery echocardiogram (echo), images of the patient's heart captured during the surgery, and the vital signs. Dr. Burke also needs to outline the post surgery care for baby Jane that should be followed by Dr. Monteiro at MCH for the next two weeks.

Supporting such a communication requires voice/video conferencing, compiling the data to be shared, exchange different types of data (e.g., text, image, and video), access to medical information system, exchange (part of) the patient's medical record, and logging the consultation for later reference.

Clearly, carrying out this scenario is possible with today's technology. For instance, Dr. Burke can make a conference call with Dr. Sanchez and Dr. Monteiro using a conferencing tool. Although feasible, it would be problematic for Dr. Burke during his conversation to compile the data from baby Jane's patient record and send it in real-time. In case either Dr. Sanchez or Dr. Monteiro does not have access to this specific application, Dr. Burke would have to use a separate means to share the patient's medical record with his colleagues, either through a custom-developed telemedicine application, or extracting the part of the record into a file, or more likely, fax the data sheets to the other doctors. A paper record or separate data entry is also needed for logging the session. In general, although such scenarios can be accommodated with today's technology, the users would either have to switch between different tools (e.g., phone, email, file-sharing, and messenger applications), or to rely on custom-developed applications, which are typically expensive and rigidly designed.

In the following sections, we show how this scenario, as well as any other similar scenarios, can be satisfied on-demand and with ease using CVM.

3. CVM overview

To better understand the motivation for CVM, let us consider what has taken place in the data management area. Fig. 1 captures the parallelism between the CVM paradigm and the one for managing heterogeneous data sources. The increased use of computers and databases has resulted in data being dispersed across a large number of data sources with different data models (e.g., relational, XML, text, and so on), data schemas and querying interfaces (e.g., SQL, XQuery, and keyword search). This heterogeneity meant that applications were now required to access multiple data sources, with possibly different data models and querying interfaces, resulting in the development of specialized access mechanisms for each data source. Using specialized access mechanisms to access data is cumbersome and inflexible since, if a legacy database is replaced by a newer one, the application has to be changed. Hence, the logical data abstraction paradigm was proposed, to hide the specifics of the underlying sources and export a uniform interface to the applications for querying data. The right-hand side of Fig. 1 shows a popular mediator architecture (Chawathe et al., 1994), where XQuery (see www.w3.org/XML/Query/) is used as the common data extraction language.

In the communication domain there is a similar need to hide the underlying device and network infrastructure and provide a unified communication abstraction layer. The CVM plays the role of the mediator and it handles the execution of communication requests specified in CML (communication modeling language explained in Section 5, which corresponds to XQuery). The wrappers on the left and right side of Fig. 1 play the role of abstracting the network/device and data specifics, respectively. Finally, the SIP messages (see www.iptel.org) play the same role as break-down of XQuery queries. Notice that the dashed lines with arrows between the left and right sides of Fig. 1 depict the correspondences between the two paradigms.

In the rest of this section, we present a CVM architectural model for achieving the vision discussed earlier. There are four major tasks need to be performed to serve the users' communication needs (Deng et al., 2006):

(1) Conceive and describe the users' communication requirements. For example, (1) a multimedia conference involves specifying who the participants of the conference are and what kind of media or data are to be exchanged, and (2) a telemedicine application includes specifying the policy that governs who can access which part(s) of the patient's medical record.
(2) Transform the user-communication requirements into a sequence of commands or actions, which when executed will control the flow of user communication as dictated by the requirements.

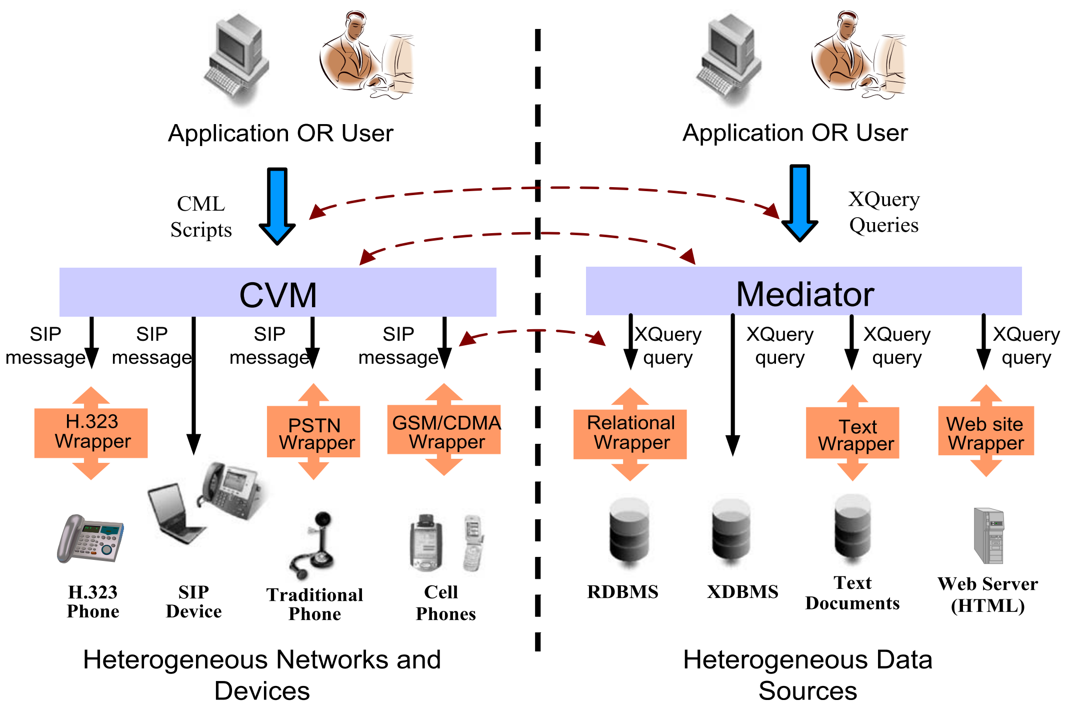


Fig. 1. Parallelism to mediator technology in data management.

(3) Provide a platform or environment in which the said sequence of commands can be executed to regulate the flow of communication.
(4) Deliver the media or data among the communicating parties through one or more communication networks.

Today, these tasks are typically hard coded in a communication system or tool, which pre-defines the way that a user will use the system. Such a stovepipe design is the root cause of the problems discussed in Section 1. At the heart of CVM is a layered architecture, which provides a clean separation and compartmentalization of these major concerns in the spirit of Buschmann et al. (1998), as illustrated in Fig. 2. The CVM architecture divides the major communication tasks into four major levels of abstraction, which correspond to the four key components of CVM mentioned above:

(1) *User-communication interface* (UCI), which provides a language environment for users to specify their communication requirements in the form of a *user-communication schema* or *schema instance*[1];
(2) *Synthesis engine* (SE), which is a suite of algorithms that automatically synthesize a user-communication schema instance to an executable form called *communication control script*;
(3) *User-centric communication middleware* (UCM), which executes the communication control script to manage and coordinate the delivery of communication services to users, independent of the underlying network configuration; and
(4) *Network communication broker* (NCB), which provides a network-independent API to UCM and works with the underlying network protocols to deliver the communication services.

This layered division of responsibility is reminiscent of the OSI layered stack model for network communication (Day and Zimmermann, 1995). Each layer has a specific role in the stack and communicates logically with the peer-layer at a remote site during communication sessions. Each layer builds on the upper layers in the stack to finally realize the user-communication schema.

UCI is responsible for providing users with a means to define and manage their communication schema, which describes the role of communicating parties and the communication logic (e.g., participants, flow of data or information). For this purpose, a communication modeling language is needed (Clarke et al., 2006). Such a language (see Section 5) should be intuitive enough to support on-the-fly communication modeling without requiring knowledge of underlying networks and yet rich enough to describe a variety of communication tasks. The language design poses many interesting research issues in its own right. In addition, it is responsible for maintaining consistency between the views of participants, and for serving as the runtime interface for users to manage their sessions.

[1] A schema is a generic model of communication; a schema instance is an instantiation of the schema for a particular communication session. We will use the terms interchangeably until Section 5.

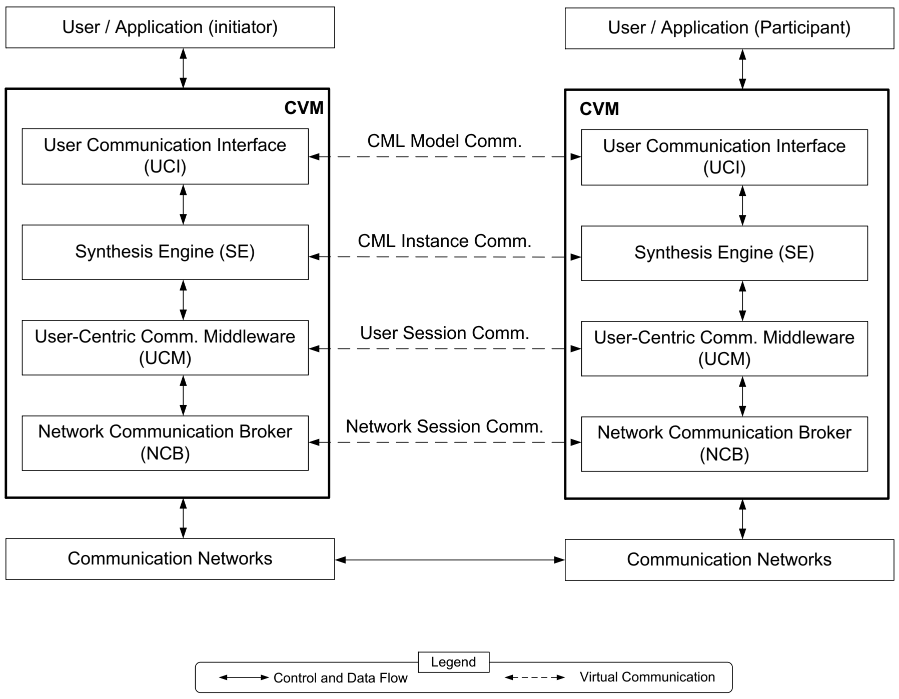


Fig. 2. Layered architecture of the communication virtual machine.

SE performs two major tasks (Rangaswami et al., 2007): The first is schema negotiation among participants of communication to ensure that all parties agree to a consistent schema. Second, SE automatically transforms the schema to an executable communication control script. This script represents the network-independent control logic for user-level communication session specified in the schema. A basic requirement for SE is that the synthesis process must be fully automated. SE uses a repository of pre-defined components for common, as well as domain-specific communication functions. SE puts together the communication control script by combining pre-defined components (e.g., communication session establishment or transmission of text message) based on the user-defined schema. Consequently, the capability of a schema synthesizer can be improved incrementally as more "middleware" components are developed. The design of automated and efficient synthesis techniques and the middleware components represents another class of interesting research issues.

UCM is the execution engine for communication control scripts. Based on the communication logic defined in the script, UCM invokes the common services provided by the NCB layer to perform tasks including (1) creating session, (2) adding a participant to the session, (3) adding a media to the session, (4) transmitting media, and (5) adjusting media QoS. UCM is also responsible for updating the user-communication schema resulted from runtime changes. These changes (received in the form of signals from the NCB layer) may include (1) initiate session, (2) receive media, (3) end media transmission, and (4) connection failed. Furthermore, UCM is responsible for providing a safe state transition between the running and updated communication control scripts. For example, when a session participant changes the communication schema by switching from a person-to-person call to a multi-way conference, SE will generate a new communication control script that reflects the change. Once the new communication control script is deployed to UCM, it should transfer the state of the old control script to the new one seamlessly and safely (Zave et al., 2004). The UCM is also responsible for the enforcement of policies contained in the communication schema instance. The policies may include communication constraints, security properties and other QoS concerns.

NCB is responsible for providing a uniform API of high-level and network-independent communication services to diverse communication applications (Zhang et al., 2006), in such a way to shield user applications from the underlying network/device heterogeneity and dynamics. It utilizes and coordinates networking functions (e.g., signaling, encoding and decoding, and transmitting and receiving) provided by the underlying networks, systems, and libraries. Given the variety and complexity of network configurations, it must exhibit a self-managing behavior that can respond to dynamics of the underlying device and network infrastructure. The concept of NCB offers a novel approach that simplifies application development and interoperation, and introduces many important research issues including self-management, dynamic configuration, definition of application independent communication API, and software framework for hiding network heterogeneity.

