

A Domain-Specific Modeling Approach to Realizing User-Centric Communication

Yali Wu^{1*}, Andrew A. Allen¹, Frank Hernandez¹, Robert France², Peter J. Clarke¹

¹ School of Computing and Information Sciences, Florida International University, Miami, FL 33199, USA

² Department of Computer Science, Colorado State University, Fort Collins, CO80532, USA

SUMMARY

Advances in communication devices and technologies are dramatically expanding our communication capabilities and enabling a wide range of multimedia communication applications. The current approach to develop communication-intensive applications results in products that are fragmented, inflexible and incapable of responding to changing end-users' communication needs. These limitations have resulted in a need for a new development approach of building communication applications that are driven by end-users and that support the dynamic nature of communication-based collaboration. To address this need, the *Communication Virtual Machine (CVM) technology* has been developed to support rapid specification and automatic realization of user-centric communication applications based on a domain-specific modeling approach. The CVM technology consists of a domain-specific modeling language (DSML), the *Communication Modeling Language (CML)*, that is used to create communication models, and a semantic rich platform, the CVM, that realized the created communication models.

In this paper, we report on our experiences of applying a systematic approach to engineering CML and the synthesis of CML models in CVM. Based on a feature model describing the taxonomy of the user-centric communication domain in a network independent manner, we develop the meta-model of CML and its different concrete syntaxes. We also present a behavioral specification (dynamic semantics) of CML that enables the dynamic synthesis of user-centric communication models into an executable form called *communication control script*. We validated the CML semantics using *Kermeta*, a meta-programming environment for engineering DSMLs, and evaluated the practicality of our approach using a CVM prototype and a set of experiments.

Copyright © 2010 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Domain-Specific Modeling Language, Meta-Models, Model-Driven Development, User-Centric Communication

1. INTRODUCTION

Advances in communication devices and technologies (e.g. iPhone [1], iPad[2] and Android [3]) are dramatically expanding our communication capabilities and enabling a wide range of multimedia communication applications. Examples range from general-purpose applications that support voice calls, video conferencing, and instant messaging using a one-size-fits-all approach, to specialized applications in disaster management, distant learning and telemedicine. It is likely that the pace of innovation of new communication applications will accelerate even further. However, many of these applications have been conceived, designed and custom-built using a silo approach with little connection to each other, resulting in fragmented and incompatible technologies [4]. The

*E-mail: ywu001@cis.fiu.edu

*Correspondence to: School of Computing and Information Sciences, Florida International University, Miami, FL 33199, USA

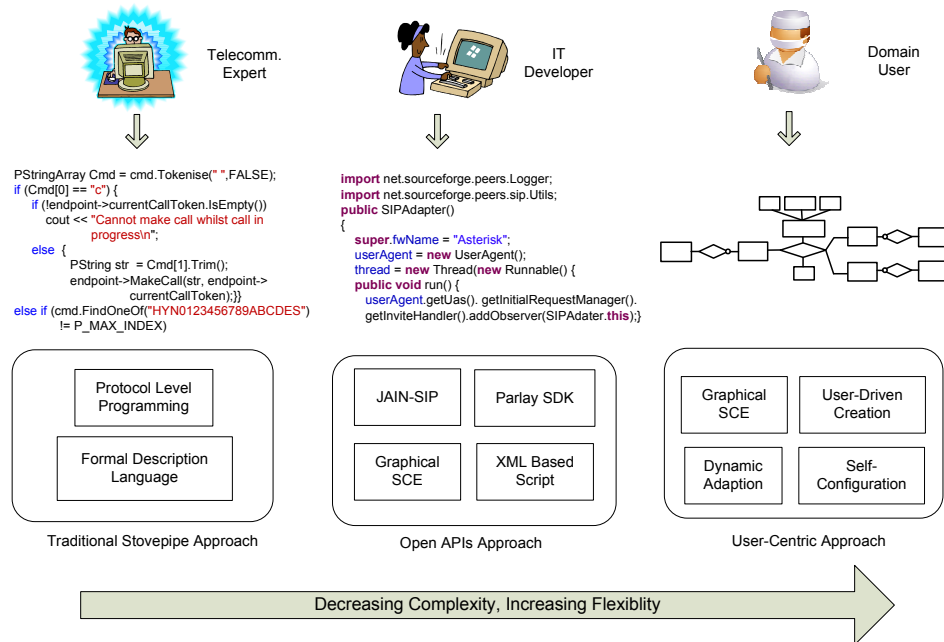


Figure 1. Paradigm shift in developing communication services/applications.

growing heterogeneities of network infrastructure and communication platforms make it both time-consuming and error-prone to develop new communications applications entirely from scratch. Much of the cost and effort stems from the fact that the heterogeneities and complexities of network-level communication control and media delivery are tightly bound with the diversity of application-dependent collaboration logic. This tight coupling also results in applications that are incapable of responding to changing user needs and the dynamics of underlying networks or devices.

To effectively decouple the development of application logic from the network infrastructure, different initiatives have developed open APIs or application frameworks that allow for rapid creation and deployment of converged communication services. These approaches (e.g., OSA/Parlay[5], JAIN-SLEE[6], and JAIN-SIP[7]) reduced the complexity of service creation by exposing a single point of interfacing with multiple protocols and resources. However, the creation of communication applications are still limited to IT developers who have to manually invoke these interfaces to program client-side applications (with potentially sophisticated collaborative logic). There is a strong demand for an easy and flexible way of building communication applications that are driven by end-users and that support the dynamic nature of communication-based collaboration. The user-centric service creation approach has emerged to address this need. Figure 1 summarizes this paradigm shift in creating and realizing communication-intensive services or applications. This evolution is also accompanied by increasing flexibility and decreasing complexity of the development process.

One initiative to realize the user-centric approach of creating communication applications is the *Communication Virtual Machine (CVM) technology* developed by Deng et al. [8]. CVM supports the rapid conception, specification and realization of communication applications through the use of a domain-specific modeling approach. The CVM technology consists of a domain-specific modeling language (DSML), the *Communication Modeling Language (CML)*, that is used to create communication models, and a semantic rich platform, the CVM, that realized the created communication models. This approach shields end-users from the complexities and heterogeneities of underlying communication technologies. We use the term *user-centric communication* to refer to communication applications or systems that are driven by end-users and mask device and application complexity while preserving the diversity and power of advanced communication tools [9]. While the term “user-centric” has a broad scope covering issues like flexibility and adaptability to user needs and context awareness, in this paper we limit the definition to the scope of

the aforementioned issues based on end-user driven specification of communication applications. User-centric service creation reduces the amount of effort required to specify communication applications. To assist in measuring aspects of user-centricity related to DSMLs, Wu et al. [10] are currently exploring metrics, such as modeling effort, cognitive effort, and scaffolding effort, among others.

In this paper we report on our experiences of applying a systematic approach to the engineering of CML. Starting from a domain analysis of end-user driven communication applications from the healthcare domain, we produce a feature model that identifies basic primitives for specifying end-user's communication needs. The feature model is then used to develop a meta-model that describes core CML elements. We also present a comprehensive behavioral specification of CML using label transition systems to describe the dynamic semantic of CML models. Finally, we present the implementation of the CVM prototype adhering to the four-layered architectural design[8]. The performance of the dynamic synthesis is evaluated by a series of experiments. The major contributions of this paper are as follows:

1. Application of a systematic approach to engineering a DSML for user-centric communication, including the development of:
 - a feature model describing the characteristics of user-centric communication
 - a meta-model for the domain-specific model language (CML)
 - a comprehensive behavioral specification of CML that enables dynamic synthesis of CML models
2. Prototypical implementation of the CVM platform
3. An evaluation of dynamic synthesis of CML models using the CVM prototype.

In contrast to previous work on CML that described certain aspects of the language design and presented these aspects in a very general way [11, 8, 12], we present a systematic and comprehensive approach to engineering CML. Clarke et al. [11] presented the first version of CML without defining the complete meta-model (abstract syntax and static semantics). Also, the initial version of CML has since evolved dramatically. Deng et al. [8] described the CVM technology more from a conceptual point of view without adequate details to sufficiently define CML as a DSML. Wang et al. [12] made the first attempt in defining the behavioral specification for CML. In this paper we provide a more comprehensive definition of the CML metamodel, and extend the semantics to include algorithms for coordinating the dynamic synthesis and a complete behavioral specification. We also provide implementation details of the synthesis process in CVM based on the behavioral specification.

In the next section we present a healthcare scenario to further motivate the problem. Section 3 describes the work related to user-centric communication and model driven initiatives applied to communication services. Section 4 describes our approach to developing CML. Section 5 presents the behavioral specification (dynamic semantics) used to dynamically synthesize CML models and validates the specification by model simulation. Section 6 describes the CVM prototype and Section 7 describes our experimental approach to evaluate dynamic synthesis using the CVM prototype. Section 8 presents our concluding remarks and future directions of this research.

2. MOTIVATION

The development of the initial version of the CVM was motivated in part by electronic communication problems that health professionals at the Miami Children's Hospital (MCH) experienced during their daily routines. Dr. Burke [13], Director of Cardiovascular Surgery at MCH, observed that setting up a conference call with other doctors using a variety of communication devices, and securely transferring patient data during the conference call is challenging. One scenario that illustrates the challenges facing healthcare professionals is given below:

Motivating Scenario: Baby Jane, who appears to have a heart condition, has been referred to MCH by Dr. Sanchez for observation and additional tests. Dr. Burke, chief cardiovascular surgeon

at MCH, has determined to perform surgery on her the following day. After Dr. Burke performs the surgery, he returns to his office and contacts Dr. Monteiro, the attending physician, to let him know how the surgery went. During the conversation, Dr. Burke brings Dr. Sanchez, the referencing physician, into the call. Dr. Burke then decides to share several aspects of Jane's medical record with him, including the post-surgery echocardiogram (echo), images of the patient's heart captured during the surgery, and the vital signs. Dr. Burke then invites Ms. Smith, a nurse practitioner, to join the conference to discuss and 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 scenario requires voice/video conferencing, compiling the data to be shared, exchanging different types of data (e.g., text, image, and video) in the patient's discharge package, access to medical information systems and logging the consultation for later reference. Clearly, realizing the above scenario is possible with today's technology. Dr. Burke would have to cobble together different communication technologies as follows: (1) he would place a phone call to reach Dr. Monteiro; (2) assuming Dr. Burke's phone has conferencing capability, he has to switch to a conference call to include Dr. Sanchez in the call, otherwise, they have to use a conferencing application such as Skype [14]; (3) Dr. Burke would then use a separate telemedicine application to compile the data from baby Jane's patient record and share it with Dr. Monteiro and Dr. Sanchez in real-time. In case they do not have access to such an application, Dr. Burke may need to extract parts of the record into a file and send them separately via email or a file sharing application.

Customized communication applications could also be developed using traditional approaches to support such scenarios. However, as mentioned before, the communication needs of such applications are usually hard-coded based on the implementation of the underlying network protocols and architecture. Therefore, the resulting applications would be inflexible and costly. Although open service APIs and application frameworks could ease the service creation process, a full development cycle is still required. Also, the resulting applications would not be able to accommodate unanticipated communication needs without incurring another development cycle.

The root cause to the above problems is the lack of a systematic, easy and flexible way of developing communication applications that are reconfigurable and adaptive at the user level. Providing the right level of abstraction to the end-user e.g., medical personnel, and making use of the available model-driven software engineering technologies e.g., DSMLs, provide us with a solution to this problem. We argue that using the CML language and CVM platform makes it easier to express user-level communication needs. CML is intended to target domain experts, persons within a communication-intensive application domain (e.g., healthcare) that have some IT knowledge, but are not software engineers or programmers. Through working with a DSML that is closer to the application domain, domain experts could achieve improved development productivity.

3. RELATED WORK

In this section we review three categories of related work including converged and user-centric communication, model driven approaches for communication and the state of the CVM technology prior to this paper. The only work, to the best of our knowledge, directly related to our work is the previous work on CVM.

3.1. Converged and User-Centric Communication

The trend for service convergence and user-centricity has been around for some time. Traditional telecommunication services like OSA/Parlay[5], JAIN-SLEE[6], and JAIN-SIP[7] provide open APIs or application frameworks that abstract the underlying network or the programming technology and enabled the convergence over wired, wireless, multimedia broadcasting and IP network. They simplified service creation by reducing the amount of telecommunication knowledge required for building communication solutions. However, developers are still required to use a general programming language such as Java in their respective IDEs to invoke such interfaces and

create a functional composition. The programming-level nature of these APIs or frameworks make them complex to use, and less usable than the user-level sessions of CVM, see Figure 1.

In the context of Service-Oriented Architecture(SOA), CVM serves as a client-side architecture, as opposed to server-side frameworks such as OSA/Parlay[5] and JAIN-SIP[7]. The server-side architecture has different concerns than the client-side architecture, which is the focus of CVM. As end-hosts are capable of handling sophisticated collaborative logic in IP networks, client-side architecture is getting as important and complicated as server-side architecture in building next-generation multimedia communication applications. Client-side architecture is becoming increasingly pertinent as more and more communication applications are deployed and executed on mobile devices. A big issue for developing communication software on mobile devices is the variety of platforms. For example, to develop communication applications on smart phones, two major platforms are iOS and Android. The CVM architecture, being inherently platform-independent, could be hosted on various mobile platforms.

Industrial user-centric communication initiatives have been described in different contexts such as: providing various customized communication services like teleconferencing and data services to businesses in specific domains [15, 9], modeling of communication environments [16], content applications that interact with communication sessions [17], and aligning user-centric communication with the unified communication paradigm. Unified communication has a similar objective to user-centric communication i.e., simplifying end-user experiences [18]. It aims to provide a way to integrate audio, video and data technologies on IP networks and allow switching among them while masking the complexities of heterogeneous back-end systems. These technologies, however, usually cannot accommodate unanticipated communication needs without first incurring a lengthy development cycle, resulting in a high cost to modify the functionality and interfaces required. Also in these approaches, user-centricity focuses on enhancing end-user experiences by providing rich yet easy-to-use communication applications. In our approach end-users drive the specification of the communication applications using an intuitive notation that provides them full control over the application functionality.

Recently, along with the emergence of Next Generation Networks (NGN), user-centricity has also motivated various convergent environments that provide user-centric service creation and delivery facilities. These include OPUCE (Open Platform for User-centric service Creation and Execution) [19], SPICE (Service Platform for Innovative Communication Environment) [20], among others[21, 22]. These environments support the combination of Internet IT services with communication services such as presence, voice calls and audio/video conferencing. They broadened the range of service creators and show directions towards user-centricity. However, the users are still required to choose or select the “base” services that they want to reuse, and build service mashups by composing two or more “base”services. Also, since mashup execution is driven by event triggers, users have to specify the service composition logic through a directed graph.

We argue that the CVM approach hides the details of service logic by presenting an abstract and declarative interface that does not require imperative encoding. In CVM, end-users are not aware of the specific set of services and resources available and only focused on their high-level application needs. Service composition is made very natural by updating the CML model to incorporate additional service specifications (such as transferring new media types and domain-specific forms). The logic for orchestrating different services could be extended via workflow-like constructs and is currently being investigated in [23]. In addition, there is a trade-off between domain-specificity and service generality in CVM, so that it can adopt a lightweight execution engine instead of a heavyweight open service creation and delivery platform.

3.2. Model Driven Initiatives For Communication

There have been various model driven initiatives for the communication domain. The MODATEL project [24], for instance, represents an initiative to develop methodologies and tools to apply model driven architectures (MDA) to the telecommunication domain. The MODATEL project is concerned with using well-established modeling languages traditionally used in the telecommunication domain within the context of the MDA. Initial endeavors include aligning OMG technologies (UML,

MOF, XMI, QVT) [25] with telecommunication languages like Service Design and Description Language (SDL) and Message Sequence Chart (MSC); reusing and integrating of MOF-based tools for telecommunication languages; and building a UML profile for communicating systems. In the MODATEL methodology, MDA tools apply a standard mapping to generate platform-specific model (PSMs) from platform-independent models (PIMs). Using this approach some code is automatically generated and some code is hand-written due to the fact that PIMs are not directly executable. Unlike these general purpose model driven initiatives, we used a DSML approach to capture the user-level communication needs. In addition, the models created by our DSML, CML, is directly executed by a semantic rich platform, the CVM.

