Towards the Operational Semantics of User-Centric Communication Models

Yingbo Wang, Yali Wu, Andrew Allen, Barbara Espinoza, Peter J. Clarke and Yi Deng
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
{*ywang002, ywu001, aalle004, bespi009, clarkep, deng*}*@cis.fiu.edu*

***Abstract*—The pervasiveness of complex communication services and the need for end-users to play a greater role in developing communication services have resulted in the creation of the Communication Virtual Machine (CVM) technology. The CVM technology consists of a Communication Modeling Language (CML) and the CVM. CML is a declarative modeling language that can be used to specify domain-specific communication services and the CVM is the platform used to realize the CML models.**

In this paper we explicitly define the operational semantics of CML to support (1) the synthesis of CML models into executable control scripts and (2) the handling of negotiation and media transfer events during communication. We specify the semantics of CML using label transition systems and describe in detail an algorithm that is essential for the interpretation of CML models. A case study is presented showing how the semantics support the rapid realization of a scenario from the healthcare domain.

***Keywords*-Collaborative Networks, User-Centric Communication, Model-Driven Development**

I. Introduction

The pervasiveness of electronic communication such as IP telephony, instant messaging and video conferencing has resulted in the need for a new approach to developing user-centric communication applications. The need is further magnified by the use of these technologies in specialized communication applications for telemedicine, disaster management and scientific collaboration [1], [2], [3]. The traditional stovepipe approach of developing communication intensive application binds the user-level communication logic with device types and underlying networks. Rapidly changing network capabilities and communication devices result in new communication needs. Unfortunately, existing applications cannot cater to unanticipated communication requirements without the development of systems resulting in high cost and a lengthy development cycle.

In response to the need to rapidly develop user-centric communication applications Deng et al. [4] created a new paradigm based on the Communication Virtual Machine (CVM) technology for modeling and rapidly realizing user-centric communication services. We use the term *user-centric* to refer to those applications that provide services to the user, offer operating simplicity, and mask the complexity of the underlying technology [5]. We limit the scope of the term communication in this paper to denote the exchange of electronic media of any format (e.g., file, video, voice) between a set of participants (humans or agents) over a network (typically IP). The development process uses models created using the Communication Modeling Language (CML). CML models capture the user communication requirements and are automatically realized using the CVM. The time and cost of developing communication applications can be largely reduced by using the CVM platform for formulating, synthesizing and executing new user-centric communication services.

The current version of CML [6] lacks a complete set of operational semantics resulting in the CVM being limited to realizing simple static communication models. In this paper, we investigate the operational semantics of CML with respect to the synthesis process in CVM presented in [4]. The contributions of this paper include:

- Defining the behavioral models for the operational semantics of CML to support (1) the synthesis of CML models into executable control scripts and (2) the handling of negotiation and media transfer events during communication.
- Defining a detailed algorithm to analyze CML models during model realization.
- Describing the synthesis of a scenario from the healthcare domain.

The rest of the paper is organized as follows. Section II introduces the CVM technology. Section III defines the operational semantics for the synthesis of CML models. Section IV details how the operational semantics are applied to a medical scenario and states the limitations of our approach. Section V presents the related work and we conclude in Section VI.

II. CVM Technology

In this section we provide background on the CVM technology [4]. The technology consists of CML [6], used to model user-centric communication requirements, and CVM, the platform to realize user communication models.

A. Communication Modeling Language

There are currently two equivalent variants of CML: the XML-based (X-CML) and the graphical (G-CML). The