These layers collectively fulfill the promise of CVM that of generating communication applications that are

Table 1
High-level tasks of CVM layers

CVM layer	Tasks
UCI	1. Create/modify the communication schema instance based on user input 2. Check the correctness and validity of the user-communication schema 3. Maintain consistency between participants' instances
SE	1. Ensure the consistency of user-communication schema through schema negotiation 2. Perform schema synthesis to obtain the communication control script 3. Deploy the script to the user-centric communication middleware
UCM	1. Execute the communication control script 2. Update the user-communication schema based on changes made by other participants 3. Perform a safe state transition from an older schema to an updated one 4. Enforcement of schema policies
NCB	1. Provide a high-level communication API, which is independent of the platform 2. Utilize and coordinate the available, low-level network and hardware services 3. Provide self-management in response to dynamics of the underlying infrastructure

reconfigurable, adaptive, and flexible based only on a high-level description of communication requirements. A summary of the high-level responsibilities assigned to each of these layers is presented in Table 1.

4. Design of CVM

In this section, we present a design for CVM that provides communication services to a user application and support the querying of a data source using a specialized CVM mediator. Fig. 3 shows the top-level architecture of CVM which consists of the following major components: the *User Interface* that provides a easy to use GUI for novice users; the *CVM Mediator* that queries the data source and formats the results using pre-defined forms; and the *CVM Repository* that stores the profile for each CVM user. The data in the CVM repository is partitioned into: (1) data on contacts – a list of participants that have been added to the user's profile; and (2) a set of communication schemas and instances – these are schemas and instances that the user decided to store to the repository. The separation of the contact information and communication schemas allows the NCB layer in the CVM to include other systems that provide low-level communication services such as Skype (Skype Limited, 2007), Google Talk, (Google, 2007), or applications developed using the Eclipse Communication Framework (Eclipse, 2007). Based on the design presented in this section we have developed a prototype, described in Section 6, that supports the practicality of our design.

In the rest of this section, we introduce the internal design of the CVM's four layers and the Mediator. The components of the four layers in the CVM communicate using the following interfaces: *Provides*, *Uses*, *Handles*, and *Signals* (adopted from Hill et al., 2000). Table 2 summarizes the method invocations and the callbacks used to realize communication services provided by the CVM. Notice that each layer uses (resp. handles) what the lower level provides (resp. signals).

4.1. User-communication interface

The architectural design of UCI, shown in Fig. 4, consists of five major components: (1) the *UI Adapter* – provides an API to the user interface that allows for the creation of a communication service schema or instance; (2) the *Communication Modeling Environment* – provides the user with an environment to develop communication

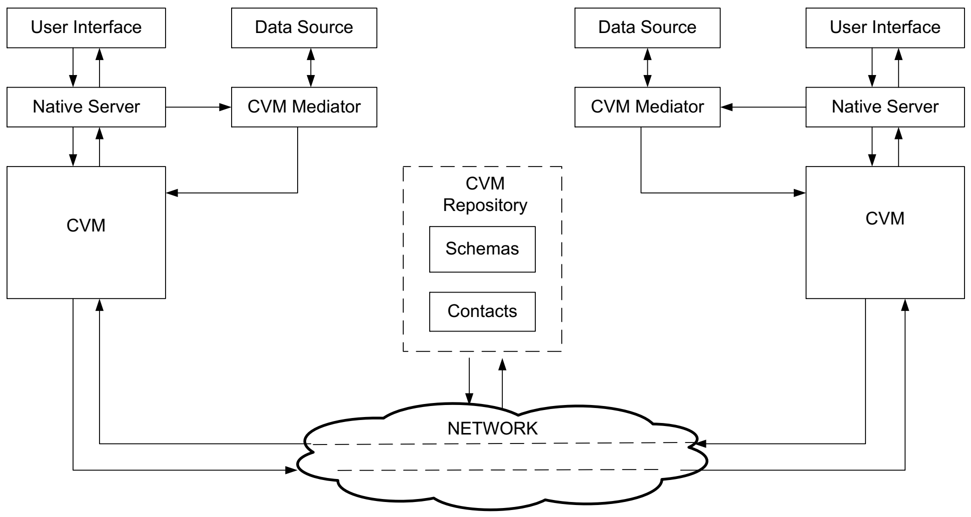


Fig. 3. Top-level architecture of CVM and associated components.

Table 2
Interface between CVM components

Layer	Provides	Signals

UCI	`msg login(userid, passwd)`
	`msg logout(userid)`
	`msg createConnection()`
	`msg addParticipant(connect_id, userid)`
	`msg addMedia(connect_id, media)`
	`msg send(connect_id, data)`
	`msg invoke(commService_id)`
	`msg store(commService_id)`
	`msg load(commService_id)`
		`notifyUserProfile(data)`
		`notifyCommServiceStatus(commServiceList)`
		`notifyConnectionException()`
		`notifyCommServiceID()`

SE	`msg login(userid, passwd)`
	`msg logout(userid)`
	`msg invoke(schema_inst)`
	`msg invoke(schema_data)`
		`notifyUserProfile(data)`
		`notifyMediaStatus(connect_id, media_id)`
		`notifyParticipantStatus(connect_id, participant_id)`
		`notifyException()`
		`notifySIStatus()`

UCM	`executeScript(string)`	`notifyUserProfile(data)`
		`notifyMediaStatus()`
		`notifyParticipantStatus()`
		`notifyException()`

NCB	`authenticate(userid, passwd)`
	`createSession(session_id)`
	`closeSession(session_id)`
	`addParty(userid)`
	`addMedia(mediaURI)`
	`applyPolicy(xmlString)`
		`notifyUserProfile(data)`
		`notifySessionStatus(session_id)`
		`notifySessionInvitation(sender_id, x-cml)`
		`notifyNetworkFailure()`

schemas and instances in G-CML[2] which are then automatically transformed to X-CML; (3) the *Schema Transformation Environment* – transforms an X-CML instance into a synthesis-ready X-CML instance or stores the X-CML model in the repository; (4) the *CVM Repository* – stores artifacts (e.g., grammar rules and CML schemas) to support the creation of CML schemas/instances; and (5) the *UCI-to-Synthesis Engine* Interface – provides a conduit for interaction with the synthesis engine. The User Interface provides a user-friendly GUI that allows users or developers to easily create communication services schemas or instances.

UCI provides several functions via the API of the UI adapter to the user interface as shown in Fig. 4. These functions are listed in the first row of Table 2. The first function `login` allows a user to be authenticated, and if successful the user's profile is retrieved from the central repository. The function `notifyUserProfile` passes a user's profile form the NCB to the UI after the user successfully logs in so that the appropriate data is made accessible to the user. The user profile consists of the contact list for the user and the set of communication schemas and schema instances accessible to the user.

The functions `createConnection`, `addParticipant`, `addMedia`, and `send`, provide the user interface with the required operations to allow the UI adapter to construct an X-CML schema instance. Once the schema instance has been created, the function `invoke(commService_id)` is called to start the negotiation process in the synthesis engine. The parameter `commService_id` is the unique identifier for a communication service being executed by the CVM. The `notifyCommServiceID` event signals the UI with the unique identifier associated with a given communication service. Note that although only one communication schema is being executed by the

[2] We developed two equivalent variants of CML: the XML-based (X-CML) and the graphical (G-CML). They are introduced in Section 5.

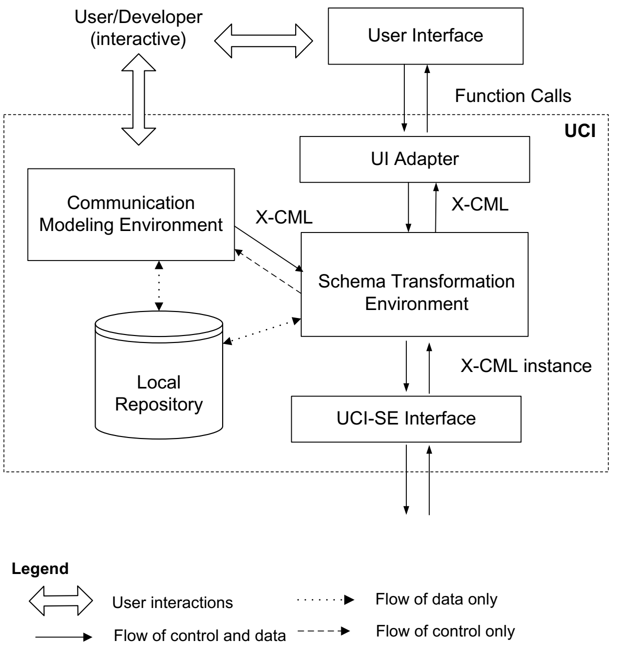


Fig. 4. Block diagram showing the detailed architecture of UCI.

CVM, it may consist of more than one application. The user/developer has no knowledge of the underlying schema instance running on the CVM to support the applications. However, the user interface needs to know the components of the applications so that they can be updated accordingly in the GUI presented to the user/developer.

The functions `load`/`store` provide the user (or developer) interacting with the user interface the ability to retrieve/save an application from/to the user's temporary profile in the local CVM repository. The functions return an acknowledgment or error message in the form of a message (`msg`). Note that when the user invokes the `logout` function his/her user profile is stored to the central repository. For example, in the scenario described in Section 2, Dr. Burke builds the communication application in the user interface, which makes the appropriate calls to the UI adapter to build the X-CML model for the schema instance. After the construction of the application is complete, Dr. Burke starts the application via the `invoke` function provided by the UCI.

In addition to constructing a communication schema instance, UCI checks the syntactic and semantic correctness of a communication schema instance by building an abstract syntax tree of the X-CML representation and traversing the tree to check type compatibility of the media types and the values of the fields of the attributes. Some schema instances require that their abstract syntax trees be annotated with meta-data associated with a communication schema (a template for a class of communication schema instances). This meta-data is defined using the communication modeling environment and stored in the repository.

During the execution of a schema instance UCI stores the current state of the schema instance, which includes the unique identifier for each application, the unique identifier for the participants and media in the communication, and the current state of the media being transmitted. The state is required during communication with the synthesis engine and is updated based on the signals from the synthesis engine (see Table 2). Any changes to the schema are passed to the user interface using the `notifyCommServiceStatus` event. If there is a problem with a particular connection the UCI signals the UI by sending the `notifyConnectionException` event. For example, if there is a problem in transmitting the images of the patient record (scenario from Section 2) to Dr. Sanchez or Dr. Monteiro (e.g., bandwidth of the connection is too small) the UCI signals the user interface of the problem. This status update allows Dr. Burke, the sender of the images, to send text describing the images.

4.2. Synthesis engine

The synthesis engine (SE) automatically transforms a declarative user-communication schema (specified using CML) to an imperative communication control script for deployment on the UCM. It is invoked by UCI via its *provides* interface functions, `invoke(schema_inst)` and `invoke(schema_data)`, as shown in Table 2. The SE is defined by its algorithms, processes, and techniques, which are used to generate the communication control scripts. These control scripts represent the network-independent control logic for user-level communication sessions.

The key challenge for the SE is complete automation, free of human intervention. In the rest of this subsection, we demonstrate how such automation can be achieved in the domain of user-centric multimedia communication services, at least for the functional aspects of the communication such as coordination of communication requirements and capabilities, as well as media delivery.

Given the role of the SE, we identify the following tasks that it must perform: (1) probe the local environment to align needs with communication capabilities and constraints and also determine the need for negotiation; (2) ensure the consistency of user-communication schema across participating end-points in a communication session; and (3) perform schema synthesis to obtain the communication control script to be deployed on the user-centric communication middleware.