The idea of using domain-specific languages (DSLs) to specify robust and adaptable communication services is not new. The Call Processing Language (CPL)[26], for instance, is a DSL for programming Internet telephony services that provide abstractions over signaling protocols and underlying systems. CPL could be used to quickly and safely specify call processing services such as call forwarding and anonymous call rejection. The goal of CPL is to ease the development of telephony services by providing users with domain-specific abstractions. Consel et. al [27] present a paradigm based on DSLs that enables networking and telecommunication experts to quickly develop robust variations of communication services within a particular domain(e.g., email processing, remote document processing, telephony services). The authors developed a programmable platform called Nova to execute the services of a particular domain written in respective DSLs. Burgy et al. [28] developed a DSL, Session Processing Language (SPL), and a virtual machine that is centered around the notion of a session in the Session Initiation Protocol.

Unlike these DSL approaches, which are mainly driven by abstractions over existing protocols or programming level networking solutions, and hence are used by engineers to develop variations of a family of telephony services, CML is designed to use the vocabulary of domain experts and focuses on user intuitive concepts without being limited by technological considerations. CML does not bear explicit relationships with existing service protocols or languages. Also, the semantics of CML allow it to be dynamically adaptable to changing user needs.

3.3. CVM Technology

Deng et al. [8] introduced the *Communication Virtual Machine (CVM) technology* as a user-centric communication initiative that supports the rapid conception, specification, and automatic realization of communication applications. Their approach is based on the model-driven engineering paradigm. Models of communication applications are specified using a DSML, *Communication Modeling language (CML)*, which are then realized using the CVM.

The initial version of CML was developed by Clarke et al. [11] and is designed as a declarative language for specifying the needs of communication applications from the users' point of view. The following design goals for CML are identified: (1) *simplicity*: be simple and intuitive; (2) *Network-independence*: be independent of network and device characteristics; and (3) *expressiveness*: model a large majority of communication scenarios. The primitive operations required to realize a communication application were identified based on an informal analysis of common scenarios in the healthcare domain. These operations include connection establishment, transfer of data or data stream, add/remove participants to/from a communication, group related data in a structure, and close connection. There are two equivalent versions of CML: XML-based (X-CML) and the graphical (G-CML). Two categories of communication models can be described using CML, *communication schemas* and *communication instances*, similar to the relationship between use cases and scenarios during requirements analysis. CML models are further classified as *control schemas* - configuration required to set up one or more connections in a communication (participants and the types of exchanged media); and *data schemas* - actual media (name or urls) to be exchanged during a communication. We provide additional details of CML syntax in Section 4.

CVM was designed using a layered architecture with each layer playing a separate role in realizing communication applications [8]. Figure 2 shows the layered architecture and the virtual communication between two CVMs, the virtual communication is shown using dashed lines. The four layers representing the different levels of abstraction in CVM include:

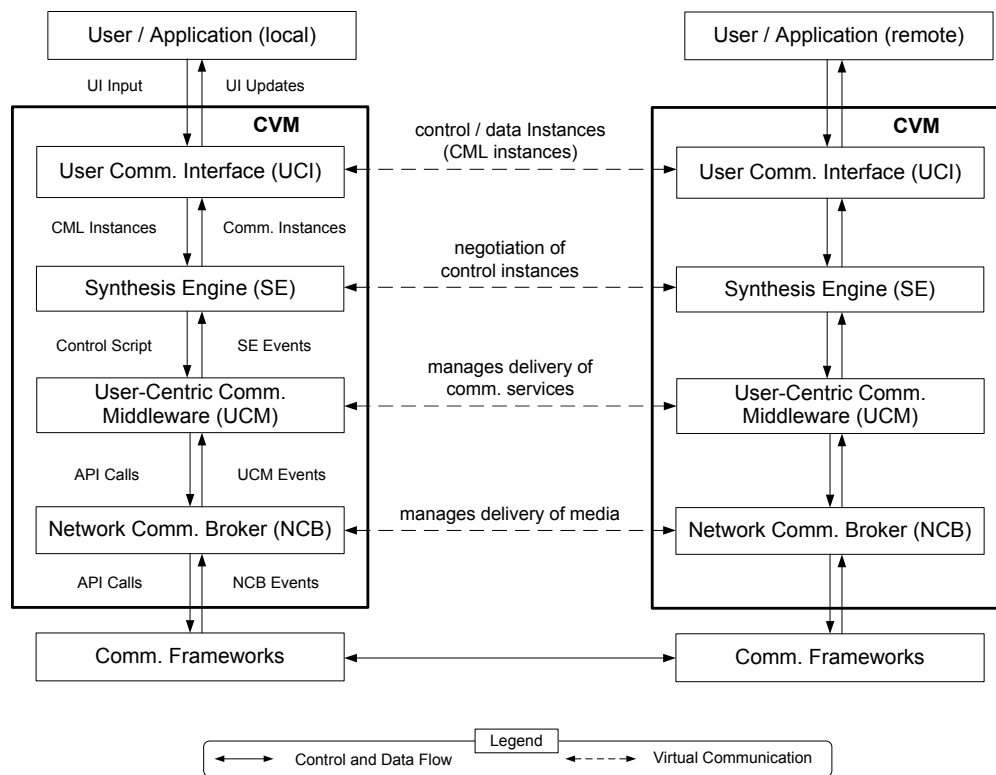


Figure 2. Layered architecture of the Communication Virtual Machine.

- *User Communication Interface (UCI)* - provides an environment where users can specify their communication requirements using CML models as either schemas or instances. Only communication instances can be realized by CVM.
- *Synthesis Engine (SE)* - negotiates the executing model with other participants in the communication, initiates media transfer and generates an executable script (*communication control script*) from a CML instance to be processed by the next layer.
- *User-centric Communication Middleware (UCM)* - executes the communication control script to manage and coordinate the delivery of communication services to users, including enabling/disabling streams and the transfer of atomic and structured data.
- *Network Communication Broker (NCB)* - provides a network-independent API to UCM and works with the underlying network protocols or communication frameworks to realize the actual communication.

Previous work by Deng et al. [8] lays the conceptual foundation for the work presented in this paper. Other papers published on the CVM technology have reported on different aspects of the research, such as the semantics to realize CML models developed by Wang et al. [12], the initial implementation of the UCM by Wu et al. [29], the design and implementation of the NCB by [30], and integrating autonomic capabilities into NCB by Allen et al. [31]. This paper presents a systematic and comprehensive approach to engineering CML, which includes an extension of the work by Wang et al. [12]. Wang et al. describe the operational semantics required to negotiate/re negotiate CML instances with participants in a communication and the transfer of media during communication. The semantics are specified using label transition systems (LTSs). These LTSs specify how CML models are interpreted in the synthesis engine (SE), a layer in CVM. The extension of the work by Wang et al. includes the definition of an algorithm that coordinates the dynamic synthesis of multiple CML instances, and a more formalized behavioral specification.

Although the four-layered architecture for CVM was defined by Deng et al. [8], the CVM prototype presented in this paper is the first implementation that follows this layered architecture.

4. CML - A DSML FOR USER-CENTRIC COMMUNICATION

Domain-specific modeling (DSM) is a development methodology that raises the level of abstraction beyond coding by specifying solutions directly using domain concepts [32]. DSM languages (DSMLs), therefore, offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [33]. The claimed benefits of DSMLs motivated us to design CML, a DSML that can be used to specify the requirements for user-centric communication use cases or scenarios. The use cases are modeled as communication schemas and the scenarios as communication instances, which are the communication applications providing services to the end-user.

In this section we present the structural design of CML based on the results of a domain analysis, including descriptions of the CML meta-model, and three concrete syntaxes. Unlike many DSMLs that are converted to a lower-level language and then executed, CML models are directly executed by CVM, and can be dynamically updated as a result of schema negotiation or a result of changes in the user's communication needs.

4.1. Domain Analysis

Over the past three years our group has been working with healthcare personnel at Miami Children's Hospital (MCH) to identify communication-intensive use cases. The doctors at MCH use a variety of available communication technologies when collaborating, e.g., email, instant messaging and video/teleconferencing, and problems arise when the technologies use different platforms and have different providers that offer different services. Integration of the technologies in most cases requires the doctors to be technology savvy and may require several applications to be cobbled together. As a result of analyzing the communication needs at MCH, we identified several communication-intensive scenarios, one of which is presented in Section 2.

From the identified set of scenarios, we perform domain analysis using the FODA (Feature-Oriented Domain Analysis) [34] methodology to yield a taxonomy model for user centric communication. The FODA methodology resulted in a feature model that captures commonalities (mandatory features) and variabilities (variable features) of the domain. It also represents a hierarchical decomposition of the features and their character, that is, whether they are mandatory, alternative, or optional. Feature combination rules are also specified that describe which combinations of features are valid or not.

We present the feature model as a feature diagram in Figure 3. The feature diagram shows the relationships between the different classes of features in our scenarios. Note that the identified features are by no means an exhaustive set. We chose essential features in order to build a minimal, intuitive and adequately expressive first version of CML. Advanced features related to requested properties for a connection or a particular data transfer are postponed for later versions of CML. These features include quality of service policies (e.g., if bandwidth is low then do not send video streams), security, access rights policies, and timing commands (e.g., transfer the sensor output every 5 seconds).

Figure 3 shows the domain of user-centric communication service consisting of three features: *basic communication*, *coordinated communication* and *service management*. Basic communication involves (1) connection, (2) media (primitive and composite) (3) devices for supporting the connection and media transfer, and (4) user behaviors for setting up the connection or initiating data transfer. *Connection* includes participants in the connection and connection properties. *Coordinated Communication* includes various control constructs for coordinating the behavior of basic communication activities. The service management includes features related to loading, saving and runtime adaption of communication services, and is usually offered by the service platform. The rest of the concepts in the feature diagram are self-explanatory. We are currently working on

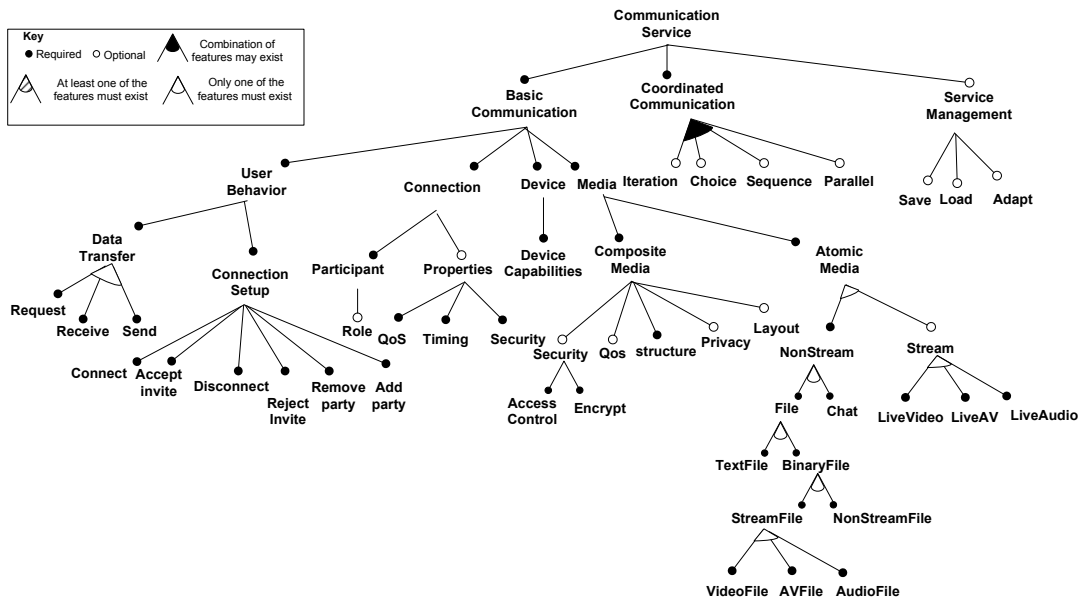


Figure 3. Feature oriented diagram for the communication domain.

extending CML to include coordinated communication, the extension will be referred to as *Workflow CML (WF-CML)*.

4.2. Meta-Model

The feature diagram, shown in Figure 3, facilitates the design of CML, a DSML, by identifying the potential constructs of the DSML. Deursen et al. in [35] described a systematic approach on how a feature diagram can be mapped to a UML class diagram. Based on the guidelines by Deursen et al. we developed the abstract syntax of CML using a class diagram, as shown in Figure 4.

The class diagram shows the various CML constructs that are used in modeling user-centric communication. The root construct `CommunicationSchema`, is either a `ControlSchema` or a `DataSchema`. A `ControlSchema` defines the static configuration of a communication which is composed of one or more `Connections`, one or more attached parties (`attachedParty`) and one or more data types (`DataType`). A `Connection` consists of one or more devices `Device`, and one or more `DataTypeDefs` (representing a reference to a `DataTypeDef` specifying the data/media that may be transmitted across the connection). Each `AttachedParty` has a `Person` attached to a `Device` through the `IsAttached` construct. `DataTypeDef` is defined as either a `MediumType` or a `FormType`. `MediumType` defines the particular type of the media that is supported in this communication, whereas `FormType` presents a composite data structure containing multiple medium types as well as user-defined attributes.

A `DataSchema` consists of an associated `DataContent` or `Request`. `DataContent` is used to define the actual data/media the user transfers/enables through an active communication, while `Request` is used to request for a media or a form structure. A medium is a piece of data or a medium stream, like a Word document or a live video feed, respectively. The type of the medium is defined in the `mediumDataType` attribute (a reference to `MediumType` class), which could be a built-in type or user-defined type supported by the system. A `mediumURL` that contains the location of the medium (e.g., a file location or a port for a data stream), as well as a `suggestedApplication`, which defines the application that can be used to view or process a medium (e.g., Powerpoint for ppt files). Composite data is represented using forms in CML, which are nested structures defined by the `FormType` class. `Form` has an `action` attribute which defines a default action that is performed on this medium during transfer. The actions “send” and “doNotSend” translate into automatic transfer of the medium in the form or wait for the user to request the medium, respectively. The

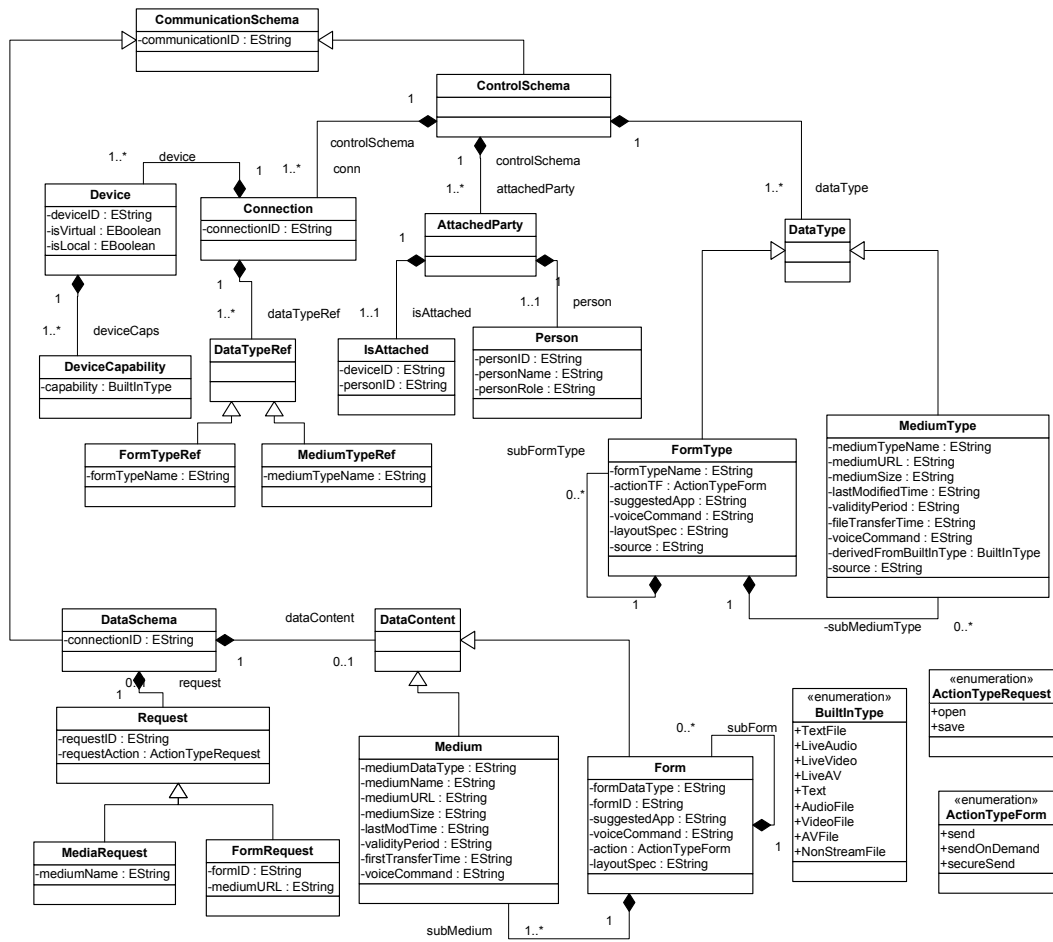


Figure 4. Abstract syntax for the CML control and data schemas represented as a class diagram.