1. *communicationSchema* ::= *controlSchema* | *dataSchema*
2. *controlSchema* ::= *partyLocal connection {connection}*
3. *connection* ::= **connection** *dataType {dataType} partyRemote {partyRemote}*
4. *partyLocal* ::= **person isAttached** *device*
5. *partyRemote* ::= *device* **isAttached person**
6. *device* ::= **device deviceCapability {deviceCapability}**
7. *dataType* ::= **mediumType** | *formType*
8. *formType* ::= **formTypeHeader** *dataType {dataType}* **formTypeEnd**
9. **connection** ::= connectionID_A bandwidth_A
10. **person** ::= personName_A personID_A personRole_A
11. **isAttached** ::= deviceID_A personID_A
12. **device** ::= deviceID_A
13. **deviceCapability** ::= builtinType_A
14. **mediumType** ::= mediumTypeName_A $\text{derivedFromBuiltInType}_A$ $\text{suggestedApplication}_A$ voiceCommand_A
15. **formTypeHeader** ::= formTypeName_A $\text{suggestedApplication}_A$ voiceCommand_A action_A
16. *dataSchema* ::= **connection** *data {data}* | **connection request**
17. **connection** ::= connectionID_A bandwidth_A
18. *data* ::= **medium** | *form*
19. *form* ::= **formHeader** *data {data}* **formEnd**
20. **medium** ::= mediumDataType_A mediumName_A mediumURL_A mediumSize_A lastModifTime_A validityPeriod_A $\text{firstTransferTime}_A$ voiceCommand_A
21. **formHeader** ::= formDataType_A formID_A $\text{suggestedApplication}_A$ voiceCommand_A action_A $\text{layoutSpecification}_A$
22. **request** ::= requestID_A mediumName_A action_A

Figure 1. EBNF representation of X-CML.

primitive communication operations that can be modeled by CML include: (1) connection establishment, (2) data (primitive and user-defined) transfer, (3) addition/removal of participants to/from a communication, (4) dynamic creation of structured data, and (5) specification of properties associated with a particular data transfer. Figure 1 shows a simplified version of X-CML using EBNF notation. The EBNF notation represents an attributed grammar where attributes are denoted using an "A" subscript, terminals are bold face and non-terminals are in italics. This version of CML is an extension of the one presented in [4].

Two categories of communication models can be described using CML, *communication schemas* and *communication instances*. The relation between a schema and an instance is similar to the relation between a class and an object in programming languages. An instance captures all information in a communication at a particular point in time and can be directly executed. On the other hand, a schema describes the possible communication configurations of a conforming instance. Rule 1 in Figure 1 defines a communication schema as either a *control schema* or a *data schema*. A control schema (Rule 2) specifies the configuration required to set up one or more connections in a communication and the data schema (Rule 16) specifies the media to be transfered across a connection at an instance in time. In Section IV we present a medical scenario and the associated G-CML communication instance.

B. Communication Virtual Machine

CVM [4] provides an environment that supports the modeling and realization of user-centric communication services. The CVM architecture divides the major communication tasks into four major levels of abstraction, which correspond to the four key components of CVM: (1) *User Communication Interface (UCI)*, provides a modeling environment for users to specify their communication requirements using CML; (2) *Synthesis Engine (SE)*, generates an executable script (*communication control script*) from a CML model and negotiates the model with other participants in the communication; (3) *User-centric Communication Middleware (UCM)*, executes the communication control script to manage and coordinate the delivery of communication services to users; (4) *Network Communication Broker (NCB)*, provides a network-independent API to UCM and works with the underlying network protocols to deliver the communication services.

C. Realizing a Communication Model

Figure 2(a) shows the flow of execution when a communication model is realized by the CVM. Execution starts when the user submits a CML model to be executed, this model is validated and converted into a *control schema* (CS) and a *data exchange schema* (DS) pair. The (CS, DS) is then passed to the SE where it is analyzed and converted into a control script to be executed by the UCM. The UCM executes the script and makes API calls to the NCB that interfaces with the underlying communication frameworks e.g., Skype [7] or Smack [8]. The NCB interacts with the communication frameworks and generates UCM or SE events that are handled by their respective CVM layers. Updates to an executing schema are sent to the UCI to be displayed to the user.

In this paper we focus on the operational semantics of CML models with respect to the SE. Figure 2(b) provides an overview of the actions performed by the SE in order to realize use-centric communication. The three major processes of the SE (shown in ovals) are *schema analysis*, *(re)negotiation* and *media transfer* [9]. The (re)negotiation and media transfer processes, enclosed in the dashed line, are created per connection. There are two components shown in the figure that support the activities of these processes, *SE Controller* - coordinates incoming schemas and events, and *SE Dispatcher* - coordinates outgoing events and scripts.