The design of the SE supports a three-stage process: (1) *schema population*, during which the SE probes the environment to determine and account for local device communication capabilities and to handle communication constraints; (2) *schema negotiation* among participants of communication, to determine the feasibility of the desired communication and to ensure that all parties agree to a consistent communication schema; and (3) *schema synthesis*, during which the SE determines the needs of communication and *automatically* transforms the schema to a *communication control script* deployable on a user-centric communication middleware. Fig. 5 depicts the architecture of the SE. The

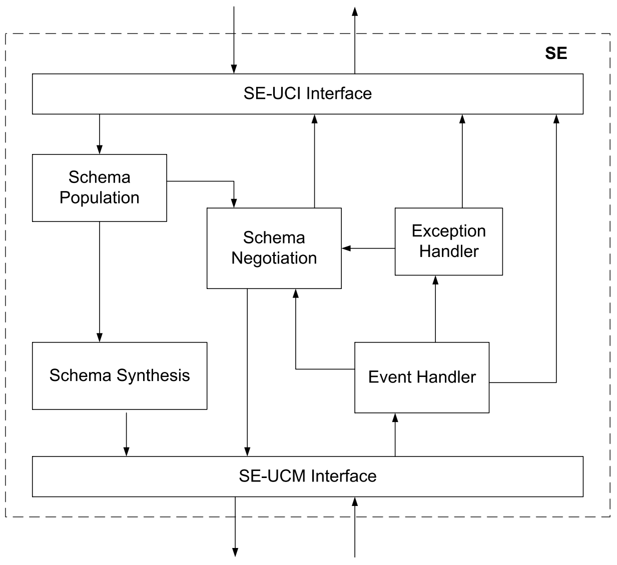


Fig. 5. Block diagram showing the detailed architecture of the synthesis engine.

arrows depict control flow. The SE also contains an event handler for external negotiation requests, media progress/delivery notifications, and exceptions. Such handling may involve re-negotiation or user notification/feedback.

4.2.1. Schema population

The first step in synthesizing the desired communication is schema population, which probes the local environment to align communication needs with local device capabilities and constraints. Schema population augments the communication schema instance with the communication capabilities of the local device such as the media types supported, including specific format (or subtype) information (e.g., real-media format of type video). The capability information is further enriched with type-specific information such as resolution and frame-rate of video or the bit-rate of audio supported. The populated schema instance is then aligned with the communication needs declared in the schema instance, employing user-feedback to resolve inconsistencies, if any. Schema population also determines the need to negotiate communication parameters with remote participants involved in the communication. The need for schema negotiation arises the first time a communication is initiated and whenever there is a modification to the current communication schema instance. Specifically, it is required in the following scenarios: (a) the initiator of a new communication instantiates the corresponding schema with remote participant information for the first time; (b) a participant in an ongoing communication adds or deletes a participant; and (c) a participant in an ongoing communication adds/deletes a medium type to the current schema. Addition or deletion of a participant requires re-negotiation to inform other participants of the change as well as to accommodate the communication capabilities of the new set of participants. Addition or deletion of a medium type requires re-negotiation to conform to the capabilities and preferences of the communicating participants. Note that the addition of a new instance of a medium-type (e.g., sending audio-file myfile.mp3) does not require re-negotiation as this addition will occur only after the "audio" medium-type has been negotiated. No new capabilities are required of the end participants.

4.2.2. Schema negotiation

Schema negotiation is required to determine the feasibility of the desired communication and to ensure the consistency of the communication schema instance across the participating end-points in a user-communication session. A communication schema instance defined by user A may require a video connection to user B. However, if B's device is not capable of video communication, this communication is not possible. In a multi-party communication scenario among A, B, and C, where A initiates the communication, C may not agree to communicate with B after it receive the invitation from A. Apart from negotiating the initial schema instance before actual communication starts, schema *re-negotiation* may also be required when a communication session is in progress.

Each participant in a communication session has a local copy of the schema instance. Any change to the schema made locally may require an update to the local schema instances at all participating end-points. If two users in a session are simultaneously altering their schemas, concurrency problems arise. The synthesis engine uses a modified non-blocking three-phase commit protocol Rangaswami et al. (2007) for schema synchronization.

Each schema instance change initiates a negotiation process, which proceeds in three distinct phases. The final phase is the commit phase. In *Phase 1*, the initiator reports the requested change to the schema instance to all remote participants, including any new participants being added, by sending the desired schema instance. In *Phase II*, the remote parties receive the changes and append their own *un-committed* changes, if any, to the schema instance. If this is the first time a schema instance is being negotiated or in case new participants are being added, the new participants also declare their device capabilities in the schema instance. Each remote participant sends this modified schema instance to the initiator. In *Phase III*, after the initiator receives the responses from all participants, all modifications from remote participants are merged. If the new schema instance differs from the original intent of the initiator, user feedback is employed to authorize the communication. The initiator then sends a final confirmation, either in the form of a consistent schema instance to be used for communication or to cancel the session.

Since multiple parties may initiate schema negotiation simultaneously, negotiation requests from remote parties are queued together with the locally generated negotiation requests in a *synchronized negotiation request queue*. These requests are handled in order to ensure the consistency of the append operations described above. Out of order requests at multiple nodes are automatically handled due to the mechanics of the negotiation algorithm, which would merge requested changes to create a consistent schema.

4.2.3. Schema synthesis

As shown in Fig. 5, the schema synthesis process is invoked either directly after schema population or after negotiation. Regardless of the path taken, the schema synthesis process is the same. Its purpose is to transform the declarative communication schema instance into an imperative communication control script, executable on the user-centric communication middleware (described in Section 4.3).

A schema instance for a communication session defines all device types and device instances that are part of the session, followed by the attributes of all participants, and the association between participants and device instances in the session. The synthesis algorithm is as follows:

(1) Obtain the difference (X-CML') between the current X-CML schema instance and the previous (already

synthesized) schema instance. If no previous schema instance exists, the entire new schema instance is used as X-CML'.

(2) Augment X-CML' with context information including session ID and connection ID for each new entity (e.g., new participant or new medium instance), if absent. Now the X-CML' is composed of one or more connection blocks.
(3) Create an empty communication control script. For each connection block in the X-CML', (i) for each connection, if this connection did not exist in the previous version of the schema, add to the script a command to create a new session that implements this connection; (ii) for each participant, add to the script a command for adding a participant to the corresponding session; and (iii) for each medium instance, add to the script a command for adding the medium instance to the corresponding session.
(4) Dispatch the communication control script to the UCM layer for execution.

The X-CML schema of any communication session defines all devices, persons, and associations of the session, in sequence as a tree. As the synthesis algorithm processes the X-CML as described above, it is evident that code for all features of the communication session will be generated.

In Section 6, we elaborate an actual instance of synthesis for the telemedicine communication scenario that we described in Section 2.

4.2.4. Event handler

The *event handler* of the SE handles system notifications, exceptions, or error conditions and dispatches the event to the appropriate subhandler. Remote negotiation requests are dispatched to the negotiation handler by adding them to the *synchronized negotiation request queue*. Exception conditions such as loss of communication with a specific participant or temporary loss in network connectivity are dispatched to the *exception handler*, which may either initiate a re-negotiation request to handle the exception or intimate the user via the UCI layer if the exception cannot be handled internally due to schema instance-specific constraints. Finally, communication status updates such as the amount of progress in media delivery are directly notified to the UCI layer.

The synthesis engine delivers four types of notifications to the UCI layer. The `notifyMediaStatus` and `notifyParticipantStatus` signals notify the UCI about media delivery and participant connectivity, respectively. The `notifySI Status` signal notifies the UCI about changes to the schema instance as a result of external changes due to other participants such as addition of new participant to an existing session or a change in capabilities of an existing participant, etc. Finally, the `notifyException` signals the UCI about exceptions such as lost network connection, when it cannot be handled internally.

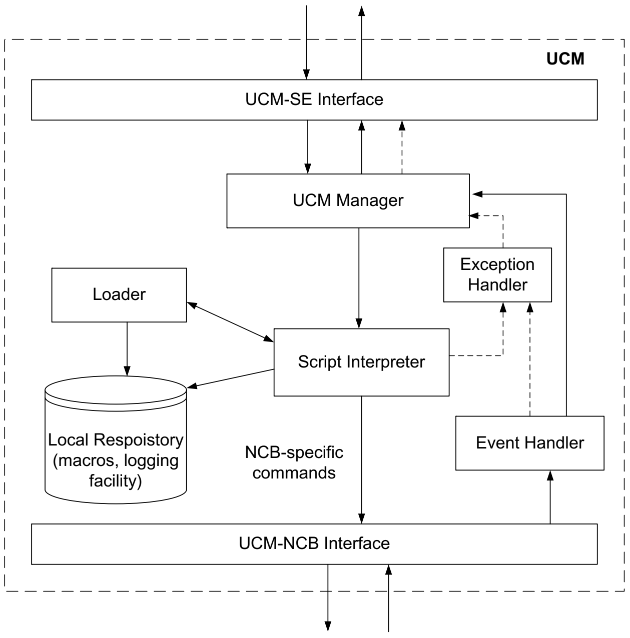


Fig. 6. Block diagram for the user-centric communication middleware.

4.3. User-centric communication middleware

UCM is responsible for executing the communication control script and for maintaining the states of user-level communication (as opposed to network level one). Fig. 6 shows the architecture of UCM. The components of the UCM include *UCM-SE interface* and *UCM-NCB interface* that exposes the *provides* and *signals* functions shown in Table 2, Row 3; *UCM Manager* that coordinates the activities of UCM; *Script Interpreter*, interprets the control script, load libraries and generates a sequences of calls to the NCB to realize communication; *Loader* loads the appropriate component macros required for enforcement of policies such as communication constraints, security properties and other QoS concerns; *Local Repository* stores the macros, current state and logs to support the operations of UCM; *Exception Handler* passes exceptions to the UCM manager to be handled; and *Event Handler*, handles events, including exceptions, generated by the underlying communication services provider, NCB.

The UCM provides one service to the SE, `executeScript(string)`, that takes a string consisting of the communication control script generated by the SE to be executed. The control script contains the control flow logic to either perform schema negotiation, or realize the communication schema instance. The UCM manager takes the control script and passes it to the Script Interpreter for translation and execution. During the translation process the control script is parsed and the Loader is invoked, if there are any script commands that correspond to macros required during execution. Although we have not yet incorporated the facilities to process policies such as communication constraints (e.g., if bandwidth is low then substitute video with images) or security properties (e.g., encrypt all patient data), we expect that such policies will be introduced in the future and a macro facility will be needed to support the enforcement of the policies. In the current prototype the Script Interpreter takes the control script generated by the SE and invokes the appropriate NCB commands. We provide an example of the code generated by the Script Interpreter for the scenario in Section 6.

The Exception Handler processes exceptions from both the Script Interpreter and the Event Handler. Possible exceptions from the Script Interpreter include `illegalScriptFormat` or `missingMacro`. These exceptions are passed to the UCM manager where a decision is made to interrupt the Script Interpreter and terminate execution or allow execution to continue. The Event Handler processes all the events coming from the NCB including the exceptions generated from the NCB (e.g., `networkFailure`, which is passed on to the SE via the UCM manager). Other events handled by the Event Handler include `notifyUserProfile(data)` containing the user profile after the user has successfully logged into the CVM repository and retrieved the user's schemas (see Fig. 3); `notifySessionStatus(session_id)` informs the UCM of the current status of a session related to a particular connection; and `notifySessionInvitation(sender, x-cml)` is an event that invites the user to join a communication session, the `x-cml` contains the schema for the communication.

The Local Repository stores the state for the communication application being executed. This state information includes the connection unique identifier and associated session unique identifiers; logs containing the policy macros that were executed and the exceptions that were raised during execution; and locations of data that needs to be sent on demand during the connection session.

4.4. Network communication broker

NCB's job is to manage network sessions. (It should be clear that each user session may result in many network and transport layer connections and connectionless communications.) Each participant of a session can multicast to all the other participants. The NCB API (detailed in Zhang et al., 2006) to UCM is both application- and network-independent, through which high-level communication tasks can be specified. A new session is created by invoking the `createSession` call provided by NCB, with a session ID, which maintains a unique association between each user and network sessions. NCB provides `addParticipant`, and `addMedia` services to UCM to dynamically add participants and media types in user sessions. The NCB interface allows an application to customize NCB behavior under specific network and system conditions, based on user or application preference. The interface, `applyPolicy`, takes as input an XML string which describes the policy for self-management. The NCB callback interface presented in Table 2 allows it to signal the status of the network, the status of the existing sessions, and a session invitation from a remote user (i.e., the new session will be created after the local user agrees to join the session).