“startApplication” action signals the system to open the suggested application of the data/medium after it is received by a participant.

We use OCL [36] to specify the static semantics rules for CML, as shown in Figure 14 in Appendix A. For instance, the first constraint is defined on the control schema, which indicates that the communication id should be unique with a non-null value, and the communication should have at least two parties, etc. The second constraint indicates that any media or form types associated with a connection must be supported by the device capabilities of all involved parties within the connection. The remaining constraints can be explained in similar manners.

4.3. Concrete Syntax

We designed three different representations of the concrete syntax of CML, namely the XML version, the graphical version, and a user interface (UI-CML) version. The XML version (X-CML), is an internal representation of the language that CVM understands and processes. The graphical version (G-CML) is developed for domain experts to rapidly develop communication services with user-intuitive concepts. The UI version presents an even simpler interface that allows casual users to realize communication services by loading and executing pre-defined CML models, as well as to visualize the progress of running services. We show screen shots for a three-way conference in the motivating scenario using the three concrete syntaxes as follows: (1) G-CML, see Figure 15 in Appendix B, (2) UI-CML, see Figure 16 in Appendix C, and (3) X-CML, see Figure 17 in Appendix

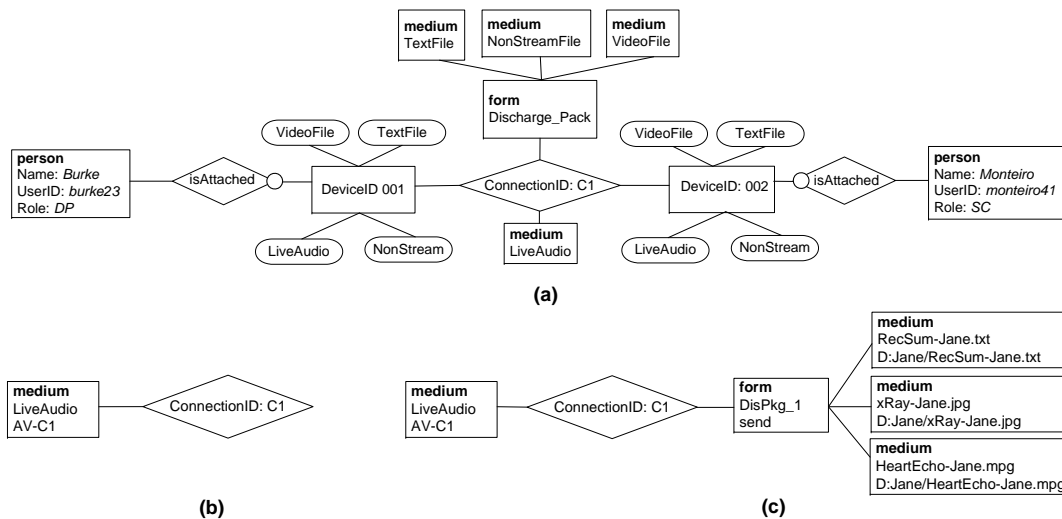


Figure 5. G-CML representation for: (a) the control instance for the 2-way call between Dr. Burke and Dr. Monteiro, (b) data instance to enable `LiveAudio`, (c) data instance to send the form `DisPkg_1`.

D. Note that Figure 16 represents a running instance of the scenario using a UI representation (UI-CML), while Figure 15 and 17 show a graphical (G-CML) and a XML (X-CML) representation of the CML instance respectively. Additional details on the concrete syntactical definitions are not shown in this paper but can be found on our project website*.

To automate the transformation between different representations of CML, we borrowed techniques used in model-driven engineering. Heckel and Voigt [37] state that pair grammars are used to describe the source and target languages together with a correspondence between source and target sentences. The authors show how pair grammars may be used to transform UML models to BPEL4WS [37]. We use a similar approach to convert between different representations of communication models. But our job is easier in that: (1) our paradigm tailors to a more restricted domain thereby providing us with the capability of easily automating the transformation process, and (2) since all these concrete syntaxes share the same abstract syntax of CML, the common meta-model could be used to in the mapping process.

4.4. Illustrative Example

In this section we show how the patient discharge communication scenario mentioned in Section 2 may be modeled using CML. We show the models used during the realizing of the scenario. Figure 5 shows the G-CML representations for the scenario. As discussed in the previous section, the frequency of exchanging data during communication is much higher than updating the communication configuration (e.g., list of participants, device capabilities), therefore we partition a CML instance into a control part (configuration) and a data part (exchanged media). Note that in practice the user only needs to create the G-CML or X-CML schema once, then for each instantiation of the schema, the user assigns values to the entities of the schema, e.g., assign names to participants and media.

Figure 5 shows several G-CML models (one control instance and two data instances) used by Dr. Burke in this scenario. Dr. Burke interacts with the user-friendly UI similar to the one shown in Figure 16, Appendix C. Figure 5(a) shows an instance of the control schema loaded by Dr. Burke. The control schema is created using a G-CML editor similar to the one shown in Figure 15, Appendix B. The devices for all participants in this connection need to support live audio/video streaming capabilities, as well as the transfer of patient discharge package.

*<http://www.cis.fiu.edu/cml/>

We represent composite data/media structures, such as a patient discharge package, using forms in CML. The discharge package form is defined to be composed of three pieces of media: a text file, a non-stream file and a video file. Detailed properties of the form (such as action or layout specification) are not shown in the figure. Figure 5(b) is the data instance for enabling live audio/video between Dr. Burke and Dr. Sanchez, while Figure 5(c) shows the data instance for sending the discharge package. Note that the live audio/video stream is still enabled in Figure 5(c).

5. DYNAMIC SYNTHESIS OF CML MODELS

In this section we describe the behavioral model (dynamic semantics) of CML that enables dynamic synthesis of user-centric communication applications expressed in CML. We first present an overview of the process to dynamically synthesize CML models, and then express the high-level semantics of CML models using a denotational style of semantics. The high-level semantics are instantiated by synthesis algorithms and detailed state machines for negotiation and media transfer. To partially validate the semantics, we simulated the CML models for several scenarios using a meta-modeling framework that supports the execution of domain-specific models.

5.1. Overview

In this section we provide an overview of the process to dynamically synthesize CML models. We use the term *synthesis* to refer to the automatic derivation of an executable form called *communication control script* from a CML model; and *dynamic* refers to the possibility of runtime updates to an executing CML model. These runtime updates may include adding new media types to an executing communication application, or adding a new participant, which requires renegotiation.

To assist the reader in understanding the process of dynamic synthesis, we show the overall realization process in Figure 6(a) and dynamic synthesis in Figure 6(b). Recall that realization refers to the dynamic synthesis of control scripts from CML models and subsequent execution of the behavior specified in CML models by executing these control scripts. As Figure 6(a) illustrates, realization of CML models starts when the user loads and instantiates a CML model in UCI. The realization process could be summarized as the following steps:

1. UCI validates the CML model and converts it into a *control schema instance* (CI) and a *data schema instance* (DI) pair, which is passed to SE
2. SE, upon receiving the CML instance pair (CI, DI), analyzes and synthesizes it into a control script to be executed by the UCM
3. UCM then executes the script and makes API calls to the NCB, which interfaces with the underlying communication frameworks
4. NCB interacts with the communication frameworks and generates UCM or SE events that are handled by their respective CVM layers.

In the above realization process that involves all CVM layers, SE is the layer that specifically handles the dynamic synthesis. Figure 6(b) shows a block diagram representing the executing processes in SE. The four major processes of SE include: *SE Controller*, *Schema Analyzer*, *Connection_i* (representing the i^{th} connection in the CML model), and *SE Dispatcher* [38]. We describe the functionality of each component as follows:

- *SE Controller*: coordinates the execution of incoming CML instances by interacting with the *Schema Analyzer* and *Connection* process. It also maintains and updates the SE environment accordingly during dynamic synthesis.
- *Schema Analyzer*: performs an analysis of the input instance pair and extracts the changes in the model, which are then fed into the *Connection* process.
- *Connection*: consists of (*re*)*negotiation* and *media transfer* processes. The negotiation process handles the negotiation logic for a given connection instance, while the media transfer process maintains the state of media transfer within the connection.

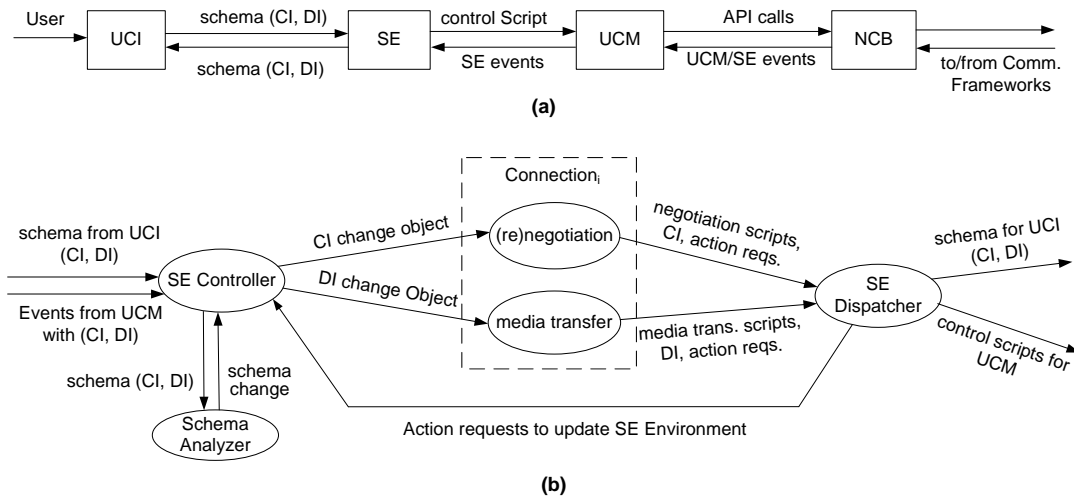


Figure 6. (a) Execution of a schema in the CVM. (b) Execution of a schema in the synthesis engine (SE). CI - Control instance; DI - Data exchange instance.

- *SE Dispatcher*: coordinates outgoing schemas to be displayed to UCI, control scripts to be passed to UCM, and action requests invoked back to the SE controller for updating the SE environment.

To visualize the process of dynamically synthesizing CML models, we show a series of CML instances and the control scripts generated as a result of dynamic synthesis in Figure 7. The table has three columns: the source of input, G-CML representations of the X-CML instances, and the generated control scripts. The synthesis of these CML instances at runtime realized the scenario described in Section 2. For easy readability, we only show a skeleton graphical version of the CML instance. We also break down the realization of communication services into three phases: initial negotiation of a two-way communication, enabling live audio/video, renegotiation into a three-way communication, and the media transfer of the patient discharge form. Note that SE also utilizes its environment during this process updating it accordingly, not shown in Figure 7.

We show the attributed grammar of the output control script language using EBNF-like notation in Figure 8. The grammar contains productions for all of the script commands used in the control script language. Rule 1 states that a control script consists of one or more script commands, and Rule 2 shows the various script commands. The strings in bold represent the actual command, and the attributes represent the parameters that the command can take. For example, Rule 3 states that *createConnectionCmd* is composed of the string **createConnection** and takes one parameter *connection ID*. The complete metamodel definitions for the control script language can be found on our project website*.

5.2. Semi-formal Behavioral Model for Dynamic Synthesis

We formalize the behavioral model of dynamically synthesizing CML models using a labeled transition system defined in terms of states (or configurations) stored in the SE environment, and a transition relation. This behavioral model is an extension of the work presented by Wang et al. [12]. The transition relation essentially specifies the rules for state changes based on the input CML model and the SE environment. It also defines the mapping from the input CML model to the output

*<http://www.cis.fiu.edu/cml/>

| Source | Skeleton of input CML | Output Control Script |
|---|-----------------------|--|
| Negotiation: (Establishing two-way communication) | | |
| UCI Layer | | <pre>createConnection("C1"); sendSchema("C1", "burke23", "monteiro41", "CI1, null")</pre> |
| UCM Layer (from monteiro41) | | <pre>sendSchema("C1", "burke23", "monteiro41", "CI2, null") addParticipant("C1", "monteiro41")</pre> |
| Media Transfer: Audio streaming | | |
| UCI Layer | | <pre>enableInitiator("C1", "LiveAudio") sendSchema("C1", "burke23", "monteiro41", "CI2, DI1")</pre> |
| UCM Layer (from monteiro41) | | <pre>enableReceiverMedia("C1", "LiveAudio");</pre> |
| ReNegotiation: (Establishing three-way communication) | | |
| UCI Layer | | <pre>sendSchema("C1", "burke23", "monteiro41, sanchez12", "CI3, DI3")</pre> |
| UCM Layer (from monteiro41) | | NA |
| UCM Layer (from sanchez12) | | <pre>sendSchema("C1", "burke23", "monteiro41, sanchez12", "CI5, DI3") addParticipant("C1", "sanchez12")</pre> |
| Media Transfer: Audio streaming and transfer of patient record | | |
| ⋮ | ⋮ | ⋮ |
| UCI Layer | | <pre>sendForm("C1", "Patient-Jane", "D:\Jane\RecSum-Jane.txt"); sendForm("C1", "Patient-Jane", "D:\Jane\heartEcho-Jane.mpg"); sendSchema("C1", "burke23", "monteiro41, sanchez12", "CI5, DI5");</pre> |

Figure 7. CML models and control scripts used in the motivating scenario in Section 2.

| | |
|---|---|
| <ol style="list-style-type: none"> 1. <i>controlScript</i> ::= <i>command</i> {<i>command</i>} 2. <i>command</i> ::= <i>createConnectionCmd</i> <i>closeConnectionCmd</i> <i>addParticipantCmd</i> <i>removeParticipantCmd</i> <i>sendSchemaCmd</i> <i>enableMediaInitiatorCmd</i> <i>enableMediaReceiverCmd</i> <i>disableMediaInitiatorCmd</i> <i>disableMediaReceiverCmd</i> <i>sendMediaCmd</i> <i>sendFormCmd</i> <i>declineConnectionCmd</i> <i>requestFormCmd</i> <i>requestMediaCmd</i> <i>sendNegTokenCmd</i> <i>requestNegTokenCmd</i> 3. <i>createConnectionCmd</i> ::= createConnection connectionID_A 4. <i>closeConnectionCmd</i> ::= closeConnection connectionID_A 5. <i>addParticipantCmd</i> ::= addParticipant connectionID_A personID_A {personID_A} 6. <i>removeParticipantCmd</i> ::= removeParticipant connectionID_A personID_A {personID_A} 7. <i>sendSchemaCmd</i> ::= sendSchema connectionID_A sender-personID_A receiver-personID_A {receiver-personID_A} schema_A 8. <i>enableMediaInitiatorCmd</i> ::= enableInitiatorMedia connectionID_A mediaName_A | <ol style="list-style-type: none"> 9. <i>enableMediaReceiverCmd</i> ::= enableReceiverMedia connectionID_A mediaName_A 10. <i>disableMediaInitiatorCmd</i> ::= disableInitiatorMedia connectionID_A mediaName_A 11. <i>disableMediaReceiverCmd</i> ::= disableReceiverMedia connectionID_A mediaName_A 12. <i>sendMediaCmd</i> ::= sendMedia connectionID_A mediaName_A mediumURL_A 13. <i>sendFormCmd</i> ::= sendForm connectionID_A formID_A mediumURL_A {mediumURL_A} action_A 14. <i>declineConnectionCmd</i> ::= declineConnection sender-personID_A receiver-personID_A {receiver-personID_A} 15. <i>requestFormCmd</i> ::= requestForm connectionID_A formID_A mediumURL_A {mediumURL_A} requestAction_A 16. <i>requestMediaCmd</i> ::= requestMedia connectionID_A mediaName_A requestAction_A 17. <i>sendNegTokenCmd</i> ::= sendNegToken personID_A 18. <i>requestNegTokenCmd</i> ::= requestNegToken connectionID_A |
|---|---|