The schema analysis process accepts as input a schema consisting of a (CS, DS) pair from the UCI or from the UCM via an SE event. The schema is compared to the current

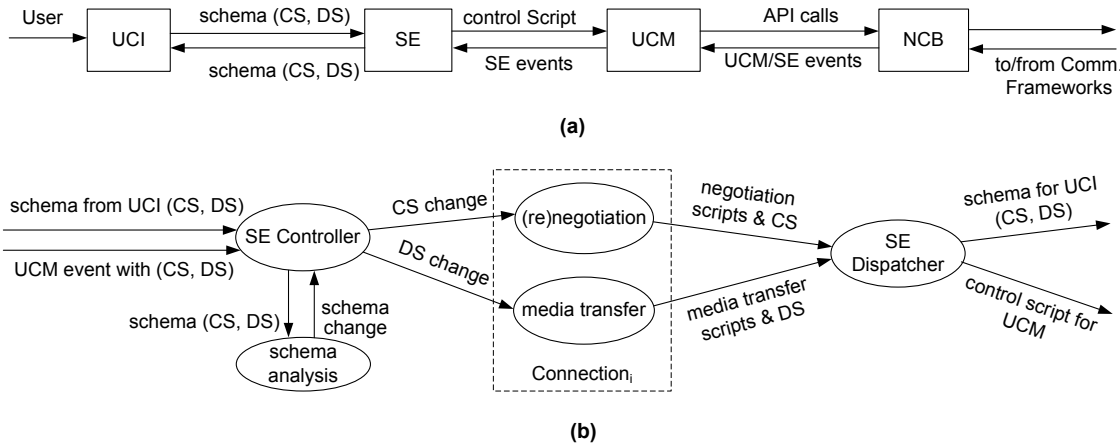

Figure 2. (a) Execution of a schema in the CVM. (b) Execution of schema in the synthesis engine (SE). CS - Control schema; DS - Data schema

schema in the SE, which may be null schema, and the results passed to the SE Controller. Based on the type of CS event generated the (re)negotiation process is started or the current negotiating process is updated. Similarly, a DS event may cause a media transfer process to start or be updated. Both the (re)negotiation and media transfer processes generate control scripts to be processed by the UCM or schemas to be processed by the UCI. Additional details of the SE are presented in [4], [10].

Figure 3 shows a grammar for the control script language using EBNF. Note the grammar is not complete, e.g., there are no type definitions or brackets for the parameters. Rule 1 states that a control script consists of one or more commands and Rule 2 shows the various script commands. The strings in bold represent the actual command and the attributes represent the parameters the command can take. For example, Rule 3 states that *createConnectionCmd* is composed of the string **createConnection** and takes one parameter consisting of a connection id. We have defined the complete metamodel definitions for CML and the control script language using ECore [11], however for readability purposes we show them using an attributed EBNF grammar.

III. Operational Semantics for Synthesis

In this section we describe the operational semantics required to realize user-centric communication. We do not use the inference rule notation to describe the semantics but describe them using labeled transition systems [12] represented in tabular form. Recall SE is responsible for executing the semantic-rich CML model, which includes schema negotiation and renegotiation, and media transfer.

A. Overview of Synthesis

We describe the operational semantics of CML models during synthesis as a set of model transformations from the metamodel of CML (input to SE) to the metamodel of the control script (output of SE). Figures 1 and 3 show the metamodels for CML and the control script language, respectively. In addition to the transformation from CML models to control scripts, the synthesis process also involves the use of special control commands from the UCI. Examples of these special control commands are *login, logout* and *accept/reject connection*. In this paper we do not explicitly model these commands as input to the transformation process.

The behavior associated with a connection in a communication scenario is specified as a sequence of schema pairs of the form (CS_i, DS_i), where $i = 0, 1 \cdots n$, *CS* is a control schema and *DS* a data exchange schema. We define a connection as a link between participants in the same communication space. The initial schema pair (CS_0, DS_0) represents the initial state of the system with respect to some new connection to be established. That is, CS_0 and DS_0 both represent null schemas. The schema pair (CS_1, DS_1) represents the schema pair that carries the control schema (CS_1) with the initial configuration for a new connection. Note that in this schema pair the data schema (DS_1) is null.