NCB translates a high-level communication task into a series of operations that control the underlying networking facilities. It encapsulates and abstracts the heterogeneity of the network protocols and their interfaces. As illustrated in Fig. 7, the internal architecture of NCB is complex in that it coordinates both the control plane (i.e., signaling protocols negotiating the communication) and the data plane (i.e., transport protocols delivering media). The NCB core further includes modules such as *Session Management*, *Participant Management*, *Media Management*, and *QoS* and *Self-Management*. The current prototype implementation utilizes the JAIN SIP and the JMF library, and supports SIP and RTP as underlying networking protocols.

The communication messages between different NCBs following standard networking protocols may have their own notions of low-level network sessions. To encapsulate various *network sessions* (e.g., audio, video, etc.) within one user session, the NCB must internally maintain the mapping from the user-level session ID to the network-level session IDs of the underlying protocols. In the rest of the paper, the term "session" is used to denote a NCB user session, unless otherwise stated. In Fig. 7, only the modules above the dotted line are aware of user sessions, while all the modules below that line are responsible only for individual network-level sessions. The extensible design of the NCB can facilitate the integration of new communication features over heterogeneous network condition. We briefly describe and discuss each module as follows.

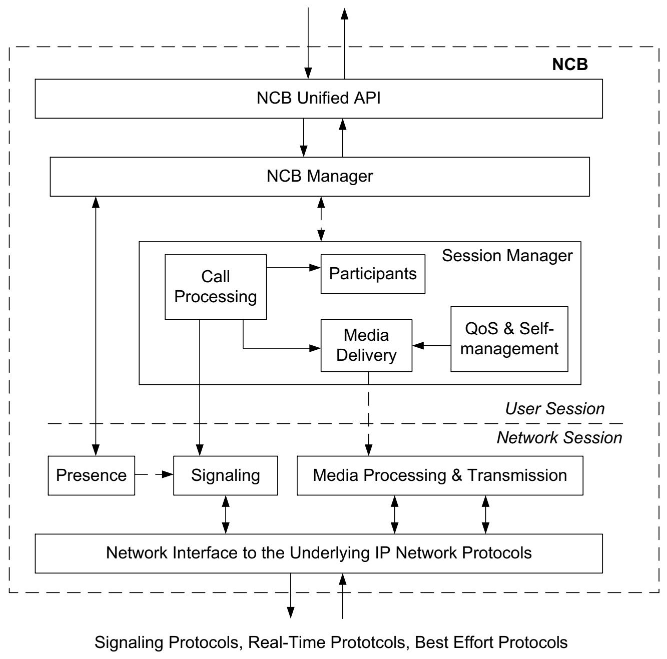

NCB
NCB Unified API
NCB Manager
Session Manager
Call Processing
Participants
Media Delivery
QoS & Self-management
User Session
Network Session
Presence
Signaling
Media Processing & Transmission
Network Interface to the Underlying IP Network Protocols
Signaling Protocols, Real-Time Prototcols, Best Effort Protocols

Fig. 7. Block diagram showing the detailed architecture of the network communication broker.

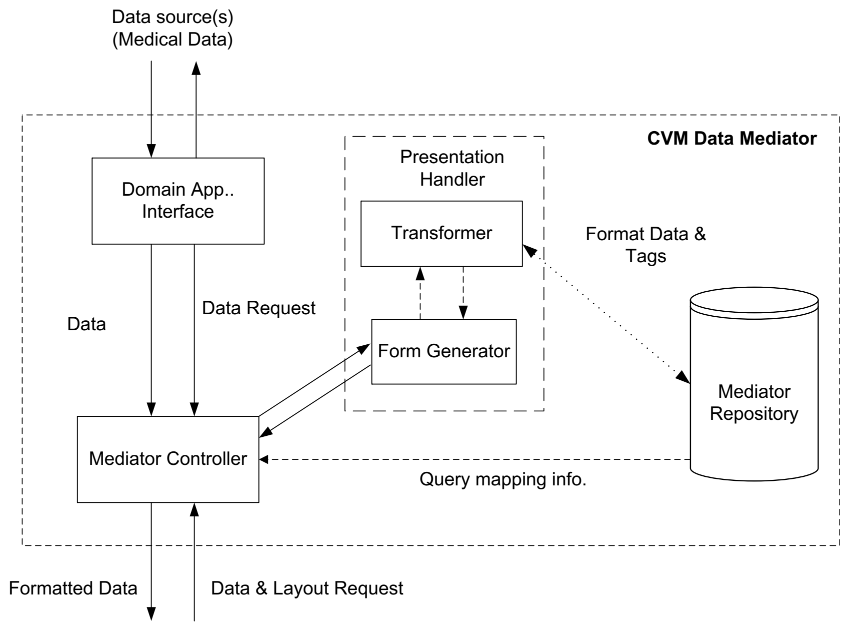


Fig. 8. Block diagram showing the architecture of the CVM data mediator.

The *NCB Manager* is responsible for the initialization and the configuration of the entire NCB middleware. For example, it maintains the signaling server information (IP address, etc.). Upon receiving an application request for creating a new session (at the caller side), or a signaling message INVITE (at the callee side) from a remote user negotiating a new conversation, NCB Manager creates a new Session Manager (see below) to handle the new communication session. The NCB Manager maintains the list of Session Managers for all active sessions.

A *Session Manager* deals with a single user session. Since the states associated with a session include the call status, the participants, and the media transfer, this module further delegates the tasks to the "Call Processing", "Session Participants", and "Media Delivery" submodules within the Session Manager. The Session Participants module keeps the list of participants of this session. The *Call Processing* module controls, at the level of user sessions, the logic of a session. It converts high-level control actions (such as "`addParticipant`") of a user session to low-level signaling operations, based on the underlying Signaling module, which carries out the basic signaling. It maintains the states of the user session, such as the mapping between the user session and individual SIP signaling sessions maintained by the *Signaling* module. The *Media Delivery* module manages, at the level of user sessions, the transfer of media in a session. It translates an "`addMedia`" call from the application into a number of internal operations. It first relies on the Call Processing module to negotiate transmission parameters (port numbers and encoding/decoding schemes) before the actual media transmission. It then pass the control to the "Media Processing and Transmission" module to actually transmit the media.

The *Media Processing and Transmission* carries out media transmission and reception. In addition, media will be pre-processed (e.g., encoding) at the sender side, and will be post-processed (e.g., decoding) at the receiver side. The *Signaling* module carries out the basic signaling operations according to the signaling protocols, such as registration, invite a participant, media type and parameter negotiation. The Signaling module encapsulates the signaling heterogeneity, such as different signaling protocols (SIP vs. H.323), with or without NAT traversal.

The *QoS and Self-Management* module autonomously monitors and adapts the behavior of the Media Delivery module. The self-management behavior of this module follows the high-level policies specified through the `applyPolicy` interface (see Table 2) as the guideline from upper-layer applications. For example, if the available bandwidth is low, depending on user/application preferences specified through high-level policies, this module can instruct the Media Delivery module to either (i) use encoding schemes that provide less resolution; or (ii) suspend video transmission in order to maintain high-quality voice communication; or (iii) slow down (by decreasing socket buffer sizes) file transfer for high-quality video/audio.

4.5. CVM data mediator

The CVM Data Mediator, shown at the top of Fig. 3, is an extension to the prototypical design presented by Deng et al. (2006). Although the mediator is external to the CVM, it provides an interface that allows the CVM to extract data from domain specific information systems to be sent to participants in a communication. The task of interfacing with other information systems is particularly important in the domain we are currently exploring, the healthcare domain (Hristidis et al., 2006).

The architecture of the CVM Data Mediator, shown in Fig. 8 consists of four major components: (1) the *Mediator*

Controller – accepts the data request from the user interface and coordinates the packaging of the data to be returned to the CVM; (2) the *Domain Application Interface* – formulates the query to be sent to the domain application, e.g., in the healthcare applications; (3) the *Presentation Handler* – processes the data returned from the Domain Application Interface and applies any security/privacy policies on the data content, the formatting of the data using the appropriate template, and generating the retrieval mapping data; and (4) the *Local Repository* – stores the templates, tags and mapping information used in the CVM data mediator. The data from the mediator is passed to the CVM using the X-CML data format. The mediator invokes the send function provided by the UCI.

5. Communication modeling language

In this section, we introduce our approach to communication modeling through an overview on the requirements for communication modeling and description of our communication modeling language.

5.1. Requirements

We have identified the following requirements for the communication modeling language: (1) *Simplicity* – be simple and intuitive; (2) *Network-independence* – be independent of network and device characteristics; and (3) *Expressiveness* – model the large majority of communication scenarios.[3] In order to refine the expressiveness requirement, we identify the following primitive communication operations:

(1) Establish a connection.
(2) Transfer a piece of data or a data stream.
(3) Add/Remove participants to a communication (which corresponds to the conferencing capability of current systems).
(4) Specify requested properties for a connection or a particular data transfer. These properties include quality of service, security, access rights, suggested handling of transferred data.
(5) Group the transferred data such that the receiving sides become aware of their association.
(6) Close connection.

Notice that the above operations are by no means an exhaustive list. We considered a much longer list but chose the above in order to build a minimal, intuitive and adequately expressive first version of CML. Other operators which we have considered but have postponed for later versions include communication constraints (e.g., if bandwidth is low then do not send video streams) and timing commands (e.g., transfer the sensor output every 5 s). The goal of CML as the first language in this area is to trigger the research community to start developing communication languages in order to eventually reach a well-accepted standard.[4]

```
1.userSchema ::= local connectionNT { connectionNT}
2.connectionNT ::= mediaAttached connection remote { remote}
3.local ::= person isAttached deviceNT
4.remote ::= deviceNT isAttached person
5.mediaAttached ::= { medium} { formNT}
6.deviceNT ::=  device deviceCapability { deviceCapability}
7.formNT ::= { formNT} { medium} | form
8.person ::=  personName_A personID_A personRole_A
9.device ::= deviceID_A
10. medium ::= builtinType_A mediumURL_A suggestedApplication_A
11. deviceCapability ::= builtinType_A
12. form ::= suggestedApplication_A action_A
13. action_A ::= "send" | "doNotSend" | "startApplication"
```

Fig. 9. EBNF representation of X-CML.

5.2. CML

We present an intuitive communication modeling language (CML) (Clarke et al., 2006) for modeling user communication requirements. We developed two equivalent variants of CML: the XML-based (X-CML) and the graphical (G-CML). The former is the version that CVM understands and processes, while the latter is the user-friendly graphical form. G-XML is analogous to the E–R diagram (Chen, 1976) in the database domain. In Clarke et al. (2006), we show how these two variants can be automatically converted to each other.

5.2.1. X-CML

For presentation purposes we have converted (and simplified) parts of this XML Schema to EBNF form (stripping out reference constraints that cannot be represented in EBNF), shown in Fig. 9, in order to explain the basic components of CML. Notice that Fig. 9 depicts an attribute grammar, where attributes are denoted by an "A" subscript, terminals by boldface and non-terminal by italics. Fig. 10 shows the hierarchy of built-in types (`builtinTypeA`) currently supported in the CML implementation in CVM. An example of the EBNF productions shown in Fig. 9 is as follows. Production 6 states that the *deviceNT* non-terminal is composed of an actual device (`device` terminal) e.g., PC, that has one or more capabilities (`deviceCapability` terminal) i.e., the types the device can process. In production 9 the actual device currently has one attribute a unique device identifier `deviceID`$_A$.

5.2.2. G-CML

Table 3 shows the graphical grammar for G-CML. Columns 2 and 3 of Table 3 show the non-terminal definitions

[3] Similar to SQL, which can express the large majority of data queries.

[4] In the same way as XQuery emerged by combining multiple earlier XML language proposals.