Figure 8. EBNF-like grammar for the control script.

communication control script. The transition relation (R_{Syn}) is defined as follows:

$$((CI_{in}, DI_{in}), Env_i) \Longrightarrow ((CI_{out}, DI_{out}), (Script_{Neg}, Script_{MT}), Env_{i+1}) \quad (1)$$

where:

- (CI_{in}, DI_{in}) - input control and data instance pair to be processed (see left side of Figure 6(b)).
- Env_i - current environment including the complete user's view of the communication schema instance currently being executed (*userSchema*) and a connection environment (*Conn_Env_i*) for each of the connection processes currently being executed. The state of each connection, *Conn_Env_i*, is defined as a two-tuple (Neg_i, MT_i) , where:
 - Neg_i - current environment of a specific connection process with respect to (re)negotiation and it includes the control instance (CI_i) currently being negotiated or executed ($CI_i \in \{CI_{Neg}, CI_{exe}\}$), negotiation state (*Neg_State*), number of received responses (*num_responses*) and the presence of a negotiation token (*hasNegToken*). *Neg_State* is maintained in the *Negotiation* state machine discussed in Section 5.4.
 - MT_i - current environment of a specific connection process with respect to media transfer and it includes the currently executing *DI* (DI_{exe}), media transfer state (*MT_State*), and list of streams (*num_streams*). *MT_State* is maintained in the *Media Transfer* state machine discussed in Section 5.5.
- (CI_{out}, DI_{out}) - output instance pair generated during the transition process. This pair contains the *CI* and *DI* that may be sent to the UCI representing the currently executing connection process. The UCI updates the user's view of the communication using this instance pair.
- $(Script_{Neg}, Script_{MT})$ - control script pairs generated for the UCM to invoke the execution of the negotiation and media transfer macros. Note that these scripts may contain control instances and data instances for the remote participants in the communication. (see the rightmost column of Figure 7).

- Env_{i+1} - updated environment after the most recent transition. The structure is similar to Env_i stated above.

The behavior model associated with a connection is based on a sequence of instance pairs of the form (CI_i, DI_i) , where $i = 0, 1 \dots n$, CI is a control instance and DI a data instance. The initial instance pair (CI_0, DI_0) represents the initial state of the system with respect to some new connection to be established. Then the incoming CI_1 is compared to CI_0 resulting in changes to the system state and the negotiation process initiated. The future behavior of the system is determined based on subsequent incoming instance pairs and the current state of the system.

The input instance pair (CI_{in}, DI_{in}) may contain the user's specifications for multiple connections. Each instance pair is broken down into a set of instances per connection and handled separately. For each connection, we break down the transition relation R_{Syn} , defined in equation (1), into a relation for handling control instances during negotiation, and one for handling data instances during media transfer. The following relations are defined per connection j :

$$(CI_{in}, Env_i.Conn_j.Neg_k) \implies (CI_{out}, Script_{Neg}, Env_{i+1}.Conn_j.Neg_{k+1}) \quad (2)$$

$$(DI_{in}, Env_i.Conn_j.MT_k) \implies (DI_{out}, Script_{MT}, Env_{i+1}.Conn_j.MT_{k+1}) \quad (3)$$

R_{Neg} , shown in equation (2), represents the relation transforming the control instance during negotiation, and R_{MT} , equation (3), the relation for transforming the data instance during media transfer. A key part in our semantic definition is the changes identified between two successive models, therefore we define the functions $analyze_{CS}$ and $analyze_{DS}$ that identify the changes in the control schema instances and data schema instances, respectively. These changes are used to trigger events in the labeled transition systems for negotiation and media transfer. To specify the transition relations, R_{Neg} and R_{MT} in more detail, we define the following functions :

$$analyze_{CI} : ControlSchema \times EnvNeg \rightarrow ControlChangeType \quad (4)$$

$$analyze_{DI} : DataSchema \times EnvMT \rightarrow DataChangeType \quad (5)$$

$$update_{Neg} : ControlSchema \times EnvNeg \times ControlChangeType \rightarrow \\ ControlSchema \times EnvNeg \times NegScript \quad (6)$$

$$update_{MT} : DataSchema \times EnvMT \times DataChangeType \rightarrow \\ DataSchema \times EnvMT \times MediaScript \quad (7)$$

where:

- $ControlSchema$ is the type for control instances, CI (CI_{in} or CI_{out}), or the CI stored in the environment($Neg_i.CI_{Neg}$)
- $DataSchema$ is the type for data instances, DI (DI_{in} or DI_{out}) or DI stored in the environment($MT_i.DI_{exec}$)
- $ControlChangeType$ and $DataChangeType$ are the enumerated types for changes related to the input CI or DI , respectively
- $EnvNeg$ and $EnvMT$ are the types for the negotiation environment (Neg_i) and media transfer state (MT_i), respectively in $Conn_Env_j$
- $NegScript$ and $MediaScript$ are the types for the control scripts generated for negotiation ($Script_{Neg}$) and media transfer ($Script_{MT}$), respectively

The transition relation R_{Neg} can be defined by $analyze_{CI}$ in equation (4) and $update_{Neg}$ in equation (6) as follows:

$$R_{Neg} = update_{Neg}(CI_{in}, Env_i.Conn_j.Neg_k, analyze_{CI}(CI_{in}, Env_i.Conn_j.Neg_k)) \quad (8)$$

Similarly, we specify the translation relation R_{MT} as follows, using $analyze_{DI}$ in equation (5) and $update_{MT}$ in equation (7) :

$$R_{MT} = update_{MT}(DI_{in}, Env_i.Conn_j.MT_k, analyze_{DI}(DI_{in}, Env_i.Conn_j.MT_k)) \quad (9)$$

Algorithm 1 Algorithm of SE Controller for Synthesizing CML models.

```

1: synthesisCML (ref (CIin, DIin))
2: connSchemas ← decomposer.split((CIin, DIin))
3: /* connSchemas contains a list of (CIj, DIj), each
   being an input instance pair for jth connection */
4: if (CIin, DIin) is from UCI then
5:   new_conn ← getNewConn(Envi.userSchema, connSchemas)
6:   /* getNewConn returns the set of connections to be created */
7:   if not (new_conn == null) then
8:     for each conn ∈ new_conn do
9:       conn_proc ← ConnProc.create(conn)
10:      addConnProc(Envi, conn_proc)
11:      conn_proc.start()
12:     end for
13:   end if
14: end if
15: for each (CIj, DIj) ∈ ConnSchemas do
16:   conn_proc ← locateConnProc (j)
17:   exec_CML_CI(conn_proc, CIj)
18:   exec_CML_DI(conn_proc, DIj)
19: end for
20: old_conn ← getOldConn(Envi.userSchema, connSchemas)
21: /* GetOldConn returns the set of connections to be torn down */
22: if not (old_conn == null) then
23:   for each conn in old_conn do
24:     conn_proc ← locateConnProc(j)
25:     exec_CML_CI(conn_proc, null)
26:     exec_CML_DI(conn_proc, null)
27:     /* clean up the state machines before tearing down the process */
28:     conn_proc.terminate()
29:   end for
30: end if

31: exec_CML_CI(ref conn_proc, ref CI)
32: CCI ← analyzer.analyzeCI(CI, conn_proc.Conn.Envi, source)
33: /* Extract the difference in control instance and pass it
   to the Negotiation machine in the connection process */
34: conn_proc.Nego.stepNego(CCI)

35: exec_CML_DI(ref conn_proc, ref DI)
36: /* Assume a data schema carries one piece of media to be transferred */
37: CDI ← analyzer.analyzeDI(DI, conn_proc.Conn.Envi, source)
38: conn_proc.MT.stepMT(CDI)

```

The specifics of these functions will be explained further in subsequent sections in terms of synthesis algorithms and state machines. To relate to the execution flow shown in Figure 6(b), the Env_i is maintained in the SE Controller, while the negotiation and media transfer states are maintained by the negotiation and media transfer processes, respectively.

5.3. Synthesis Algorithms

In this subsection, we present synthesis algorithms that provide the basis for dynamically synthesizing CML instances. Based on the execution architecture presented in Figure 6, the algorithms for the SE Controller and Schema Analyzer processes are described. Detailed behavioral logic for controlling the process of negotiation and media transfer are modeled as concurrent state machines in the connection process. Definitions of these machines will be introduced in Section 5.4 and 5.5, respectively.

CML models are received as input by SE Controller in the form of a (CI, DI) instance pair, the `synthesisCML` method of the SE Controller object is invoked to process the incoming instance pair. As shown in Algorithm 1, the `synthesisCML` algorithm starts by invoking an internal `split` method, line 2, which splits the input (CI, DI) into a list of connection schemas each in the form of (CI_j, DI_j) ($0 \leq j \leq n$, n being the number of connections in the model). Recall

Algorithm 2 Algorithm to analyze CI.

```

1: analyze_CI (ref CIi+1, ref Conn.Envi, sourceCI)
   /*Input: CIi+1 - schema from the UCI or UCM
   Conn.Envi - current environment object for a specific connection
   sourceCI - source of CIi+1, UCI or UCM
   Output: cci, an object with CI changes and an event trigger */
2: cci ← compare(CIi+1, Conn.Envi.CIi)
3: if sourceCI == UCI then
4:   if cci.enum ∈ {initialCI} then
5:     cci.addEvent(InitiateNeg)
     /* SE Controller uses this event to start the state
     machines (SMs) for negotiation and media transfer
     and passes the initiateNeg to the negotiation SM */
6:   else if cci.enum ∈ {partyRemoved} then
7:     cci.addEvent(removeSelf)
8:   else if cci.enum ∈ {selfRemoved} then
9:     cci.addEvent(removeParty)
10:  else if not cci.enum ∈ {noChange} then
11:    cci.addEvent(InitiateReNeg)
12:  end if
13: else if sourceCI == UCM && self.isInitiator then
14:   if cci.enum ∈ {partyRemoved} then
15:     cci.addEvent(removeParty)
16:   else if cci.enum ∈ {noChange} then
17:     cci.addEvent(localSameCI)
18:   else
19:     cci.addEvent(localChangeCI)
20:   end if
21: else
22:   /*sourceCI == UCM && !self.isInitiator */
23:   if Conn.Envi.CIi == null then
24:     cci.addEvent(inviteNeg)
     /* CI Controller uses this event to start the state
     machines (SMs) for negotiation and media transfer
     and passes the inviteNeg to the negotiation SM */
25:   else if cci.enum ∈ {partyRemoved} then
26:     cci.addEvent(removeParty)
27:   else if cci.enum ∈ {noChange} then
28:     cci.addEvent(remoteSameCI)
29:   else
30:     cci.addEvent(remoteChangeCI)
31:   end if
32: end if
33: return cci

```

that a CML model could contain multiple connections. For each new connection we create a new connection process (containing the concurrent negotiation and media transfer state machines) and add it to the pool of connection processes, line 7 to 13.

Next we iterate through the set of instance pairs (CI_j, DI_j), and for each pair, the appropriate connection process is located by invoking an internal `locateConnProc` method (lines 16). The `exec_CML_CI` method, line 17, is invoked to analyze the control instance (CI) for this connection identifying the changes to the CI, and trigger an execution step in the corresponding negotiation state machine. A similar approach is used for the data instance by invoking `exec_CML_DI` on line 18.

Connections that are to be removed, lines 20 to 30, are selected by comparing the current schema instance in the environment and the input list of connection instances. For each of these old connections, its negotiation and media transfer state machines are terminated by passing them a null schema before the connection process is terminated.

The schema analysis process is invoked by the SE Controller after receiving a schema from the UCI or a schema event from the UCM, see Figure 6(b). The algorithm for analyzing a schema instance is composed of two algorithms: (1) `analyze_CI` for analyzing control schema instances, shown in Algorithm 2, and (2) `analyze_DI` for analyzing data schema instances, shown in Algorithm 3. The input parameters to both algorithms include a connection instance from the SE

Algorithm 3 Algorithm to analyze DI.

```

1: analyze_DI (ref DIi+1, ref Conn_Envi, sourceDI)
   /*Input: DIi+1 - schema from the UCI or UCM
     Conn_Envi - current environment object
     sourceDI - source of DIi+1 is either UCI or UCM
   Output: - cdi, an object with DI changes and an event trigger */
2: cdi ← compare(DIi+1, Conn_Envi.DIi)
3: if sourceDI == UCI then
4:   if cdi.enum ∈ {streamAdded} then
5:     cdi.addEvent(enableStream)
6:   else if cdi.enum ∈ {streamRemoved} then
7:     cdi.addEvent(disableStream)
8:   else if cdi.enum ∈ {nonStreamAdded} then
9:     cdi.addEvent(sendNonStream)
10:  else if cdi.enum ∈ {formAdded} then
11:    cdi.addEvent(sendForm)
12:  end if
13: else
14:   /*sourceDI == UCM*/
15:   if cdi.enum ∈ {streamAdded} then
16:     cdi.addEvent(enableStreamRec)
17:   else if cdi.enum ∈ {streamRemoved} then
18:     cdi.addEvent(disableStreamRec)
19:   else if cdi.enum ∈ {nonStreamAdded} then
20:     cdi.addEvent(recNonStream)
21:   else if cdi.enum ∈ {receiveForm} then
22:     cdi.addEvent(recForm)
23:   end if
24: end if
25: return cdi

```

Controller (CI_{i+1} or DI_{i+1}, for connection j), a reference to the current connection environment of the SE (Conn_Env_i), and the source of the schema to be analyzed (UCI or UCM). Each algorithm returns an object that contains the specific changes between the new connection instance and the current instance in Conn_Env_i, including the event that triggers the transitions in the state machines for (re)negotiation and media transfer. Algorithms `analyze_CI` and `analyze_DI` correspond to equations (4) and (5), respectively.

The `compare` function, line 2, in both algorithms computes the change between the new connection instance (CI_{i+1} or DI_{i+1}) and current one in Conn_Env_i, and stores them in the object `cci`, for control instance changes, or `cdi`, for data instance changes. The fields in this object include an enumeration that specifies the category of change, e.g., `initialCI` as shown on line 4 in Algorithm 2, among other information used during the execution of the state machines such as values for evaluating guards and information needed to generate control scripts. The SE Controller uses the contents of the `cci` or `cdi` object to trigger appropriate transition(s) in corresponding state machines for the execution of specific communication services.

The running time of `synthesisCML` shown in Algorithm 1 is dependent on the maximum number of connections in both the incoming control instance (CI_{in}) and the control instance stored in the environment (Env_i), and the product of the number of nodes in the two control instances analyzed in Algorithm 2. Each of the control instances, CI_{i+1} and CI_i, in algorithm `analyze_CI` has only one connection. We deduce that the worst case running time for the algorithm `synthesisCML` is $O(m^2 + (m * n^2))$ where m is the maximum number of connections in either CI_{in} or the control instance in Env_i, and n if the maximum number of nodes in either CI_{i+1} or CI_i. Since $m \ll n$ for most communication instances then the worst case running time will be closer to $O(n^2)$.

5.4. Negotiation

The behavioral logic for controlling the negotiation of communication services is captured in the `negotiation` state machine in our semantic specification, see Equations (6) and (8) in Section 5.2. The separation of negotiation logic from the schema analysis ensures the extensibility of the

Table I. State machine for schema (re)negotiation.

| <i>T</i> | <i>Source_State</i> | <i>Target_State</i> | <i>Event</i> | <i>Guard</i> | <i>Action</i> |
|----------|---------------------|---------------------|------------------|---------------------------------|---|
| 0 | Initial | NegReady | | | |
| 1 | NegReady | NegInitiated | initiateNeg | hasNegToken | addNegBlock(CI_{neg}) genConnection_Script |
| 2 | NegReady | NegInitiated | initiateReNeg | hasNegToken | addNegBlock(CI_{neg}) |
| 3 | NegInitiated | WaitingSameCI | | # remoteParty != 0 | genSendCS_Script |
| 4 | WaitingSameCI | WaitingSameCI | localSameCI | # responses < # remoteParty | |
| 5 | WaitingSameCI | NegComplete | localSameCI | # responses == # remoteParty | genSendCS_Script |
| 6 | NegComplete | NegReady | | | $CI_{exe} \leftarrow CI_{neg}$ UCI.notify(CI_{exe}) |
| 7 | WaitingSameCI | WaitingAnyCI | localChangeCI | # responses < # remoteParty | update(CI_{neg}) |
| 8 | WaitingSameCI | NegComplete | localChangeCI | # responses == # remoteParty | update(CI_{neg}) genSendCS_Script |
| 9 | WaitingAnyCI | WaitingAnyCI | localSameCI | # responses < # remoteParty | |
| 10 | WaitingAnyCI | WaitingAnyCI | localChangeCI | # responses < # remoteParty | update(CI_{neg}) |
| 11 | WaitingAnyCI | NegComplete | localSameCI | # responses == # remoteParty | genSendCS_Script |
| 12 | WaitingAnyCI | NegComplete | localChangeCI | # responses == # remoteParty | update(CI_{neg}) genSendCS_Script |
| 13 | WaitingSameCI | WaitingAnyCI | after t sec. | # remoteParty > 1 | update(CI_{neg}) |
| 14 | WaitingAnyCI | WaitingAnyCI | after t sec. | # remoteParty > 1 | update(CI_{neg}) |
| 15 | WaitingSameCI | NegTerminateInit | after t sec. | # remoteParty == 1 | |
| 16 | WaitingAnyCI | NegTerminateInit | after t sec. | # remoteParty == 1 | |
| 17 | NegReady | NegRequested | inviteNeg | | notifyUCI_InviteNeg |
| 18 | NegRequested | NegTermRemote | UCI.rejectInvite | | genRejectInvite_Script |
| 19 | NegTermInit | Final | | | UCI.notify(CI_{exe}) genCloseConnect_Script |
| 20 | NegTermRemote | Final | | | |
| 21 | NegRequested | WaitingConfirm | UCI.acceptInvite | | genConnection_Script genSendCS_Script |
| 22 | WaitingConfirm | NegComplete | remoteSameCI | | |
| 23 | WaitingConfirm | WaitingConfirm | remoteChangeCI | | update(CI_{neg}) genSendCS_Script |
| 24 | WaitingConfirm | NegTermRemote | after t sec. | | UCI.notify(CI_{exe}) genCloseConnect_Script |
| 25 | NegReady | SelfRemoved | removeSelf | hasNegToken | genRemoveSelf_Script |
| 26 | NegReady | NegReady | removeParty | # remoteParty > 1 | update(CI_{neg}) $CI_{exe} \leftarrow CI_{neg}$ genRemoveParty_Script |
| 27 | NegReady | Final | removeParty | # remoteParty == 1 | genCloseConnect_Script UCI.notify(CI_{exe}) |
| 28 | SelfRemoved | Final | | | |

negotiation logic in future iterations of the semantic specification. As shown in Figure 6(b), the input to the negotiation state machine is a control instance change object (output of analyze_CI method). The output of the state machine consists of a generated control script for negotiation, control instances to be sent to UCI, and relevant action requests for updating the SE environment. Note that not all components of the output need to have values for a particular transition rule.

Table V shows the state machine for negotiation and renegotiation. The table contains six columns, the columns from left to right are: the transition number, the source and target states, the event to trigger the transition, the guard to be satisfied before the transition can be triggered and the action to be taken after the transition has been triggered. Since some actions will change the environment of the SE, the SE Dispatcher will direct them back to the SE Controller, such as update(CI_{neg}) (See Figure 6(b)).

An example of a transition in the state machine is Transition 1, Row 2 of Table V, from the source state `NegReady` to the target state `NegInitiated`, which is triggered by the `initiateNeg` event, assuming that the environment has the negotiating token (`hasNegToken` is true). As a result of the transition a negotiation block is added to the new control instance, `addNegBlock(CIneg)`, and a script to create the connection is generated, `genConnection.Script`. In Table V we use the following notation for guards and actions:

- `hasNegToken` - negotiating token that must be obtained before starting a negotiation.
- `# remoteParty` - number of remote participants in the negotiation.
- `# responses` - number of responses from the remote participants
- `addNegBlock(CIneg)` - block in the schema that keeps data associated with the negotiation process, e.g., sender's id, negotiation initiator's id. This is an action request delegated to the SE Controller.
- `genXXX.Script` - generates the XXX control script. Recall a control script may contain one or more script commands.
- `update(CIneg)` - updates the schema being negotiated based on changes such as, removal of a participant or removing self from the schema. This is an action request delegated to the SE Controller.
- `UCI.notify(CIout)` - send the output CI to the UCI to inform the user of the result of the negotiation.

During negotiation the state machine waits for responses from remote participants. All remote participants replying with the same control instance leads to `NegComplete` state and a control script is generated to inform all remote participants that negotiation is complete. Any CI received from a remote participant that is different from the one that was sent will result in a transition to the `WaitingAnyCI` state and the `CIneg` is updated to include this change. If all the CIs received from the remote participants are the same as the one sent, then the system transitions to the `NegComplete` state followed by the `NegReady` state and awaits another negotiation cycle. A successful negotiation always leads to `NegComplete` state and replace the executing schema `CIexe` in the `userSchema` with `CIneg`. No response received in t seconds from any remote participants implies a timeout and either results in the removal of the specific participant from the schema or a negotiation failure.

5.5. Media Transfer

In a similar manner to `negotiation`, the `media transfer` state machine, shown in Table VI, is constructed to separate the media transfer logic from the specifics of the schema analysis process. Similar to `negotiation`, the input to the `media transfer` state machine is a data instance change object (output of `analyze_DI` method). The output of the state machine includes a generated control script for media transfer, data instances to be sent up to UCI, and relevant action requests for updating the SE environment. Similar to the output of the `negotiation` state machine, not all components need to have values for a particular transition rule. The semantic specification for `media transfer` is defined in Equations (7) and (9) in Section 5.2.

Although we do not show it in Table VI the executing DI is updated for Transitions 1 through 17, i.e., the entry $DS_{exec} \leftarrow DS_{in}$ should be in the *Action* column. We use a similar notation for the guards and actions as shown below:

- `IsStreamEnabled` - is a boolean that represents if a particular live stream m (such as audio, video, audio-video) specified in `DSexe` is enabled, see the predefined types for CML in [8]. We leave out the parameter m to simplify the notation in the table.
- `# streams` - represent the number of active live streams.
- `UCI.notify(DIout)` - send the data schema to the UCI to inform the user that a new media is enabled or received from a remote participant.

The change object `cdi` returned from the `analyze_DI` contains the information required by the state machine to trigger transitions and perform actions. For example, Transition

Table II. State machine for media transfer.

| Tr. | Source_State | Target_State | Event | Guard | Action |
|-----|----------------|---------------|----------------------------------|--|--|
| 0 | Initial | Ready | initiateNeg initiateInviteNeg | | |
| 1 | Ready | StreamEnabled | enableStream | | genStreamEnable_Script |
| 2 | Ready | StreamEnabled | enableStreamRec | | genStreamEnableRec_Script UCI.notify(DS_{i+1}) |
| 3 | StreamEnabled | StreamEnabled | enableStream | !IsStreamEnabled | genStreamEnable_Script |
| 4 | StreamEnabled | StreamEnabled | disableStream | IsStreamEnabled && # streams > 1 | genStreamDisable_Script |
| 5 | StreamEnabled | StreamEnabled | enableStreamRec | !IsStreamEnabled | genStreamEnableRec_Script UCI.notify(DS_{out}) |
| 6 | StreamEnabled | StreamEnabled | disableStreamRec | IsStreamEnabled && # streams > 1 | genStreamDisableRec_Script UCI.notify(DS_{out}) |
| 7 | StreamEnabled | StreamEnabled | sendNonStream | | genNonStreamSend_Script |
| 8 | StreamEnabled | StreamEnabled | sendForm | | genSendForm_Script |
| 9 | StreamEnabled | StreamEnabled | recNonStream | | UCI.notify(DS_{out}) |
| 10 | StreamEnabled | StreamEnabled | recForm | | UCI.notify(DS_{out}) |
| 11 | StreamEnabled | Ready | disableStream | # streams == 1 | genCloseStream_Script |
| 12 | StreamEnabled | Ready | disableStreamRec | # streams == 1 | genCloseStreamRec_Script UCI.notify(DS_{out}) |
| 13 | Ready | Ready | sendNonStream | | genNonStreamSend_Script |
| 14 | Ready | Ready | sendForm | | genSendForm_Script |
| 15 | Ready | Ready | recNonStream | | UCI.notify(DS_{out}) |
| 16 | Ready | Ready | recForm | | UCI.notify(DS_{out}) |
| 17 | Ready | Final | terminate | | |

1, Row 2 in Table VI, represents the transition from source state Ready to target state StreamEnabled. This transition does not have a guard and the resulting script generated, genStreamEnabled_Script, enables the stream on the sender's side of the communication channel.

5.6. Validation of Dynamic Synthesis using Kermeta

We simulated the semantics of CML models for dynamic synthesis as a form of validation after developing the semantics. To perform the simulation, a meta-modeling tooling facility was needed that could facilitate the definition of the syntax and semantics of CML. In addition, the tool should allow for the definition of executable models. As a result of investigating several meta-modeling-based simulation tools, we decide to use Kermeta [39]. Kermeta is a powerful meta-programming environment for engineering domain-specific languages like CML, with strong tooling support. It allows the description of meta-models whose models are directly executable, and also it is easy to use due to the extensive documentation available online and accessibility of the developers.

Simulation approach: In our simulation, we first specified the CML meta-model using ECORE (Figure 4). To specify the behavioral semantics of CML, we first modeled the synthesis engine (SE) and used the generation tool, Ecore2Kermeta, in the workbench to translate the SE model to a Kermeta compatible version. We then coded the semantic specification defined previously as operations within the SE model. Figure 9 shows the class *NegController* (part of the SE model) for our implementation of the semantics using Kermeta. The operation *stepNego*, line 5, shows part of the logic for the negotiation state machine.

In order to simulate the dynamic synthesis of CML models we created a harness which included stubs and drivers. The stubs included a pretty printer that generated control scripts as output and updated the system environment. A test driver was also created, which initiated a SE controller object, loaded a CML model instance and passed it to the SE controller (See Figure 6(b)). To simulate the negotiation of CML models between multiple parties, a series of CML instances were fed into the driver during runtime to simulate the reply from each remote participant. The output of the simulation program was the control script to be deployed in UCM, and the updated environment of the synthesis engine.

```

1 aspect class NegController {
2   reference negotiation : Negotiation;
3   reference schemaAnalysis : SchemaAnalyzer;
4   reference scriptGen : ScriptGenerator;
5   operation stepNego(changeCS: SchemaChangeObj): NegotiationEvent is do {
6     var event: NegotiationEvent;
7     if(changeCS.getEvent() == "initiateNegEvent" and
8       negotiation.currentState == negoState.NegReady )
9       negotiation.startNeg_Init()
10      scriptGen.genConnectionSendCS_Script(negotiation, changeCS)
11    end
12
13    if(changeCS.getEvent() == "acceptInvite" and
14      negotiation.currentState == negoState.NegRequested )
15      negotiation.startNeg_Rev()
16      scriptGen.genConnectionSendCS_Script(negotiation, changeCS)
17    end
18
19    if(changeCS.getEvent() == "removeSelfEvent" and
20      negotiation.currentState == negoState.NegReady )
21      negotiation.removeSelf()
22      scriptGen.genRemoveSelf_Script(negotiation)
23    end
24  }
25  /* Remaining code left out due to space limitation */
26 end}

```

Figure 9. An excerpt of Kermeta Code For NegController class

Limitations of Simulation: The main limitation of the simulation relates to the lack of simulating concurrent processes in Kermeta. Due to Kermeta's indirect support for concurrent operations, we had to use interleaving semantics to model concurrency for the (re)negotiation and media transfer processes (See Figure 6(b)), as well as inter-process concurrency. Interleaving semantics proved enough for the purpose of simulating CML models.

6. CVM PROTOTYPE

The CVM prototype consists of four major subsystems and a utility package as shown in Figure 10. These subsystems are the four components previously described in Section 3.3. A more in-depth conceptual description of these components are presented by Deng et al. [8]. Unlike previous CVM prototypes described in the literature, this prototype is the first one to follow the four layered architecture and contain a graphical modeling environment. In this section we focus our description on the user communication interface (UCI) and the synthesis engine (SE). A summary of the components used in the prototype is given below:

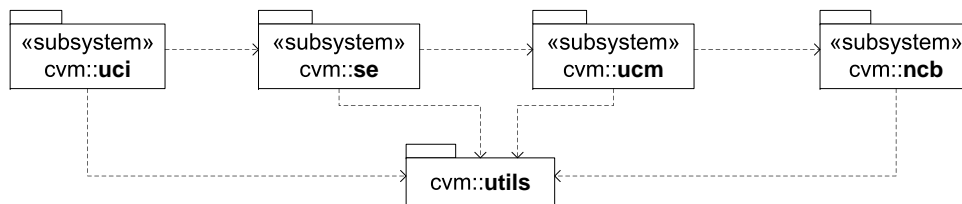


Figure 10. Top level architecture of the CVM prototype.

cvm::uci - the User Communication Interface (UCI) subsystem provides users will the ability to create, load, save and validate CML models. The UCI also passes CML models to SE for realization.

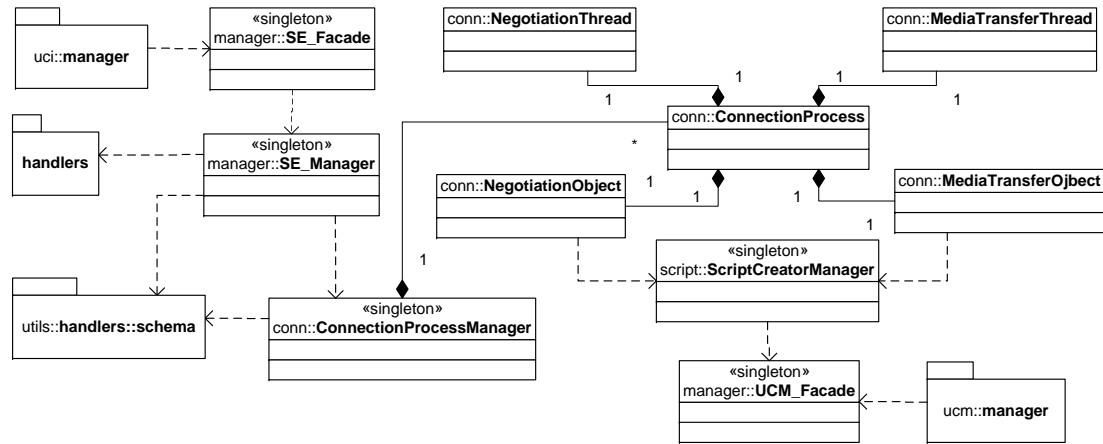


Figure 11. Class diagram for the synthesis engine.