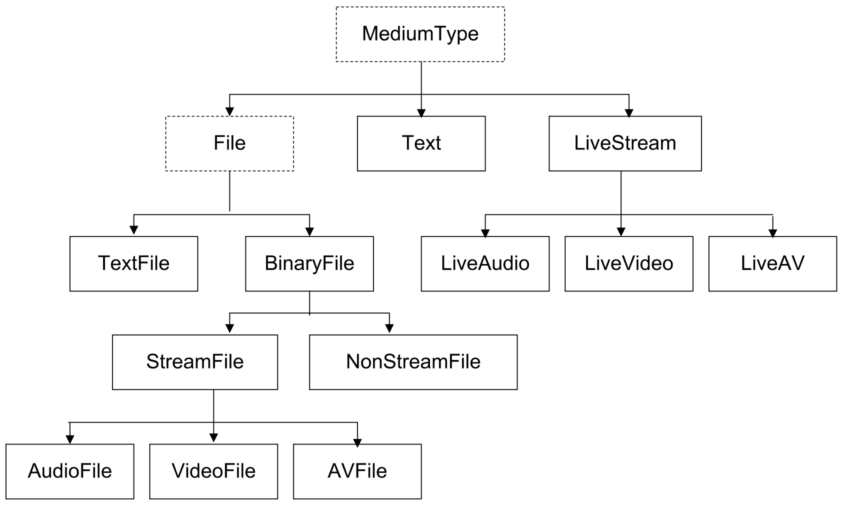


Fig. 10. Pre-defined (built-in) types for CML.

of G-CML, while Columns 4 and 5 show the terminal definitions. The number in the first column of Table 3 corresponds to the equivalent production in the EBNF representation of X-CML, shown in Fig. 9. For instance, the first row of Table 3 shows the structure of the *userSchema* non-terminal, which consists of the *local* non-terminal connected to one or more connection non-terminals (*connection*). To indicate that a symbol may be repeated we use the character "*" for zero or more repetitions and "+" for one or more repetitions. The right side of the first row in Table 3 shows the terminal for connection, which consists of a diamond shaped box with the label "connection". The remaining non-terminals and terminals in Table 3 can be described in a similar manner. G-CML uses a notation similar to E-R diagrams, however the semantics are different. The symbols used in G-CML can be classified into three categories, these include: (1) entities – `person`, `device`, `medium` and `form`; (2) relationships – `connection` and `isAttached`; (3) attributes – properties of the entities (e.g., `suggestedApplicationA`[5]). We have built a tool to support the construction of G-CML models using the Eclipse Modeling Framework (Wang et al., 2007). The tool guides the user when building G-CML diagrams in an intuitive drag-n-drop manner and automatically validates the model as it is being built.

5.2.3. CML for scenarios

Figs. 11 and 12 show the (instantiated) G-CML and X-CML representations for the scenario of Section 2, respectively. Since the frequency of exchanging data during communication is much higher than updating the communication's configuration (e.g., list of participants, device capabilities), we partition a CML instance into control part (configuration) and data part (exchanged media), as shown in Fig. 12. Sending the control part, only when a change occurs, leads to reduced processing and parsing delays. Note that in practice the user only needs to create the G-XML or X-CML schema once. Then for each instantiation of the schema, the user only needs to assign values to the entities of the schema, e.g., assign names to participants (persons) and media. A *connection* in CML is a user session, and is defined as a communication among a group of participants, where exchanged data is by default broadcasted to all participants. In addition to the local side, a connection contains a set of media (`mediaAttached`) currently transferred in the connections (user-communication session) and a set of remote participants (remote). Both local and remote participants are associated with a communication device (e.g., PC, cell phone), which is associated by a set of capabilities (`deviceCapability`).

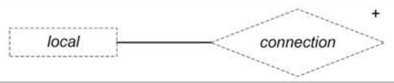
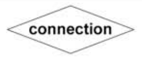
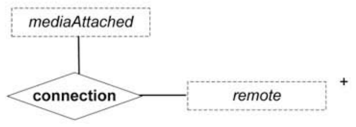
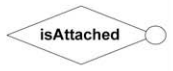
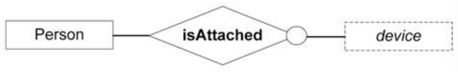
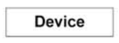
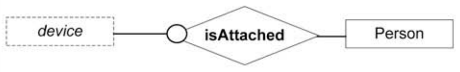
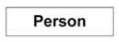
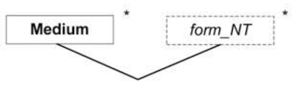
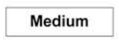
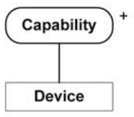
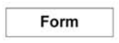
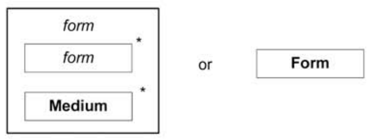
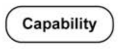
Notice that the specific characteristics of a device, such as its type (e.g., PC or cell phone) or its connection type to the network (e.g., IP or cellular), are not defined nor required. The reason is that CML operates on an abstraction of the underlying network and devices. We assume there is a single virtual device per person, which has the union of the capabilities of all physical devices attached to the user.

A medium is a data piece or data stream, like a Word document or a live video feed respectively. A medium has a type which is one of the pre-defined types supported by the system, a `mediumURL` that contains the location of the medium (a file location for a data piece or a port for a data stream), and a `suggestedApplication` which defines the application that can be used to view or process a medium (e.g., `Powerpoint` for ppt files).

Finally, composite data are represented using forms in CML, which are nested structures that contain media as well as user-defined attributes (e.g., media with common `suggestedApplication` can be grouped together in a form). For example, it is common in medical scenarios to

[5] There are attributes associated with entities that are not shown as special symbols in the diagram but are captured as textual notation in the associated symbols, e.g., `personRole` of Person. We chose this design option to avoid having diagrams that are cluttered with symbols.

Table 3
Grammar for G-CML

No.	Name	Non-Terminal Production	Name	Terminal
1	*userSchema*		**connection**	
2	*connection*		**isAttached**	
3	*local*		**device**	
4	*remote*		**person**	
5	*mediaAttached*		**medium**	
6	*device*		**form**	
7	*form*		**deviceCapability**	

* *indicates zero or more occurrences of the symbol.*
+ *indicates one or more occurrences of the symbol*
The link connecting the symbol also repeated

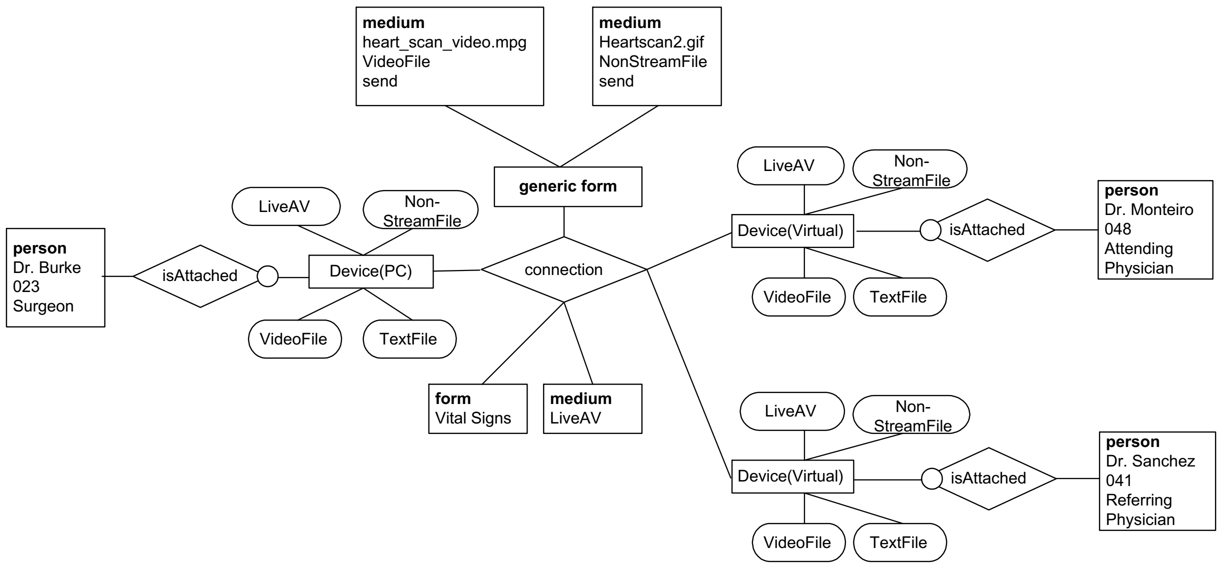


Fig. 11. G-CML example for our scenario.

```
Control:

<?xml version="1.0" encoding="ISO-8859-1"?>
<userSchema>
<connection connectionID = "connection1">
        <device deviceID = "001" isVirtual = “false”>
                <deviceCapability>LiveAV</deviceCapability>
                <deviceCapability>VideoFile</deviceCapability>
                <deviceCapability>TextFile</deviceCapability>
                <deviceCapability>NonStreamFile</deviceCapability>
        </device>
        <device deviceID = "002" isVirtual = “true”>
                <deviceCapability>LiveAV</deviceCapability>
                <deviceCapability>VideoFile</deviceCapability>
                <deviceCapability>TextFile</deviceCapability>
                                <deviceCapability>NonStreamFile</deviceCapability>
        </device>
        <device deviceID = "003" isVirtual = “true”>
                <deviceCapability>LiveAV</deviceCapability>
                <deviceCapability>VideoFile</deviceCapability>
                <deviceCapability>TextFile</deviceCapability>
                <deviceCapability>NonStreamFile</deviceCapability>
        </device>
</connection>
<mediumType mediumTypeName = "LiveAV">
<formType formTypeName = "Generic" suggestedApplication =”default” voiceCommand =”” action = "send">
        <mediumDataType>AudioFile</mediumDataType>
        <mediumDataType>NonStreamFile</mediumDataType>
        <mediumDataType>TextFile</mediumDataType>
</formType>
<formType formTypeName = "Vital Signs" suggestedApplication =”default” voiceCommand =”” action = "send">
        …
        <!--This is a complex form consists of too many items to be presented>
</formType>
<person personName = "Dr. Burke" personID = "023" personRole = "Surgeon"/>
<person personName = "Dr. Sanchez" personID = "041" personRole = "ReferringDoctor"/>
<person personName = "Dr. Monteiro " personID = "048" personRole = "AttendingDoctor"/>
<isAttached personID = "023" deviceID = "001"></isAttached>
<isAttached personID = "041" deviceID = "002"></isAttached>
<isAttached personID = "048" deviceID = "003"></isAttached>
</userSchema>


Data:
<?xml version="1.0" encoding="ISO-8859-1"?>
<data connectionID="connection1">
<form formDataType =”Generic” formID = "baby jane" suggestedApplication =”default” voiceCommand =”” action =
"send">
        <medium mediumDataType = “VideoFile " mediumName = "heart_scan_video.mpg"
                mediumURL = "http://www.fiu.edu/heart_scan_video.mpg " action = "send"/>
        <medium mediumDataType = “NonStreamFile " mediumName = "heartscan2.gif"
                mediumURL ="http://www.fiu.edu/heartscan2.gif" action = "send"/>
</form>
<form formDataType =”Vital Signs” formID = "PatientHospitalization866803" suggestedApplication =”default”
voiceCommand =”” action = "send">
        …
</data>
```

Fig. 12. X-CML example for our scenario.