Validation ensures that the models have appropriate values for all the required fields making them communication instances, for example each participant needs to have a valid user id. The main components of the UCI are (1) *Communications Modeling Environment* (CME), a graphical model environment, that provides expert users with the facility to create G-CML domain-specific communication models, mainly control schemas; (2) *User Interface* (UI), a user-friendly interface, that provides casual users with the facility to create communication models, both control and data instances, and (3) *Schema Transformation Environment* (STE) converts and validates models created in CME and UI into X-CML model instances before being passed onto SE. Figure 15 in Appendix A shows a diagram of the model created for the scenario described in Section 2. Figure 16 in Appendix B shows the UI representation of the model for the control instance and two data instances. The X-CML representation of the control instance and two successive data instances is not shown due to space limitations.

cvm::se - the functionality of SE is described in details in Section 5 and will not be revisited in its entirety at this point. Figure 11 shows a class diagram containing the main classes and packages used by SE. SE interfaces with UCI and UCM via the classes *SE.Facade* and *UCM.Facade*, respectively. The *SE.Manager* coordinates the activities of SE and the *ConnectionProcessManager* keeps track of the *ConnectionProcess* objects created per connection. The basic functionality of the *SE.Manager* (same as *SE.Controller* in Figure 6(b)) is described in Algorithm 1. The *ConnectionProcess* class coordinates the activities associated with each connection which includes the (re)negotiation and media transfer processes, shown in Figure 6. The classes *NegotiationObject* and *MediaTransferObject* define the state machines shown in the Tables V and VI, respectively. Schema analysis is performed in the utility package *utils::handlers::schema*. This functionality, described in algorithms *analyze_CI* (Algorithm 2) and *analyze_DI* (Algorithm 3) is, placed in the *utils* package since the UCI also needs to track changes to the control and data instances when updating the UI after receiving updated instances from SE.

cvm::ucm - UCM is designed to support the execution of communication control scripts, including system initialization, macro loading and interpretation, and exception and event handling and runtime media management. UCM handles events from the network communication broker (NCB) as well as internal events. Macros are loaded dynamically by the UCM manager which then delegates the the execution of the macros to the script interpreter. The runtime media management supports temporary storage of non-stream media, handling local user media request, and performing the actions associated with different form types. Wu et al. [29] describe the UCM subsystem used

Table III. Static metrics for the CVM prototype.

| <i>Metrics</i> | <i>utils</i> | <i>uci</i> | <i>se</i> | <i>ucm</i> | <i>ncb</i> | <i>Totals</i> |
|-------------------|--------------|------------|-----------|------------|------------|---------------|
| <i>SLOC</i> | 1,949 | 5,108 | 963 | 737 | 1,659 | 10,416 |
| <i># packages</i> | 5 | 6 | 6 | 5 | 7 | 29 |
| <i># classes</i> | 48 | 192 | 22 | 28 | 59 | 349 |
| <i># methods</i> | 407 | 746 | 156 | 131 | 333 | 1,773 |

in the current version of the prototype, providing additional details on how the functionality is implemented.

cvm::ncb - NCB exposes an API to the UCM that allows it to interact with the underlying communication frameworks/applications. The current version of the prototype interacts with Skype [14] through Java API Skype4Java [40]. NCB is developed using an autonomic architecture that will support other communication frameworks/applications and self-* capabilities. The prototype uses a bridge, NCBBridge, that interacts the Skype adapter, SkypeAdapter. Other adapters under development include Smack [41], JMML [42], and SIP Communicator [43]. The NCB is coordinated by the NCBManager that interacts with the communication services manager CommServicesManger that coordinates the activities of the various adapters that implement the NCBBridge. Allen et al. [31] provide additional details on the implementation of the NCB.

cvm::utils - this package provides support for the operations in other packages. The classes contained in cvm::util are used to analyze a schema since this functionality is required for both the UCI and SE. Schema analysis in the UCI is required when changes are made to the schema in SE and it is then sent to the UCI to be displayed in the UI. Section 5.3 describes the use of the schema analysis in SE. There is also an abstract class defined for event handlers that is used by all the layers in the CVM.

Table VII shows some of the static metrics for the individual layers of the CVM. The first column contains the names of metrics, columns two through six contain the values of the metrics for the packages cvm::utils, cvm::uci, cvm::se, cvm::ucm and cvm::ncb, respectively. The last column on the left of the table contains the totals for the entire CVM. For example, the first row represents the single lines of code (SLOC), where package cvm::utils has 1,949 lines of code and the CVM has 10,416 lines of code. The metrics were obtained using Dependency Finder [44].

7. EVALUATION

To evaluate the effectiveness of realizing CML models using dynamic synthesis we performed a series of experiments using the CVM prototype. The results of the experiments provided us with an estimate of the overhead required to dynamically synthesize CML models during negotiation and media transfer.

7.1. Experiment Setup

Design of Experiments: We deployed CVM on seven machines (desktops and laptops), and on both wired and wireless local area networks. Seven demonstration user accounts were created for the seven end-users required for the communication scenarios. These user accounts, “cvmtester1” to “cvmtester7”, were created in both the CVM and Skype [14]. To evaluate the time required for the overall realization of the communication service and the overhead of synthesizing the communication models, we performed two sets of experiments:

- Experiment Set 1: Measure the execution time overhead for dynamic synthesis in the overall realization process for a set of scenarios, and
- Experiment Set 2: Measure the execution time for the major components of the SE for the same set of scenarios.

Table IV. Average Time for Realizing N-way Audio Conferencing.

| <i>N</i> | <i>Synthesis</i> (<i>SE Driver, UCM Stub</i>) (sec) | <i>Realization</i> (<i>SE Driver</i>) (sec) | <i>Execution</i> (<i>UCM Driver</i>) (sec) | <i>Synthesis</i> <i>Overhead</i> (%) |
|----------|---|---|--|--|
| 2 | 1.1718 | 8.9921 | 7.4123 | 15.24 |
| 3 | 1.8004 | 23.7443 | 21.2950 | 11.27 |
| 4 | 2.0645 | 24.3403 | 22.1056 | 7.94 |
| 5 | 2.4239 | 26.9299 | 23.8750 | 10.52 |
| 6 | 2.7581 | 34.4373 | 32.4534 | 6.46 |
| 7 | 3.1679 | 45.3998 | 43.4514 | 5.24 |

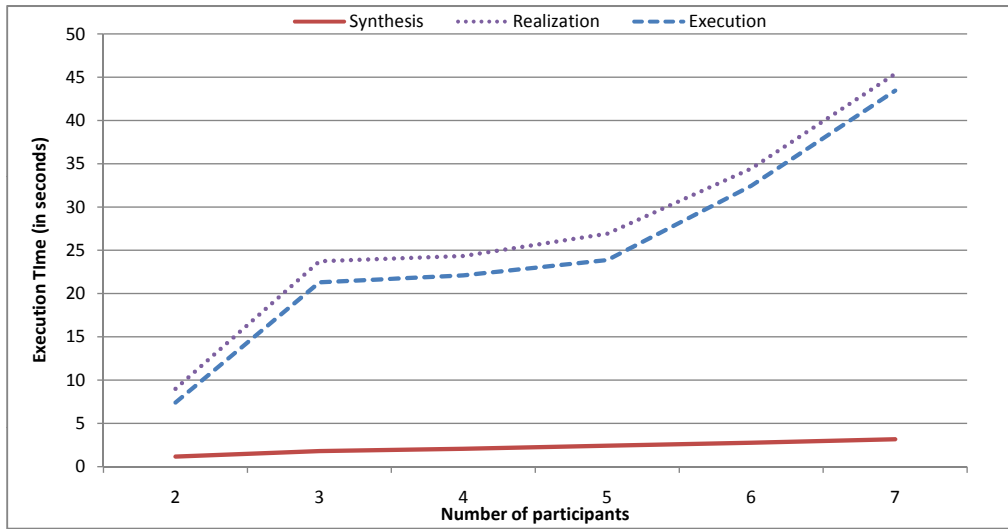
Since the focus of the experiments is on the execution of communication services, as opposed to the modeling and specification of them, we bypassed the User Communication Interface (UCI) layer of the CVM architecture by developing and executing test drivers for the SE and UCM. For Experiment Set 1, we created test drivers for SE, that actually invoked SE and the lower layers of CVM and connected to the remote CVMs. Drivers were also created for the UCM that invoked the NCB and connected to the remote CVMs. For Experiment Set 2, we isolated the synthesis engine by creating test drivers for SE and corresponding stubs for the UCM. To obtain inputs for the test drivers, we ran the scenarios from the user interface and retrieved the X-CML from the log files. The X-CML was used by the test driver and provided as input to SE in order to initiate the specific scenario. The stubs for the UCM were created with low-level events to simulate the response from remote CVMs during the negotiation phase. These events contained the received X-CML schema instances from the remote CVMs during the negotiation.

Choice of Scenarios: The set of communication scenarios that we carried out includes: N-way audio conferencing (where $N = 2$ to 7), video conferencing (currently limited to 2-way due to restrictions of the Skype framework), text messaging, file transfer and form transfer. We also chose scenarios for dynamic re-negotiation, where one participant initiates modifications to the schema instance and results in a consistent updated schema at various end-points. Specifically we report on the results of an N-way audio conference dynamically synthesized to an (N+1)-way conference, where $N = 2$ to 6.

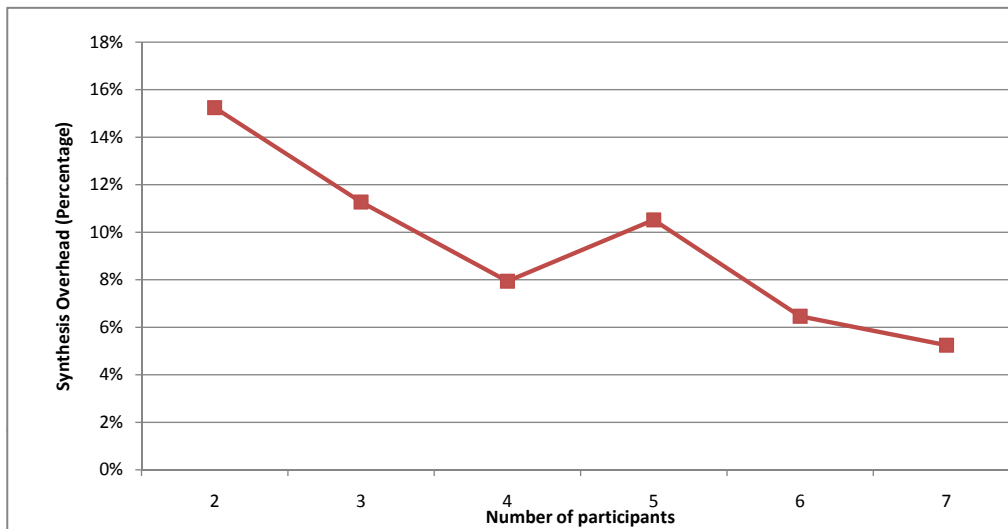
Data Collection: For each of the above scenarios, we performed the two sets of experiments mentioned earlier. In the Experiment Set 1, we evaluated the overhead of realizing CML models by measuring the elapsed time from the initiation of the scenario to the time audio starts streaming on actual networks, including schema negotiation and media transfer. For Experiment Set 2 we measured the elapsed execution time of a stand alone SE for dynamic synthesis (using stubs for lower layers of CVM without incurring actual communication delivery). We also instrumented our prototype with Eclipse Testing & Performance Tooling Platform (TPTP) [45] for detailed execution time analysis. The execution time analysis from TPTP provided us with data on how major SE components of the prototype implementation are affecting the overall performance of the dynamic synthesis process. For each experiment in the set we executed the scenario ten times, discarded the highest and lowest values and took the average of the remaining values.

7.2. Results

The results of the experiments are shown in Table VIII, Figure 12 and Figure 13. Table VIII shows the average execution time, in seconds, for N-way audio conferencing. The columns from left to right are the value of N, number of participants in the conference; time for dynamic synthesis using a standalone SE, i.e., using SE driver and UCM stub; time for realization using a SE driver and remote CVMs; time for execution (realization minus dynamic synthesis) using a UCM driver and remote CVMs; and the overhead of dynamic synthesis as a percentage. The overhead of the dynamic synthesis is computed as a percentage of the ratio of time for synthesis over the time for realization. Recall that overall realization time is measured from the point when the X-CML, for the scenario, is



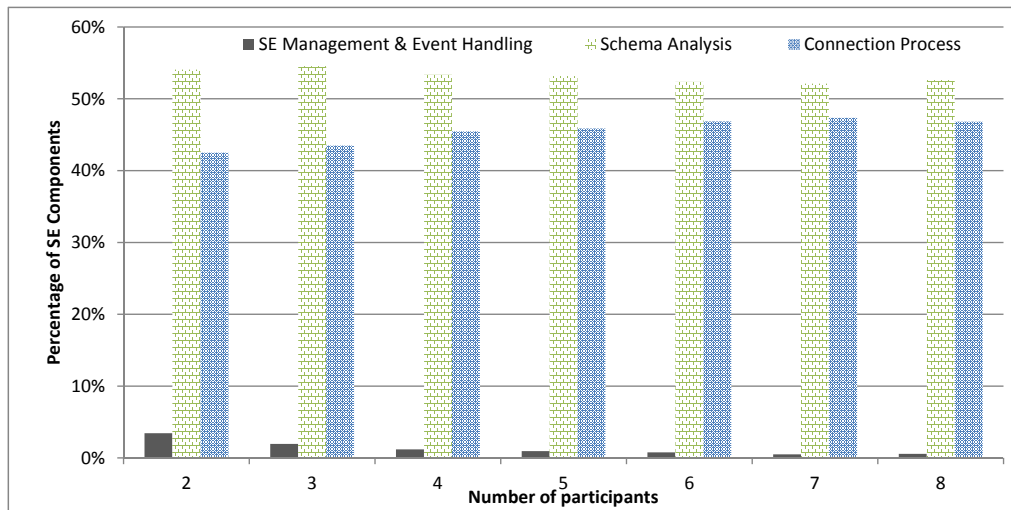
(a)



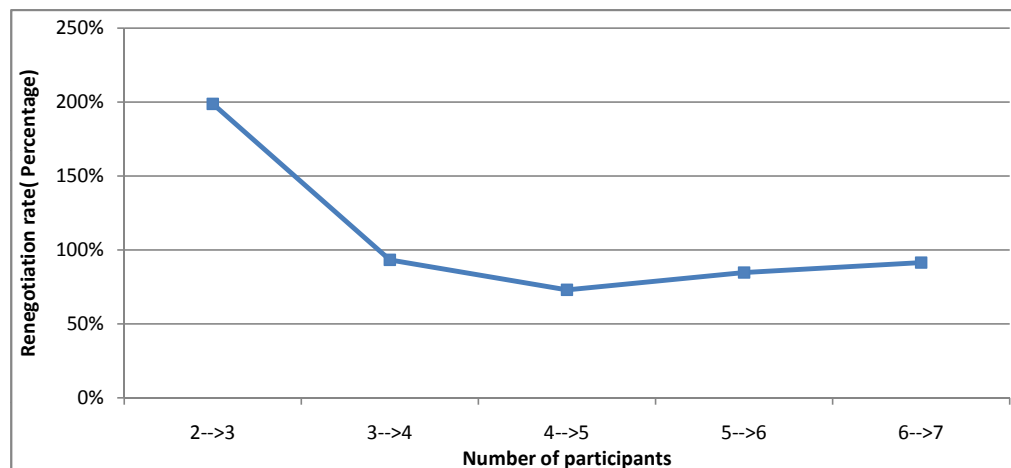
(b)

Figure 12. (a) Average Time for Realizing/Synthesizing N-way Audio Conferencing. (b) Synthesis Overhead(Percentage)

sent to the SE until the point when audio streaming starts. Realization therefore includes the schema negotiation and medium transfer. Figure 12(a) is a graphical representation of the experimental results in Table VIII, where the topmost graph represents the realization time, the middle graph the execution time, and the graph closest to the x-axis the synthesis time. Figure 12(b) shows the overhead for the synthesis process. The overhead for the synthesis process lies between 5% and 15% of the time for realizing the models. As the number of participants in the conference increases the overhead drops except for the 5-way conference.



(a)



(b)

Figure 13. (a) Percentage on each SE component (b)Renegotiation ratio for N-way to (N+1)-way (Percentage)

Figure 13(a) shows the break-down of the execution time for the different components of the synthesis engine. We see that the *Connection Process* component, the rightmost bar in each cluster, and *Schema Analysis* component, the middle bar in each cluster, take up the majority of the execution time. The execution time for the Connection Processes component increases as the number of participants in the audio conference increase, starting at 43% for a 2-way audio conference to 46% for a 7-way audio conference. The execution time for the Schema Analysis component decreases slightly as the number of participants in the audio conference increase, starting at 54% for the 2-way conference to 52% for the 7-way conference. The execution time for the *SE Management and Event Handling* components decreases as the number of participants in the audio conference increases going from 4% for a 2-way conference to less than 1% for the 7-way conference.