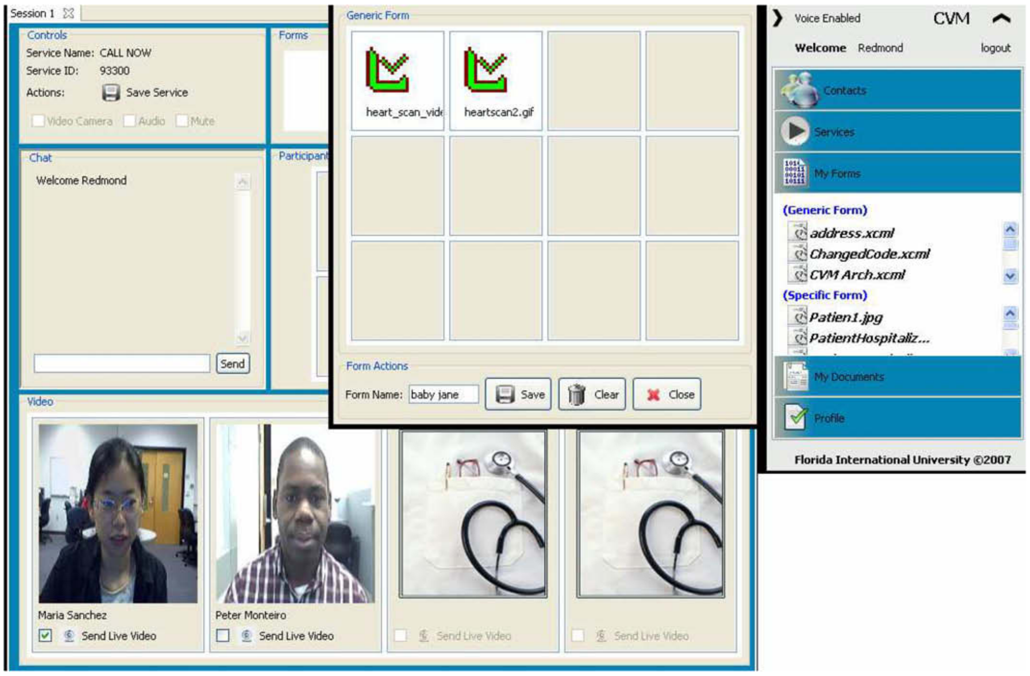
require transferring complex medical data consisting of multiple simple media (e.g., a page in a patient medical record). Form has an action attribute which defines a default action that is performed on this medium during form transfer. The actions “`send`” and “`doNotSend`” translate into transfer automatically the medium in the form or wait for the user to request the medium, respectively. The “`startApplication`” orders the system to open the suggested application of the medium once transferred.

6. Prototype implementation

A CVM prototype has been implemented using the following technologies. A Web-based user interface has been deployed with the Opera 8.5, a voice-enabled browser. This prototype enables creation, modification, and use of communication schema instance using voice commands. The technologies used at the browser side are HTML, Javascript for dynamic effects and the program logic, and XHTML+Voice[6] for voice conversation. Part of the Javascript code uses AJAX[7] technology (Asynchronous JavaScript and XML) to make web requests and responses in the background, without having to refresh the web pages. The rest of the CVM layers, including another user interface (currently being the most updated version of our UI), are implemented in Java, deployed on each node. JAIN SIP and Java Media Framework (JMF) are used for control and data communications, respectively. Finally, we used SER (SIP Express Router) server for registration and presence and Asterisk for connection to PSTN and audio mixing.

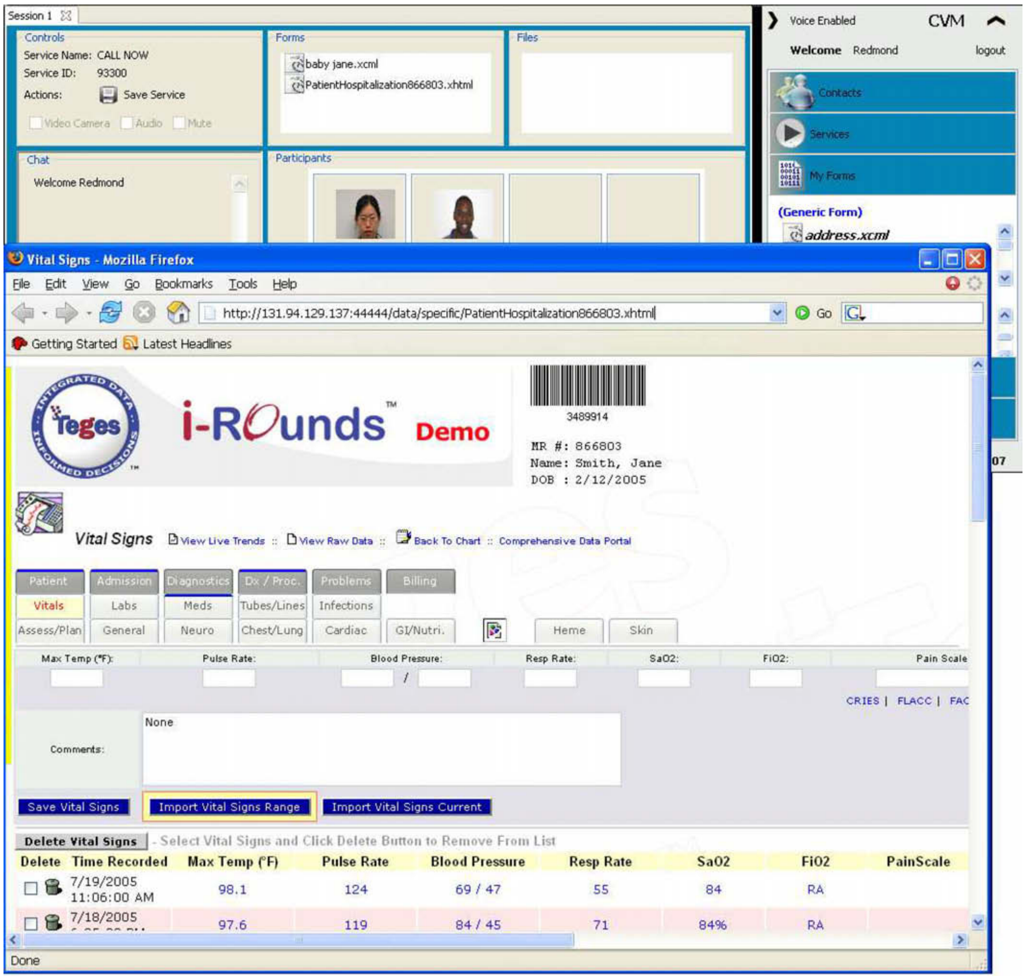
Figs. 13 and 14 show two screenshots form the current prototype with the Java-based user interface (for Web-based UI, please refer to Deng et al., 2006). These screen

[6] http://www.voicexml.org/specs/multimodal/x+v/12/.
[7] http://java.sun.com/developer/technicalArticles/J2EE/AJAX/.



Service Name: CALL NOW
Service ID: 93300
Actions: Save Service
Welcome Redmond
Form Name: baby jane
Maria Sanchez
Peter Monteiro

Fig. 13. Screenshot of CVM prototype showing an overview of active communications.


Session 1
Controls
Forms
Files
Chat
Participants
Voice Enabled
CVM
Welcome Redmond
logout
Contacts
Services
My Forms
(Generic Form)
address.xcml
Vital Signs - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Getting Started Latest Headlines
Teges INTEGRATED DATA INFORMED DECISIONS
i-ROunds™ Demo
3489914
View Live Trends :: View Raw Data :: Back To Chart :: Comprehensive Data Portal
Patient Admission Diagnostics Dx / Proc. Problems Billing
Vitals Labs Meds Tubes/Lines Infections
Assess/Plan General Neuro Chest/Lung Cardiac GI/Nutri. Heme Skin
CRIES | FLACC | FAC
Save Vital Signs
Import Vital Signs Range
Import Vital Signs Current
Delete Vital Signs
Done

Service Name: CALL NOW
Service ID: 93300
Actions: Save Service
baby jane.xcml
PatientHospitalization866803.xhtml
Welcome Redmond

http://131.94.129.137:44444/data/specific/PatientHospitalization866803.xhtml

MR #: 866803
Name: Smith, Jane
DOB : 2/12/2005

Vital Signs

Max Temp (°F): Pulse Rate: Blood Pressure: / Resp Rate: SaO2: FiO2: Pain Scale

Comments: None

- Select Vital Signs and Click Delete Button to Remove From List

Delete	Time Recorded	Max Temp (°F)	Pulse Rate	Blood Pressure	Resp Rate	Sa02	Fi02	PainScale
☐	7/19/2005 11:06:00 AM	98.1	124	69 / 47	55	84	RA	
☐	7/18/2005	97.6	119	84 / 45	71	84%	RA	

Fig. 14. Screenshot of CVM prototype showing integration of CVM with the iRounds system.

shots are taken from Dr. Burke's perspective, who initiated the communication application in the scenario presented in Section 2. The user interface of CVM can be divided into two parts:

(1) The right side is the *CVM dashboard* which consists of: *Contacts* – user's contact list; *Services* – all available communication services; *My Forms* – list of previously created forms; *My Documents* – refer to recently used files; and *Profile* – the user's personal information.
(2) The left side panel is the communication application space and it is divided into six composites. It includes: *Controls* – the communication service information; *Chat* – the space of exchanging instant message; *Forms* – refer to shared forms; *Files* – refer to shared files; *Participants* – remote persons involved in the current communication; and *Video* – consisting of live video stream windows for all remote persons in the current communication.

An additional window will pop up for the construction of forms. It can be a *Generic Form* window or a *Specific Form* window. Fig. 13 shows the screenshot of the CVM during the construction of a *generic* form. The three doctors in the scenario are having a three-way video conference. Live video streams from Dr. Sanchez and Dr. Monteiro are shown in the bottom left corner of the communication application space. The video streams are displayed in two windows in the *Video* composite. Dr. Burke is constructing a generic form "`baby jane`" in the Generic Form window. The Generic Form window contains two media entries a video file "`heart_scan_video.mpg`" and an image "`heartscan2.gif`".

Fig. 14 shows the screenshot of sharing a *specific* form generated from the iRounds[8] system. There are two items in the Form composite on top of the CVM side panel. One is the generic form "`baby jane.xcml`" shown in Fig. 13; the other is a specific form "`PatientHospitalization866803.xhtm`" shown in Fig. 14. The content of the specific form is shown in the "`Vital Signs`" window in the foreground of Fig. 14. The CVM Mediator retrieves the information from the iRounds system for "`Jane Smith`" and combines it with a pre-defined layout template to generate the specific form.

The UCI modifies the schema instance to reflect the above changes and invokes the synthesis engine. The synthesis engine generates the following script after a process of negotiation is completed:

```
createConnections(''connectionl'');
addParticipants(''connectionl'', ''Dr. Sanchez, Dr. Monteiro'');
sendSchema(''connectionl'', ''Dr. Burke'', control-xcml, data-xcml);
addMedia(''connectionl'', ''LiveAV'');
sendForm(''connectionl'', ''baby jane'', "http://www.cis.fiu.edu/heart_scan_video. mpg; http://www.cis.fiu.edu/heartscan2.gif");
sendForm(''connectionl'', ''PatientHospitalization866803'','''');
```

The above script is delivered to the UCM using the `executeScript()` of UCM *provides* interface (as presented in Table 2). The UCM then invokes appropriate NCB functions to accomplish the actual operations requested by the user. Following is an outline of the logic generated and executed by UCM interpreter for the control script shown above:

```
/* createConnections(''connectionl''); */
sid = ''sl'' /*create a UCM session ID shared with NCB*/
map(''connectionl'', sid) /*maps connectionID to sessionID*/
if(!ncb.createSession(sid)) /*calls NCB to create session*/
    throw noSessionException
/*addParticipants(''connectionl'', ''Dr. Sanchez, Dr. Monteiro''); */
sid = getSessionID(''connectionl'')
store participantList/*list contains ''Dr. Sanchez, Dr. Monteiro''*/
if (sid==null)
    throw noSessionException
for all p ∈ participantList
begin
    if( !(ncb.addParty(sid, p)))
        throw partyNotAddedException(p)
end
/*sendSchema(''connectionl'', ''Dr. Burke'', control-xcml, data-xcml)*/
sid = getSessionID(''connectionl'')
participantList = getParticipantList (''sid'')
if (sid==null)
        throw noSessionException
if !(control_xcm==null)
    if( !(ncb.sendSchema(sid, ''Dr. Burke'', participantList, control_xcml)))
        throw controlSchemaNotSentException
if( !(datel_xcm==null))
    if( !(ncb.sendSchema(sid, ''Dr. Burke'', participantList, data_xcml)))
        throw dataSchemaNotSentException
/*addMedia(''connectionl'', ''LiveAV'')*/
sid = getSession(''connectionl'')
if (sid==null)
    throw noSessionException
if !(ncb.sendMdia(sid, ''LiveAV'', null))
    notify SE.unavailableMeida()
```

[8] i-Rounds is an integrated clinical information system developed by Teges and currently being used in Miami Children's Hospital.

```
/*sendForm(‘‘connection1’’, ‘‘baby jane’’, http://
www.cis.fiu.edu/heart_scan_video.mpg,”
    “http://www.cis.fiu.edu/heartscan2.gif”);*/
sid = getSession(‘‘connection1’’)
if (sid==null)
    throw noSessionException
formURL = getFormURL(‘‘baby jane’’)
if (formURL==null)
throw formNotFoundException
/*Check if the form is a specific form*/
if (formURL.type==specific){
    if   (!ncb.sendMedia(sid,   FileTransfer,
formURL))
        throw mediaNotSentException
}
else{
/* Form is generic. URLString contains “http://
www.cis.fiu.edu/heart_scan_video.mpg,
    http://www.cis.fiu.edu/heartscan2.gif”*/
mediumURLList = URLString.Tokenize()
/*Extracts each mediumURL from the list*/
while (mediumURLList.hasNext())
    if(!ncb.sendMedia(sid,      FileTransfer,
mediumURLList.next()))
        throw mediaNotSentException
}
/*The sendForm code is repeated for the specific
form:
    sendForm(‘‘connection1’’, ‘‘PatientHospi-
talization866803’’, ‘‘’’); */
```

We have tested our prototype implementation with several other case studies both in general purpose applications such as multimedia conferencing and in domain specific applications such as Telemedicine. Our Telemedicine scenarios have been provided by our partners at Miami Children Hospital.