Figure 13(b) shows the renegotiation ratio of converting an N-way audio conference to an (N+1)-way audio conference. We define the renegotiation ratio to be the execution time of a renegotiation process (in this case, converting an N-way audio conference to an (N+1)-audio conference) divided by the time for initial negotiation (in this case an N-way audio conference). From the graph in Figure 13(b) we see that as N, the number of participants in the conference, increases the renegotiation drops to under 100%. The highest value for renegotiation is converting from a 2-way to 3-way conference taking twice as long (200%) as the initial 2-way negotiation. The renegotiation for the N-way to the (N+1)-way conference for N = 3 to 6 are all less than the initial negotiation, i.e. between 75% to 90%.

7.3. Discussion

The results of the experiments show that the overall realization time for schema negotiation and media transfer is scalable with respect to the number of participants for N = 2 to 7. The realization process itself is dominated by execution time of the UCM and NCB. It was not practical to perform the experiments for greater than 7 participants in the audio conference due to lack of experimental infrastructure. As can be seen from the graphs in Figure 12(a), dynamic synthesis appears to be scalable as the number of participants in the conference increases, even for N > 7. We can conclude that the overhead for dynamic synthesizing of CML models during schema negotiation and media transfer is negligible in the overall execution cost of the CVM runtime (including SE, UCM and NCB).

The breakdown of the execution time for the major processes in the SE during dynamic synthesis, as shown in Figure 13(a), reveals that the Connection Process and Schema Analysis use over 95% of the time during synthesis. The Connection Process is responsible for handling state changes in schema negotiation and media transfer, and the Schema Analysis process handles the marshaling and demarshaling of schemas and the parsing and population of X-CML (XML documents). The SE Management and Event Handling only take up a small portion (less than 5%) of the total execution time and is responsible for process management (establishing, maintaining and destroying communication processes) and the event handling (receiving and notifying low level events). Using Eclipse TPTP [45] we measured several other metrics not presented in Section 7.2, including the number of threads. There was no change in the number of threads used in the SE. This was expected since all the control instances used in the experiments only contained one connection, with a varying number of participants.

Threats to validity: The threats to our experimentation can be classified under two major headings 1) threats to internal validity, and 2) threats to external validity. Threats to internal validity includes: (1) the occurrence of unexpected network delays and events in actual communication channels; (2) the response delay at various participant end-points, and (3) the instrumentation of the synthesis engine using TPTP as a single measuring instrument appears not to be suitable for the experiments. Using Eclipse TPTP to capture the metrics during execution of the SE resulted in the time for each iteration for a scenario being much longer than anticipated. In addition, the machine on which the experiments were performed had to be restarted several times during the experiments. To mitigate the threats to internal validity, we repeated each scenario ten times, removed the maximum and minimum values and computed the average of the remaining 8 values.

Threats to external validity include: the generalization of the chosen networks, demonstration computers and specific communication frameworks that CVM uses e.g., Skype [14]. To avoid threats to external validity, we chose different machines (both desktops and laptops), in a combination of wired and wireless local area networks. However, the current prototype implementation relies on Skype for actual delivery of communication services, which diminishes the generality of the approach on multiple communication frameworks. Skype does not identify the machine it uses to mix the audio in a conference call and this may have impacted the results for realization times. Part of our future work is to integrate other frameworks such as Smack[41], Asterisk [46] and MSN Messenger [42] into the NCB. We are also working on an approach for dynamic adaptation that would move a conference call from Skype to Asterisk when the numbers of participants in a call reaches a particular threshold. Asterisk is a software implementation of a telephone private branch

exchange (PBX) that supports VoIP services. This adaptation would reduce the high execution time required for conference calls with more than 7 participants as shown in Figure 12(a).

8. CONCLUDING REMARKS

We presented a model-driven approach for the on-demand specification, synthesis and delivery of user-centric communications services. The approach is centered around the Communication Modeling Language (CML), an interpreted domain specific modeling language for modeling user-centric communication services. Communication models (communication schemas) developed using CML are automatically realized using the CVM platform. We applied a systematic approach to the engineering of CML, and discussed its meta-model (abstract syntax and static semantics), behavioral specification and prototypical implementation. Note that CML does not require end-users to be technology or modeling savvy. The user-friendly interface in CVM is suited to casual users and allows pre-defined models to be loaded and realized. For most end-users (e.g., a doctor), the graphical modeling aspect of CVM will be hidden, because the communication models they use will be packaged as pre-defined services. These models will be created by their service providers or domain experts in their organizations (e.g., a hospital).

We also presented an evaluation of our approach through a set of prototypical experiments. Initial experimental results show that dynamic synthesis is an effective approach in that the overhead for dynamic synthesizing of CML models during schema negotiation and media transfer is negligible. Also the overall execution cost is scalable with respect to the number of participants, demonstrating the practicality of our approach.

Several classes of issues including security, performance, and adaption rules are not addressed in the current version of CML. We argue, however, that CML represents a new paradigm for structuring and delivering communication solutions and services, which is more user-centered than the current ways of service specification. The extensible nature of CML allow new features to be seamlessly added as they become a primary concern e.g. the coordination of communication services using workflow concepts.

9. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants HRD-0833093, OISE-0730065, CCF 1018711 and CNS 0854988. Students working on this project were supported in part by a US Dept of Ed. grant P200A070543 and two Florida International University Dissertation Year Fellowships - Yali Wu and Andrew Allen. The authors would like to thank Yingbo Wang for her work on the initial version of CML and CVM. We would also like to thank Professor Jean-Marc Jezequel and the Triskell Research Team at INRIA Rennes for hosting Yali Wu and Andrew Allen during their visit in Summer 2009.

REFERENCES

1. Apple Inc.: Facetime for iphone 4 (2010) <http://www.apple.com/iphone>.
2. Apple Inc.: ipad (2010) <http://www.apple.com/ipad/>.
3. Google: Android developers (2010) <http://developer.android.com>.
4. Krebs, D.: The mobile software stack for voice, data, and converged handheld devices (2005) Mobile and Wireless Practice Venture Development Corporation.
5. Group, T.P.: Parlay/osa specifications (2010) <http://etsi.org/WebSite/Technologies/OSA.aspx>.
6. Sun Microsystems and OpenCloud: Jain-slee (2008) <http://www.jainslee.org/>.
7. The JAIN-SIP Project: Java api for sip signaling (2010) <https://jain-sip.dev.java.net/>.
8. Deng, Y., Sadjadi, S.M., Clarke, P.J., Hristidis, V., Rangaswami, R., Wang, Y.: CVM - a communication virtual machine. *JSS* **81** (2008) 1640–1662
9. Lasserre, P., Kan, D.: User-centric interactions beyond communications. *Alcatel Telecommunications Review* (2005) http://www1.alcatel-lucent.com/doctypes/articlepaperlibrary/pdf/ATR2005Q1/S0503-UCBB_interactions-EN.pdf (March 2007).

10. Wu, Y., Hernandez, F., Ortega, F., Clarke, P.J., France, R.: Measuring the effort for creating and using domain-specific models. In: Proceedings of 10th Domain Specific Modeling Workshop. (2010)
11. Clarke, P.J., Hristidis, V., Wang, Y., Prabakar, N., Deng, Y.: A declarative approach for specifying user-centric communication. In: International Symposium on Collaborative Technologies and Systems, IEEE (2006) 89 – 98
12. Wang, Y., Wu, Y., Allen, A., Espinoza, B., Clarke, P.J., Deng, Y.: Towards the operational semantics of user-centric communication models. In: COMPSAC 09, IEEE (2009) 254–262
13. Burke, R.P., White, J.A.: Internet rounds: A congenital heart surgeon's web log. *Seminars in Thoracic and Cardiovascular Surgery* **16** (2004) 283–292
14. Skype Limited: Skype (2010) <http://www.skype.com/>.
15. User Centric Communications New York, N.: User centric communications (2009)
16. Arbanowski, S., Meer, S.V.D., Steglich, S.: User-centric communications. In: Proceedings of the 8th IEEE International Conference on Telecommunications (ICT 2001). (2001) 425–444
17. Dhara, K.K., Ross, T.I., Krishnaswamy, V.: A framework for developing user-centric services for communication end-points. In: Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE. (2009) 1 –6
18. IBM: The power of unifying communications and collaboration. Unified communications and collaboration White paper (2009)
19. Yelmo, J.C., del Alamo, J.M., Trapero, R., Martin, Y.S.: A user-centric approach to service creation and delivery over next generation networks. *Computer Communications In Press, Corrected Proof* (2010) –
20. Tarkoma, S., Bhushan, B., Kovacs, E., van Kranenburg, H., Postmann, E., Seidl, R., Zhdanova, A.V.: Spice: A service platform for future mobile ims services. In: World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a. (2007) 1–8
21. Shin, Y., Yu, C., Chung, S., Kim, S.: End-user driven service creation for converged service of telecom and internet. *Advanced International Conference on Telecommunications* **0** (2008) 71–76
22. MORFEO open source community: Ezweb project (2010) <http://ezweb.tid.es>.
23. WF-CML Development Team: Workflow communication modeling language (2010)
24. group, T.M.T.: The moda-tel project (2004) <http://www.modatel.org/>.
25. Object Management Group: OMG' Formal Specifications (2010) <http://www.omg.org/>.
26. Network Working Group, C.U.: Call Processing Language (CPL): A Language for User Control of Internet Telephony Services (2004) <http://www.ietf.org/rfc/rfc3880.txt>.
27. Consel, C., Réveillère, L.: A dsl paradigm for domains of services: A study of communication services. In: Domain-Specific Program Generation; International Seminar, Dagstuhl. (2004)
28. Burgy, L., Consel, C., Latry, F., Lawall, J., Palix, N., Réveillère, L.: Telephony software engineering: A domain-specific language approach. Technical report, CCSd/HAL : e-articles server (based on gBUS) [<http://hal.ccsd.cnrs.fr/oai/oai.php>] (France) (2005)
29. Wu, Y., Allen, A.A., Hernandez, F., Wang, Y., Clarke, P.J.: A user-centric communication middleware for cvm. In: The 12th IASTED International Conference on Software Engineering and Applications (SEA 2008), ACTA (2008) 210–215
30. Zhang, C., Sadjadi, S., Sun, W., Rangaswami, R., Deng, Y.: A user-centric network communication broker for multimedia collaborative computing. *Multimedia Tools and Applications* **50** (2010) 335–357 10.1007/s11042-009-0385-6.
31. Allen, A.A., Wu, Y., Clarke, P.J., King, T.M., Deng, Y.: An autonomic framework for user-centric communication services. In: CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research, New York, NY, USA, ACM (2009) 203–215
32. Forum, D.: Domain specific modeling (2010)
33. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Comput. Surv.* **37** (2005) 316–344
34. C.Kang, K., Cohen, S.G., James A. Hess, Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis. Technical Report CMU/SEI-90-TR-21, CMU (1990)
35. Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology* **10** (2002) 2002
36. Group, o.: Object Constraint Language (OCL) (2007)
37. Heckel, R., Voigt, H.: Model-based development of executable business processes for web services. *LNCS* **3098** (2004) 559–584
38. Wang, Y., Clarke, P.J., Wu, Y., Allen, A., Deng, Y.: Runtime models to support user-centric communication. *Models@runtime Workshop in conjunction Models 2008* (2008) <http://www.comp.lancs.ac.uk/~bencomo/MRT/> (Jan. 2009).
39. Triskell Team: Kermeta - breathe life into your metamodels (2009) <http://www.kermeta.org/>.
40. Hisano, K., Lamot, B.: Skype4Java (2010) https://developer.skype.com/wiki/Java_API.
41. Ignite Realtime: Smack api 3.1.0 (2009) <http://www.igniterealtime.org/>.
42. JML Development Team: Java msn messenger library (2010) <http://sourceforge.net/projects/java-jml/>.
43. SIP Communicator Development Team: SIP Communicator (2010) <http://sip-communicator.org/>.
44. Tessier, J.: Dependency Finder (2010) <http://depfind.sourceforge.net/>.
45. The Eclipse Foundation: Eclipse test and performance tools platform (tptp) (2010) <http://www.eclipse.org/tptp/>.
46. Digium: Asterisk (2010) <http://www.asterisk.org/>.

Appendices

A. CML STATIC SEMANTICS

1. **context** ControlSchema
inv: self.communicationID <> null
and self.allInstances->forAll (cs1,cs2| cs1.communicationID<>cs2.communicationID)
and self.attachedParty->size() >1
and self.dataType->forAll(d | d.oclsTypeOf(MediumType) implies
self.conn->exists(c| c.dataTypeRef.exists(df | df.oclsType(MediumTypeRef) and
df.mediumTypeName==d.mediumTypeName) and
and c.device->forAll(d| d.deviceCaps->exists(cap|cap.capability == d.derivedFromBuiltInType))
and self.dataType->forAll(d| d.oclsTypeOf(FormType) implies
self.conn->exists(c| c.dataTypeRef.exists(df | df.oclsType(FormTypeRef) and
df.formTypeName==d.formTypeName) and
and d.subMediumType->forAll (m| c.device->forAll(d| d.deviceCaps->exists(cap| cap.capability ==
d.derivedFromBuiltInType)))
and d.subFormType.flatten->asSet()->forAll(t| t.r.oclsType(MediumTypeRef) implies
c.device->forAll(d|d.deviceCaps->exists(cap|cap.capability==d.derivedFromBuiltInType)))
2. **context** Connection
inv: self.connectionID <> null
and self.device->size() >1
and self.allInstances->forAll (c1,c2| c1.connectionID<>c2.connectionID)
and self.dataTypeRef ->forAll{dr| dr.oclsType(MediumTypeRef) implies self.controlSchema.dataType ->exists (dt|
dt.oclsTypeOf(MediumType) and dt.mediumTypeName == dr.mediumTypeName)}
and self.dataTypeRef ->forAll{dr| dr.oclsType(FormTypeRef) implies self.controlSchema.dataType ->exists (dt|
dt.oclsTypeOf(formType) and dt.formTypeName == dr.formTypeName)}
3. **context** AttachedParty
inv: self.person.personID == self.isAttached.personID
4. **context** IsAttached
inv: self.allInstances->forAll (ia1,ia2| ia1.deviceID<>ia2.deviceID or ia1.personID<>ia2.personID)
and self.controlSchema.conn.device->exists(d|d.deviceID==self.deviceID))
5. **context** MediumType
inv: self.derivedFromBuiltInType <> null and mediumTypeName <> null
6. **context** FormType
inv: self.subFormType->size() + self.subMediumType->size()>0
7. **context** Device
inv: self.deviceID <> null
and self.allInstances->forAll (d1,d2| d1.deviceID<>d2.deviceID)
and self.conn.controlSchema ->exists(ap| ap.isAttached.deviceID== self.deviceID)
and self.allInstances->select(d| d.isLocal==true)->size() ==1
8. **context** Person
inv: self.personID <> null
and self.allInstances->forAll (p1,p2| p1.personID<>p2.personID)
9. **context** DataSchema
inv: self.dataContent->size()+ self.request->size() >0
10. **context** Form
inv: self.allInstances->forAll (f1,f2| f1.formID<>f2.formID)
and self.subForm->size() +self.subMedium->size()>0
11. **context** Medium
inv: self.allInstances->forAll (m1,m2| m1.mediumName<>m2.mediumName)

Figure 14. CML Static Semantics.