Limitations. The current version of the CVM prototype supports the design presented in Section 4. However, the following functionality have not yet been implemented: policies to support QoS, adaptive management, reliability, security, and access control.

7. Prototype evaluation

To show the effectiveness of synthesis engine, we obtain an estimate of the reduction in development time (and consequently, development cost) for applications with comparable functionality using two development approaches. The two approaches that were compared included the traditional stovepipe approach and the automatic synthesis approach provide by the CVM. The applications included (1) a chat application, (2) a person-to-person voice call, and (3) a person-to-person video communication service. Applications (2) and (3) used JMF (Java Media Framework API, 2005) and JAIN-SIP (JAIN-SIP, 2006) technologies and were developed by our team. Table 4 summarizes these applications, their code sizes in lines of code (loc), their estimated development time using the traditional approach, and also shows the approximate specification and synthesis time for generating these applications using the CVM prototype. Row 1 of Table 4 contains the data for a multi-user test chat application, the application using the traditional approach is Jabber-1.4.2 (jabber07, 2007), lines of code is 5528, estimated time of development is 2 months, time to develop the application using CVM is less than 5 min. Rows 2 and 3 represent the data on the application that were developed by our team for the person-to-person voice call, and person-to-person video conference respectively. To estimate the development time by a trained programmer, we used the study of Ferguson et al. (1997), whose findings reveal approximately 2500 lines of code per month per programmer.

The numbers in Table 4 demonstrate the significance of our approach. Service creation time is reduced by several orders of magnitude. Even if we assume that only 25% of the code contributes to the functional aspects of the software, improvements in development time are still over two orders of magnitude. Further, the automated process introduces fewer bugs into the code-base, improving software reliability. This underlines the importance of using automated processes for synthesizing communication applications rather than follow traditional design and development.

To evaluate the time required for the actual synthesis process, we deployed CVM to seven machines (desktops and laptops) in a combination of wired and wireless local area network. Ten demo users were created and used to represent seven users communicating with each other. To verify the correctness of the synchronization protocol within the negotiation process, we initiated simultaneous modifications to the schema instance at different sites and verified the absence of any inconsistencies in the schema instances at the various end-points automatically over numerous iterations. In addition, we instrumented the synthesis engine to obtain the time required to perform schema synthesis. The plot in Fig. 15 shows the average time in seconds required for synthesis including the negotiation/re-negotiation stages when there are 2–7 participants present in a communication session. The results of these

Table 4
Reduced development time compared to traditional design and development

Application type	Application	loc	Estimated development time	Spec/Synthesis time (min)
Multi-user text chat	Jabber-1.4.2	5528	2 months	<5
Person-to-person voice call	Custom	9478	4 months	<5
Person-to-person video communication	Custom	16,784	7 months	<5

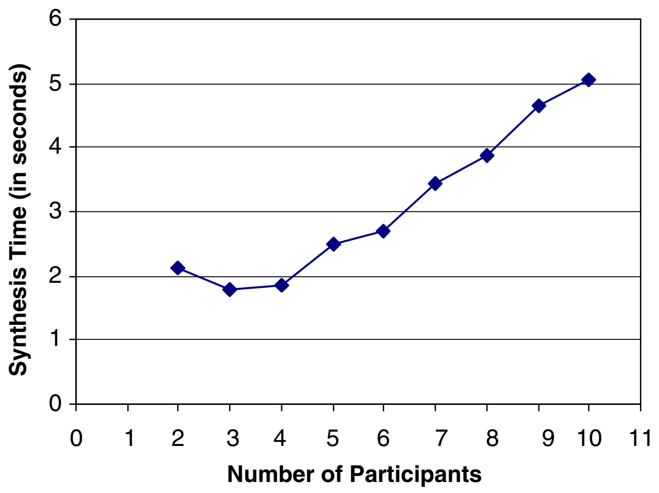


Fig. 15. Average time required for negotiation.

experiments show that the synthesis process scales linearly with the number of participants, and the process itself is dominated by the schema negotiation time. Higher numbers are not depicted due to lack of experimental infrastructure; however, we do not envision any issues limiting the linear scaling for larger participant-sets. The experiment demonstrates the practicality of schema synthesis process with a distributed negotiation algorithm; the negotiation time, which dominates the overall synthesis time incurs an acceptable delay.

8. Related work

There is a plethora of related research that addresses the individual processes and artifacts used in the various components of the CVM. However, not much has been published on how such components can be combined to provide flexible, user-centric and on-demand communication solutions. There are a number of off-the-shelf communication applications such as Yahoo! Messenger and MSN Messenger. We are also aware of several companies' efforts to integrate various tools into comprehensive communication solutions. The development approach that these products are based on dictates that none of them possesses the flexibility, on-demand, and user-centric communication solutions addressed in this paper. For example, it would be a tall order to adapt any of these tools to a comprehensive telemedicine application.

In the rest of this section, we have divided the related work into three major categories and discussed how CVM relates to them.

8.1. Model-driven engineering

The CVM approach shares some common traits with the concept of model-driven engineering (Bettin, 2004; Balasubramanian et al., 2006, 2004). In contrast to general-purpose model-driven development, automatic generation of communication services is feasible in CVM for two reasons: First, CVM is restricted to the scope of communication services and does not bear the complexity of generating general-purpose applications. The complexity of communication logic can be carefully regulated through the design of the schema modeling language. Second, CVM utilizes communication middleware components (e.g., those of ACE Schmidt and Huston (2002)) and server-side architectures (e.g., Bond et al., 2004) as building blocks to generate communication applications. Such existing components encapsulate procedures, patterns, and algorithms governing basic communication services (e.g., session establishment of person-to-person voice call, transmission of an image file, and real-time video streaming), which are well understood. The role of CVM is limited to the identification and composition of such components (McKinley et al., 2004).

More specifically, Heckel and Voigt (2004) describe how models in UML are transformed into BPEL4WS using the concept of pair grammars. We use a similar approach in the UCI but our modeling language G-CML is far more restrictive than UML and hence far more manageable and its synthesis can be automated. The implementation of the visual model in the UCI is based on the work by Costagliola et al. (2004). Costagliola et al. provide a framework that allows the user to define a visual language, create graphical models, validate these models and convert the models into strings of another language. The work in Balasubramanian et al. (2006) generates code from models using tool suites for specific application domains that were developed using a generic modeling environment. In our work, a generic SE generates control scripts from a CML description of communication logic, with restricted utility to the communication domain.

8.2. Communication middleware

There has been extensive work on communication middleware. Our work used many of the principles presented by Schmidt (1997), including using patterns and frameworks to alleviate complexity associated with a growing range of multimedia data types, traffic patterns, and end-to-end QoS requirements. Schmidt explored common pitfalls of developing communication software, including limitations of low-level native OSs and APIs and the limitations of higher-level middleware. The UCM and NCB components of CVM are designed exactly to avoid these pitfalls.

The existing protocol stacks may not be always suitable to take advantage of advanced transmission technologies and high-speed networks. Geppert and Rößler (1996) discussed how communication architectures could be made more flexible by automatically configuring communication subsystems based on a specification of desired target service. In the NCB we use a similar approach.

Stiller et al. (1999) described the Da CaPo++ system as an end-system middleware for multimedia applications adaptable to the application needs. The authors claimed that Da CaPo++ automatically configures suitable communication protocols, provides an efficient runtime support, and offers an easy to use object-oriented API, which shares some common traits with the low layers of UCM components. UCM design also leverages the concept of adaptive and reflective middleware, such as ACE and Ensemble, to provide self-management using only a high-level guideline. ACE (Schmidt and Huston, 2002) is a real-time C++ framework that wraps OS services and provides a variety of communication-related patterns. Ensemble van Renesse et al. (1998) is a groupware communication toolkit, which enables insertion of detectors in protocol graph. These detectors can trigger dynamic adaptation by distributing a new protocol-graph specification to all involved participants using a reconfiguration protocol.

JAIN-SIP (2006) is a standardized Java interface to SIP. Java Media Framework API (2005) is a library for audio and video communication. The low-level APIs of these communication libraries are still significantly complex to use, and far less usable than the user-centric session of UCM. The Java Telephony API is a high-level API for traditional telephony applications. They do not support next-generation multimedia communication applications with sophisticated business logic. Zave et al. (2004) discusses open software architectures for IP-based voice communication. Parlay (The Parlay Group, 2007) is an API for rapid creation of telecommunication services. These frameworks mostly address the server-side architecture and service creations. The server-side architecture has different concerns than the client-side middleware, which is the focus of UCM. In contrast to traditional telephone networks, in IP networks, end-hosts are capable of sophisticated communication logic. We note that the CVM principle of separating policy from mechanism has been popular in the operating systems community for several decades (Levin et al., 1975).

8.3. Computing middleware

Common computing middleware like CORBA (Vinoski, 1998; Vinoski, 1997), Java RMI (Satoshi, 1997), and DCOM (Chung et al., 1997) define higher-level distributed programming models as a set of reusable APIs and mechanisms, allowing developers to request services provided by target objects transparent of their location, programming language, OS platforms or communication protocols (Raj, 1998; Chung et al., 1997; Emmerich, 2000; Emmerich and Kaveh, 2002). Java/RMI allows Java developers to invoke object methods and execute them on Java Virtual Machines. Using JavaRMI, entire objects can be passed and returned as parameters. CORBA is a distributed systems technology that provides a higher-level, object-oriented interface on top of the basic distributed computing services. DCOM is a distributed extension to the Component Object Model (COM). Similar to CORBA, DCOM supports heterogeneous programming languages, but unlike CORBA and Java RMI, DCOM supports only Windows-based platforms.

Such computing middleware provides only a general model as a basis for developing distributed application, which serves to achieve application requirements in a distributed environment. The purpose is to eliminate the difference of using remote objects and local objects. However, the focus of their work is not on the communication process itself (Object Management Group, 1994). The high-level work flow and basic primitives of communication are not addressed in these paradigms. Even the simplest communication scenario, such as making a phone call, might go beyond the capability of CORBA, which does not inherently support APIs for real-time, interactive voice communications (Schmidt and Kuhns, 2000). Many issues such as, communication quality constraints, exchanged data types, and security definition during communication may arise to be implemented in the real communication services. Since these distributed object paradigms always ignore the location of target objects, it is hard to capture these requirements in an intuitive way. CVM is constructed to support basic communication activities and enable users to define variant communication models to capture the process of user communication. CORBA, JavaRMI, DCOM, and the like are not designed to solve these problems.

In CVM, an entire communication application is expressed using a high-level communication model, which can be rapidly realized. A communication schema provides higher-level of abstraction for capturing real world communication scenarios than the comparatively lower-level object-oriented model used in CORBA, JavaRMI, and DCOM. In this way, developers can create communication application models without the need to deal with the details of logical sequence of object method invocations.

9. Conclusion

We have presented CVM for on demand declaring, synthesizing and delivering communications services. We have discussed its architecture, supporting modeling language, components, algorithms, interfaces, and prototypical implementation. We discussed how CVM allows users to rapidly build and execute communication schemas to provide communication solutions across different application domains. It would be a misconception, however, to assume that an end-user needs to know modeling before they can use the CVM. For most end-users (e.g., a doctor), the modeling aspect will be hidden, because the schemas they use will be packaged as pre-defined services by their service providers or their organizations (e.g., a hospital).

Several classes of issues including security and performance at different layers of CVM are not addressed in this paper. A number of useful features can also be added.

Robust and effective solutions to these issues require further study, which represent exciting and interesting research topics. We argue, however, CVM represents a new paradigm for structuring and delivering communication solutions and services, which are far more effective than the current ways of development. In fact, the unique architectural traits of CVM allow new components and features to be seamlessly added as they become available. As such, CVM can serve as a communication service framework, which can be built upon and incrementally improved by the collective wisdom of the research community.

Acknowledgements

This work was supported in part by the National Science Foundation under grant HRD-0317692. We thank Eric Johnson, Andrew Allen, Yali Wu, Mario Lorenzo and Marylurdys Hernandez and for their participation in CVM prototype implementation.

References

Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S., 2006. Developing applications using model-driven design environments. Computer 39 (2), 33–40.

Bettin, J., June 2004. Model-driven software development: an emerging paradigm for industrialised software asset development. Technical Report, SoftMetaWare. <http://www.softmetaware.com/whitepapers.html>.

Bond, G.W., Cheung, E., Purdy, K.H., Zave, P., Ramming, J.C., 2004. An open architecture for next-generation telecommunication services. ACM Trans. Inter. Tech. 4 (1), 83–123.

Burke, R.P., White, J.A., 2004. Internet rounds: a congenital heart surgeon's web log. Sem. Thoracic Cardiovasc. Surg. 16 (3), 283–292.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1998. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, New York.

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J.D., Widom, J., October 1994. The TSIMMIS project: integration of heterogeneous information sources. In: Proceedings of IPSJ Conference. IEEE, New York, pp. 7–18.

Chen, P.P., 1976. The entity-relationship model: toward a unified view of data. ACM Trans. Database Syst. 1 (1), 9–36.

Chung, E., Huang, Y., Yajnik, S., Liang, D., Shih, J.C., Wang, C., Wang, Y., 1997. DCOM and CORBA side by side. Technical Report, Microsoft DCOM Technical White Paper. <citeseer.nj.nec.com/chung97dcom.html>.

Clarke, P.J., Hristidis, V., Wang, Y., Prabakar, N., Deng, Y., May 2006. A declarative approach for specifying user-centric communication. In: Proceeding of the International Symposium on Collaborative Technologies and Systems – CTS 2006. IEEE, New York, pp. 89–98.

Costagliola, G., Deufemia, V., Polese, G., 2004. A framework for modeling and implementing visual notations with applications to software engineering. ACM TOSEM 13 (4), 431–487.

Day, J.D., Zimmermann, H., 1995. The osi reference model. In: Conformance Testing Methodologies and Architectures for OSI Protocols. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 38–44.

Deng, Y., Sadjadi, S.M., Clarke, P.J., Zhang, C., Hristidis, V., Rangaswami, R., Prabakar, N., 2006. A communication virtual machine. In: COMPSAC'06: Proceedings of the 30th Annual International Computer Software and Applications Conference – COMPSAC'06. IEEE Computer Society, Washington, DC, USA, pp. 521–531.

Eclipse, March 2007. Eclipse communication framework. <http://www.eclipse.org/ecf/>.

Emmerich, W., 2000. Software engineering and middleware: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. pp. 117–129.

Emmerich, W., Kaveh, N., 2002. Component technologies: Java beans, Com, Corba, Rmi, Ejb and the Corba component model. In: Proceedings of the 24th International Conference on Software Engineering. ACM Press, New York, pp. 691–692.

Ferguson, P., Humphrey, W.S., Khajenoori, S., Macke, S., Matvya, A., 1997. Results of applying the personal software process. Computer 30 (5), 24–31.

Geppert, B., Rößler, F., June 1996. Automatic configuration of communication subsystems – a survey. Technical Report SFB 501, Report 17/96, University of Kaiserslautern, Germany. <http://www.softmetaware.com/whitepapers.html>.

Google, March 2007. Google Talk. <http://www.google.com/talk/>.

Heckel, R., Voigt, H., 2004. Model-based development of executable business processes for web services. LNCS 3098, 559–584.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K., 2000. System architecture directions for networked sensors. SIGPLAN Not. 35 (11), 93–104.

Hristidis, V., Clarke, P.J., Prabakar, N., Deng, Y., White, J.A., Burke, R.P., 2006. A flexible approach for electronic medical records exchange. In: HIKM'06: Proceedings of the International Workshop on Healthcare Information and Knowledge Management. ACM Press, New York, USA, pp. 33–40.

Jabber, September 2007. <http://www.jabber.org/>.

JAIN-SIP, March 2006. <https://jain-sip.dev.java.net/>.

Java Media Framework API, June 2005. Internet2 working groups, and special interest groups. <http://java.sun.com/products/java-media/jmf/> (May 2005).

Krebs, D., April 2005. The mobile software stack for voice, data, and converged handheld devices. Mobile and Wireless Practice Venture Development Corporation.

Levin, R., Cohen, E., Corwin, W., Pollack, F., Wulf, W., 1975. Policy/mechanism separation in hydra. In: SOSP'75: Proceedings of the Fifth ACM Symposium on Operating Systems Principles. ACM Press, New York, USA, pp. 132–140.

McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C., 2004. Composing adaptive software. Computer 37 (7), 56–64.

Object Management Group, 1994. Common Object Services Specification. John Wiley & Sons, New York. For the latest specification: <http://www.omg.org/>.

Raj, G.S., September 1998. A detailed comparison of CORBA, DCOM, and Java/RMI (with detailed code examples). Object Management Group (OMG) whitepaper.

Rangaswami, R., Sadjadi, S.M., Prabakar, N., Deng, Y., April 2007. Automatic generation of user-centric multimedia communication services. In: Proceedings of the IEEE International Performance Computing and Communications Conference – IPCCC.

Satoshi, H., 1997. HORB: Distributed execution of java programs. In: WWCA., pp. 29–42. <citeseer.ist.psu.edu/satoshi97horb.html>.

Schmidt, D.C., 1997. Applying patterns and frameworks to develop object-oriented communication software. In: Salus, P. (Ed.), Handbook of Programming Languages, vol. 1.

Schmidt, D.C., 2002. Middleware for real-time and embedded systems. Commun. ACM 45 (6), 43–48 <http://portal.acm.org/citation.cfm?id=508472&dl=portal&dl=ACM#> .

Schmidt, D.C., 2006. Model-driven engineering. Computer 39 (2), 25–31.

Schmidt, D.C., Huston, S.D., 2002. C++ Network Programming: Mastering Complexity using ACE and Patterns. Addison-Wesley/Longman, New York.

Schmidt, D.C., Kuhns, F., 2000. An overview of the real-time CORBA specification. Computer 33 (6), 56–63. <citeseer.nj.nec.com/schmidt00overview.html>.

Skype Limited, February 2007. Skype developer zone. <https://developer.skype.com/>.

Stiller, B., Class, C., Waldvogel, M., Caronni, G., Bauer, D., 1999. A flexible middleware for multimedia communication: design, implementation, and experience. IEEE J. Select. Areas Commun. 17 (9), 1614–1631.

TegesTM Corporation, December 2005. i-Rounds. <http://www.teges.com/index.asp>.

The Parlay Group, June 2007. Parlay/osa specifications. <http://www.parlay.org/en/specifications/>.

van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D., 1998. Building adaptive systems using ensemble. Softw. Pract. Exper. 28 (9), 963–979.

Vinoski, S., 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Commun. Mag. 14 (2). <citeseer.nj.nec.com/vinoski97corba.html>.

Vinoski, S., 1998. New features for CORBA 3.0. Commun. ACM 41 (10), 44–52.

Wang, Y., Clarke, P.J., Wu, Y., Allen, A., Deng, Y., 2007. Realizing communication services using model-driven development. In: Proceedings of 11th IASTED International Conference on Software Engineering and Applications (SEA 2007). ACTA Press, November 19–21, pp. 473–479.

Zave, P., Goguen, H.H., Smith, T.M., 2004. Component coordination: a telecommunication case study. Comput. Networks 45 (5), 645–664.

Zhang, C., Sadjadi, S.M., Sun, W., Rangaswami, R., Deng, Y., November 2006. A user-centric network communication broker for multimedia collaborative computing. In: Proceedings of the Second International Conference on Collaborative Computing – CollaborateCom 2006.

Yi Deng received his Ph.D. in Computer Science from the University of Pittsburgh in 1992. He has been the Dean and Professor of School of Computing and Information Sciences at the Florida International University (FIU) – the State University of Florida in Miami and one of the largest urban research universities in the US. Under his leadership, the School has grown into one of the largest computer science and information technology education programs and one of the best externally funded research programs in State of Florida University System, a national leader in diversity, and an active partner to industry. He is an accomplished leader in computing and information technology research, innovation and application. He has authored or co-authored over ninety research papers in peer-reviewed journals and proceedings, and awarded eighteen research grants as the principal or co-principal investigator totaling over $15 million, most of which from premier US federal funding agencies. He has initiated and led many large scale multidisciplinary R&D and education projects and initiatives, founded and directed three research centers, including the Center for Advanced Distributed System Engineering, the NSF Center of Emerging Technologies for Advanced Information Processing and High Confidence Systems, and the IBM Center for Autonomic and Grid Computing at FIU. He has been an active contributor to the professional and research community in various leadership capacities. He co-founded and co-chairs the Board of Governors for the Latin American Grid (LA Grid) Consortium, with members include IBM, Barcelona Supercomputing Center and twelve universities in US, Puerto Rico, Mexico, Spain and Argentina, dedicated for collaborative research, innovation and workforce development in computing.

S. Masoud Sadjadi received the BS degree in Hardware Engineering from University of Tehran in 1995, the MS degree in Software Engineering from Azad University of Tehran in 1999, and the PhD degree in Computer Science from Michigan State University in 2004. He is currently an assistant professor in the School of Computing and Information Sciences at Florida International University. He has extensive experience in software development and leading large scale software projects. Currently, he is leading several international research projects in the Latin American Grid and is co-chair of the program committee for IEEE ICNSC 2008. His current research interests include Software Engineering, Distributed Systems, and High-Performance Computing with the focus on Autonomic, Pervasive, and Grid Computing. He is PI or Co-PI of 8 grants from NSF and IBM for total of over $3.5 million. He is a member of the IEEE.

Peter J. Clarke received his BS degree in Computer Science and Mathematics from the University of the West Indies in 1987, MS degree from SUNY Binghamton University in 1996 and PhD in Computer Science from Clemson University in 2003. His research interests are in the areas of software testing, software metrics, software maintenance, and model-driven software development. He is currently an Assistant Professor in the School of Computing and Information Sciences at FIU, where he started the Software Research Testing Group (STRG) in 2005. Dr. Clarke is a member of the ACM (SIGSOFT), IEEE Computer Society and a founding member of The Association of Software Testing.

Vagelis Hristidis received his BS in Electrical and Computer Engineering at the National Technical University of Athens in 1999. He received his M.Sc. and Ph.D. degrees in Computer Science in 2000 and 2004 respectively, at the Computer Science and Engineering Department of the University of California, San Diego (UCSD). Since 2004 he has been an Assistant Professor at the School of Computing and Information Sciences at Florida International University. His areas of expertise are Databases and Information Retrieval.

Raju Rangaswami received a B.Tech. degree in Computer Science from the Indian Institute of Technology, Kharagpur, India. He obtained M.S. and Ph.D. degrees in Computer Science from the University of California at Santa Barbara where he was the recipient of the Dean's Fellowship and the Dissertation Fellowship. Raju is currently an Assistant Professor of Computer Science at the Florida International University in Miami. His research interests include operating systems, storage systems, virtualization, security, and real-time systems. He is a recipient of the NSF CAREER award and the Department of Energy Early CAREER Principal Investigator (ECPI) award.

Yingbo Wang is currently enrolled in the PhD Program in School of Computing and Information Sciences at Florida International University (FIU). She received her Masters and Bachelors degree from Institute of Software, Chinese Academy of Sciences and University of Science and Technology of China (USTC) in 2003 and 2000. She is currently a graduate assistant under Dr. Yi Deng and Dr. Peter J. Clarke in the area of software modeling and model-driven development.