B. G-CML FOR THREE-WAY CONFERENCE SCENARIO

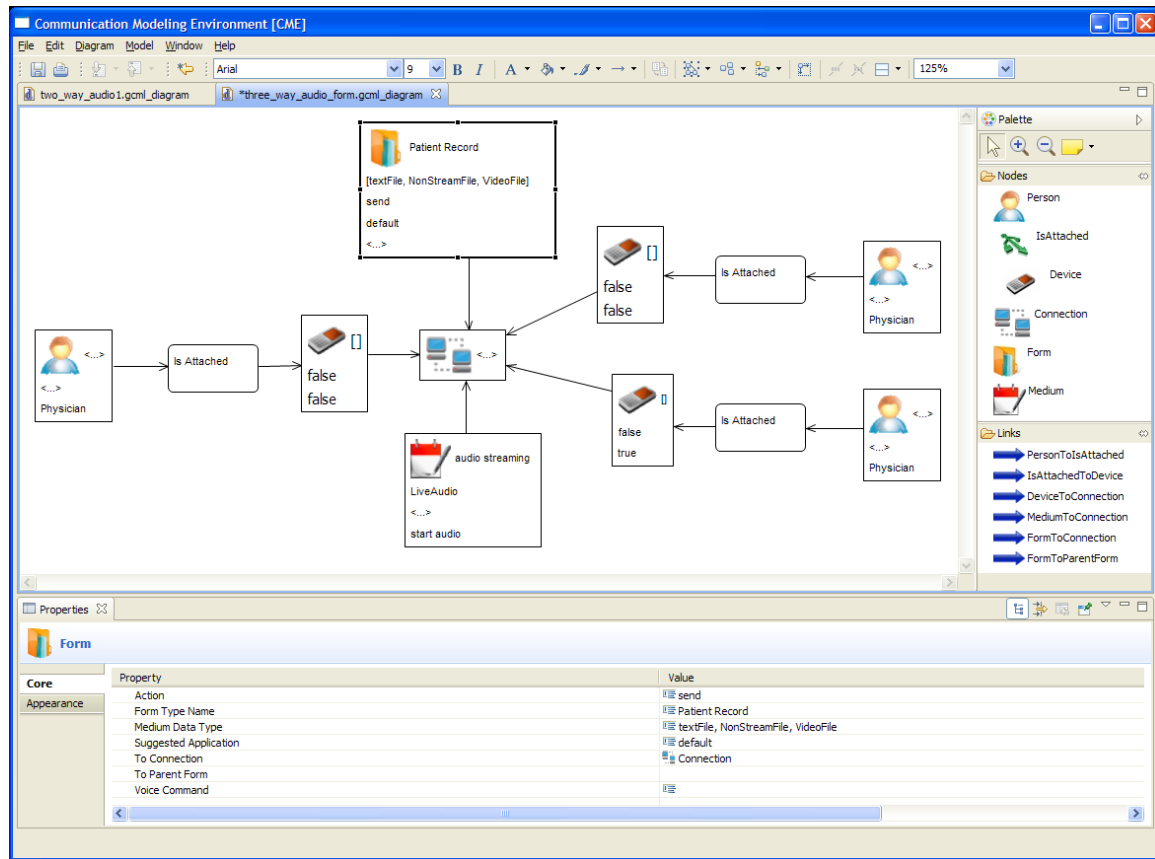


Figure 15. G-CML model created using the CVM prototype for supporting the scenario presented in Section 2. The model shown represents the scenario after Dr. Burke invites Dr. Sanchez to join the conversation with Dr. Monteiro.

C. THREE-WAY CONFERENCE SCENARIO IN USER-FRIENDLY GUI

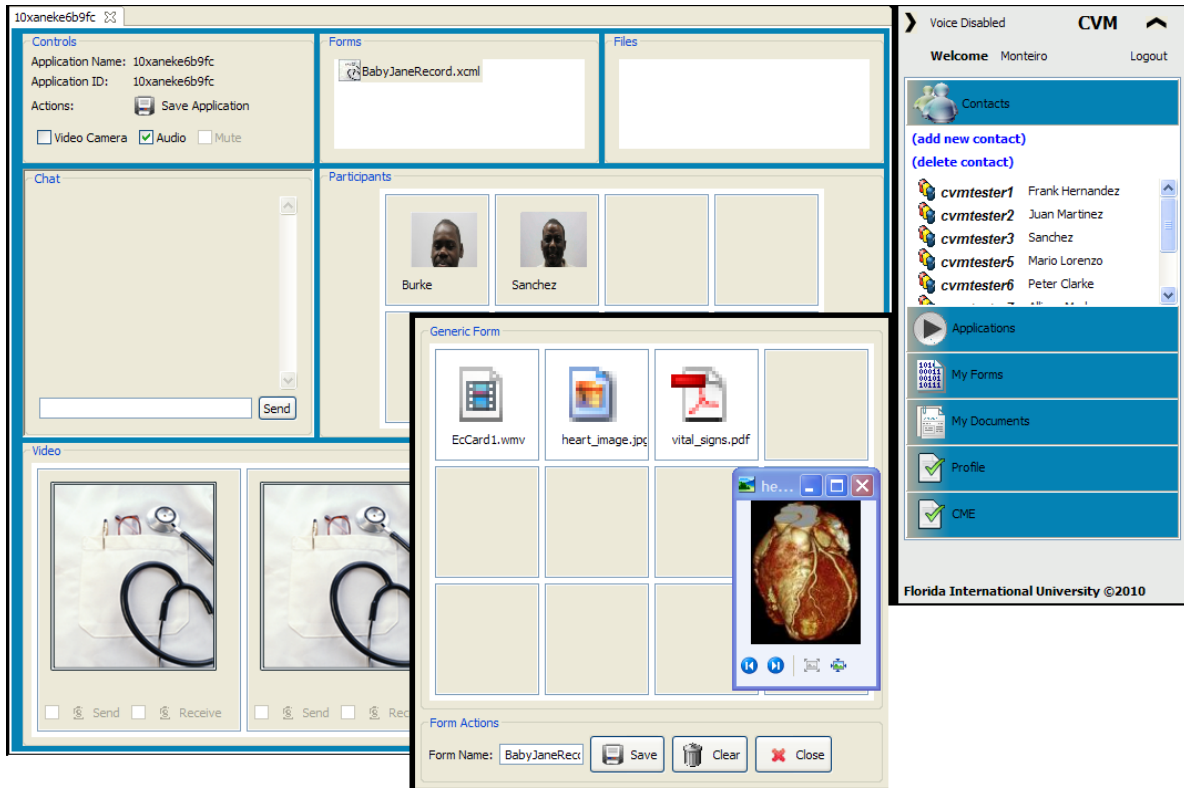


Figure 16. Dr. Monteiro's user interface showing the heart image of patient Baby Jane, during the conference with Dr. Burke and Dr. Sanchez.

D. X-CML FOR THREE-WAY CONFERENCE SCENARIO

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<controlSchema communicationID="_YCWW0DE4Ed-D97h4qLpucg">
  <connection bandwidth="c1" connectionID="_JePiQDE6Ed-D97h4qLpucg">
    <device isLocal="false" isVirtual="true" deviceID="_MGxqwDE6Ed-D97h4qLpucg">
      <deviceCapability>TextFile</deviceCapability>
      <deviceCapability>LiveAudio</deviceCapability>
      <deviceCapability>VideoFile</deviceCapability>
      <deviceCapability>NonStreamFile</deviceCapability>
    </device>
    <device isLocal="false" isVirtual="true" deviceID="_TFz5cDE6Ed-D97h4qLpucg">
      <deviceCapability>TextFile</deviceCapability>
      <deviceCapability>VideoFile</deviceCapability>
      <deviceCapability>NonStreamFile</deviceCapability>
      <deviceCapability>LiveAudio</deviceCapability>
    </device>
    <device isLocal="true" isVirtual="false" deviceID="_WNyXADE6Ed-D97h4qLpucg">
      <deviceCapability>TextFile</deviceCapability>
      <deviceCapability>LiveAudio</deviceCapability>
      <deviceCapability>NonStreamFile</deviceCapability>
      <deviceCapability>VideoFile</deviceCapability>
    </device>
    <mediumTypeNameRef>audio streaming</mediumTypeNameRef>
    <formTypeNameRef>patient record</formTypeNameRef>
  </connection>
  <mediumType voiceCommand="start audio" derivedFromBuiltInType="LiveAudio"
    mediumTypeName="audio streaming"/>
  <formType action="send" suggestedApplication="default" formTypeName="patient record">
    <mediumDataType>textFile</mediumDataType>
    <mediumDataType>NonStreamFile</mediumDataType>
    <mediumDataType>VideoFile</mediumDataType>
  </formType>
  <person personRole="Surgeon" personID="burke23" personName="burke"/>
  <person personRole="Attending Physician" personID="monteiro41" personName="monteiro"/>
  <person personRole="Referring Physician" personID="sanchez12" personName="sanchez"/>
  <isAttached deviceID="_WNyXADE6Ed-D97h4qLpucg" personID="burke23"/>
  <isAttached deviceID="_MGxqwDE6Ed-D97h4qLpucg" personID="monteiro41"/>
  <isAttached deviceID="_TFz5cDE6Ed-D97h4qLpucg" personID="sanchez12"/>
</controlSchema>

```

Figure 17. X-CML control instance for the three-way communication between Dr. Burke, Dr. Monteiro and Dr. Sanchez.

Table V. State machine for schema (re)negotiation.

| <i>T.</i> | <i>Source_State</i> | <i>Target_State</i> | <i>Event</i> | <i>Guard</i> | <i>Action</i> |
|-----------|---------------------|---------------------|------------------|---------------------------------|---|
| 0 | Initial | NegReady | | | |
| 1 | NegReady | NegInitiated | initiateNeg | hasNegToken | addNegBlock(CI_{neg}) genConnection_Script |
| 2 | NegReady | NegInitiated | initiateReNeg | hasNegToken | addNegBlock(CI_{neg}) |
| 3 | NegInitiated | WaitingSameCI | | # remoteParty != 0 | genSendCS_Script |
| 4 | WaitingSameCI | WaitingSameCI | localSameCI | # responses < # remoteParty | |
| 5 | WaitingSameCI | NegComplete | localSameCI | # responses == # remoteParty | genSendCS_Script |
| 6 | NegComplete | NegReady | | | $CI_{exe} \leftarrow CI_{neg}$ UCI.notify(CI_{exe}) |
| 7 | WaitingSameCI | WaitingAnyCI | localChangeCI | # responses < # remoteParty | update(CI_{neg}) |
| 8 | WaitingSameCI | NegComplete | localChangeCI | # responses == # remoteParty | update(CI_{neg}) genSendCS_Script |
| 9 | WaitingAnyCI | WaitingAnyCI | localSameCI | # responses < # remoteParty | |
| 10 | WaitingAnyCI | WaitingAnyCI | localChangeCI | # responses < # remoteParty | update(CI_{neg}) |
| 11 | WaitingAnyCI | NegComplete | localSameCI | # responses == # remoteParty | genSendCS_Script |
| 12 | WaitingAnyCI | NegComplete | localChangeCI | # responses == # remoteParty | update(CI_{neg}) genSendCS_Script |
| 13 | WaitingSameCI | WaitingAnyCI | after t sec. | # remoteParty > 1 | update(CI_{neg}) |
| 14 | WaitingAnyCI | WaitingAnyCI | after t sec. | # remoteParty > 1 | update(CI_{neg}) |
| 15 | WaitingSameCI | NegTerminateInit | after t sec. | # remoteParty == 1 | |
| 16 | WaitingAnyCI | NegTerminateInit | after t sec. | # remoteParty == 1 | |
| 17 | NegReady | NegRequested | inviteNeg | | notifyUCIInviteNeg |
| 18 | NegRequested | NegTermRemote | UCI.rejectInvite | | genRejectInvite_Script |
| 19 | NegTermInit | Final | | | UCI.notify(CI_{exe}) genCloseConnect_Script |
| 20 | NegTermRemote | Final | | | |
| 21 | NegRequested | WaitingConfirm | UCI.acceptInvite | | genConnection_Script genSendCS_Script |
| 22 | WaitingConfirm | NegComplete | remoteSameCI | | |
| 23 | WaitingConfirm | WaitingConfirm | remoteChangeCI | | update(CI_{neg}) genSendCS_Script |
| 24 | WaitingConfirm | NegTermRemote | after t sec. | | UCI.notify(CI_{exe}) genCloseConnect_Script |
| 25 | NegReady | SelfRemoved | removeSelf | hasNegToken | genRemoveSelf_Script |
| 26 | NegReady | NegReady | removeParty | # remoteParty > 1 | update(CI_{neg}) $CI_{exe} \leftarrow CI_{neg}$ genRemoveParty_Script |
| 27 | NegReady | Final | removeParty | # remoteParty == 1 | genCloseConnect_Script UCI.notify(CI_{exe}) |
| 28 | SelfRemoved | Final | | | |

Table VI. State machine for media transfer.

| <i>Tr.</i> | <i>Source_State</i> | <i>Target_State</i> | <i>Event</i> | <i>Guard</i> | <i>Action</i> |
|------------|---------------------|---------------------|----------------------------------|--|--|
| 0 | Initial | Ready | initiateNeg initiateInviteNeg | | |
| 1 | Ready | StreamEnabled | enableStream | | genStreamEnable_Script |
| 2 | Ready | StreamEnabled | enableStreamRec | | genStreamEnableRec_Script UCL.notify(DS_{i+1}) |
| 3 | StreamEnabled | StreamEnabled | enableStream | !IsStreamEnabled | genStreamEnable_Script |
| 4 | StreamEnabled | StreamEnabled | disableStream | IsStreamEnabled && # streams > 1 | genStreamDisable_Script |
| 5 | StreamEnabled | StreamEnabled | enableStreamRec | !IsStreamEnabled | genStreamEnableRec_Script UCL.notify(DS_{out}) |
| 6 | StreamEnabled | StreamEnabled | disableStreamRec | IsStreamEnabled && # streams > 1 | genStreamDisableRec_Script UCL.notify(DS_{out}) |
| 7 | StreamEnabled | StreamEnabled | sendNonStream | | genNonStreamSend_Script |
| 8 | StreamEnabled | StreamEnabled | sendForm | | genSendForm_Script |
| 9 | StreamEnabled | StreamEnabled | recNonStream | | UCL.notify(DS_{out}) |
| 10 | StreamEnabled | StreamEnabled | recForm | | UCL.notify(DS_{out}) |
| 11 | StreamEnabled | Ready | disableStream | # streams == 1 | genCloseStream_Script |
| 12 | StreamEnabled | Ready | disableStreamRec | # streams == 1 | genCloseStreamRec_Script UCL.notify(DS_{out}) |
| 13 | Ready | Ready | sendNonStream | | genNonStreamSend_Script |
| 14 | Ready | Ready | sendForm | | genSendForm_Script |
| 15 | Ready | Ready | recNonStream | | UCL.notify(DS_{out}) |
| 16 | Ready | Ready | recForm | | UCL.notify(DS_{out}) |
| 17 | Ready | Final | terminate | | |

Table VII. Static metrics for the CVM prototype.

| <i>Metrics</i> | <i>utils</i> | <i>uci</i> | <i>se</i> | <i>ucm</i> | <i>ncb</i> | <i>Totals</i> |
|-------------------|--------------|------------|-----------|------------|------------|---------------|
| <i>SLOC</i> | 1,949 | 5,108 | 963 | 737 | 1,659 | 10,416 |
| <i># packages</i> | 5 | 6 | 6 | 5 | 7 | 29 |
| <i># classes</i> | 48 | 192 | 22 | 28 | 59 | 349 |
| <i># methods</i> | 407 | 746 | 156 | 131 | 333 | 1,773 |

Table VIII. Average Time for Realizing N-way Audio Conferencing.

| <i>N</i> | <i>Synthesis</i> (<i>SE Driver, UCM Stub</i>) (sec) | <i>Realization</i> (<i>SE Driver</i>) (sec) | <i>Execution</i> (<i>UCM Driver</i>) (sec) | <i>Synthesis</i> <i>Overhead</i> (%) |
|----------|---|---|--|--|
| 2 | 1.1718 | 8.9921 | 7.4123 | 15.24 |
| 3 | 1.8004 | 23.7443 | 21.2950 | 11.27 |
| 4 | 2.0645 | 24.3403 | 22.1056 | 7.94 |
| 5 | 2.4239 | 26.9299 | 23.8750 | 10.52 |
| 6 | 2.7581 | 34.4373 | 32.4534 | 6.46 |
| 7 | 3.1679 | 45.3998 | 43.4514 | 5.24 |