Since the behavior for a given communication scenario is captured in a sequence of schema pairs the operational semantics will also be defined based on schema pairs. We will define the operational semantics on the schema pairs $\{(CS_i, DS_i), (CS_{i+1}, DS_{i+1})\}$ for a specific connection. The pair (CS_{i+1}, DS_{i+1}) represents input from either the UCI or UCM and the pair (CS_i, DS_i) represents previous schemas processed by the SE and stored in the SE environment (Env_i). We therefore define the operational semantics of the CML models as a set of transformations $(\Longrightarrow)$ defined as follows:

$$((CS_{i+1}, DS_{i+1}), Env_i) \Longrightarrow \\ ((CS_{out}, DS_{out}), Script_{i+1}, Env_{i+1})$$

where:
(CS_{i+1}, DS_{i+1}) - input schema pair from the UCI or UCM.
Env_i - current environment of the SE consisting of:

1. *controlScript* := *command {command}*
2. *command* := *createConnectionCmd* | *closeConnectionCmd* | *addParticipantCmd* | *removeParticipantCmd* | *sendSchemaCmd* | *enableMediaInitiatorCmd* | *enableMediaReceiverCmd* | *disableMediaInitiatorCmd* | *disableMediaReceiverCmd* | *sendMediaCmd* | *sendFormCmd* | *declineConnectionCmd* | *requestFormCmd* | *requestMediaCmd* | *sendNegTokenCmd* | *requestNegTokenCmd*
3. *createConnectionCmd* := **createConnection** connectionID_A
4. *closeConnectionCmd* := **closeConnection** connectionID_A
5. *addParticipantCmd* := **addParticipant** connectionID_A personID_A {personID_A}
6. *removeParticipantCmd* := **removeParticipant** connectionID_A personID_A {personID_A }
7. *sendSchemaCmd* := **sendSchema** connectionID_A sender-personID_A receiver-personID_A{receiver-personID_A} schema_A
8. *enableMediaInitiatorCmd* := **enableInitiatorMedia** connectionID_A mediaName_A
9. *enableMediaReceiverCmd* := **enableReceiverMedia** connectionID_A mediaName_A
10. dis*ableMediaInitiatorCmd* := **disableInitiatorMedia** connectionID_A mediaName_A
11. dis*ableMediaReceiverCmd* := **disableReceiverMedia** connectionID_A mediaName_A
12. *sendMediaCmd* := **sendMedia** connectionID_A mediaName_A mediumURL_A
13. *sendFormCmd* := **sendForm** connectionID_A formID_A mediumURL_A {mediumURL_A} action_A
14. *declineConnectionCmd* := **declineConnection** sender-personID_A receiver-personID_A{receiver-personID_A}
15. *requestFormCmd* := **requestForm** connectionID_A formID_A mediumURL_A {mediumURL_A}
16. *requestMediaCmd* := **requestMedia** connectionID_A mediaName_A
17. *sendNegTokenCmd* := **sendNegToken** personID_A
18. *requestNegTokenCmd* := **requestNegToken** connectionID_A

Figure 3. EBNF for the control script.

(CS_i, DS_i) - CS and DS in the SE that is used for comparison with (CS_{i+1}, DS_{i+1}), where $CS_i \in \{CS_{neg}, CS_{exe}\}$, CS_{neg} is the CS currently being negotiated and CS_{exe} is the currently executing CS, i.e., the most recently negotiated CS. DS_i is the currently executing DS.

Neg_i - current state of the SE with respect to (re)negotiation and includes CS_{neg}

MT_i - current state of the SE with respect to media transfer and includes DS_i.

(CS_{out}, DS_{out}) - schema pair generated during the transformation process. This pair contains the CS and DS schemas that may be sent to the UCI.

$Script_{i+1}$ - control script sent to the UCM for processing and defined using the EBNF shown in Figure 3.

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

```
1:  analyze_CS (ref CS_{i+1}, ref Env_i, sourceCS)
    /*Input: CS_{i+1} - schema from the UCI or UCM
             Env_i - current environment object
             sourceCS - source of CS, UCI or UCM
       Output: ccs, an object with CS changes and an event trigger */
2:  ccs ← compare(CS_{i+1}, Env_i.CS_i)
3:  if sourceCS == UCI then
4:      if ccs.enum ∈ {initialCS} then
5:          ccs.addEvent(initiateNeg)
            /* SE Controller uses this event to start the state
            machines (SMs) for negotiation and media transfer
            and passes the initiateReNeg to the negotiation SM */
6:      else if ccs.enum ∈ {selfRemoved} then
7:          ccs.addEvent(removeSelf)
8:      else
9:          ccs.addEvent(initiateReNeg)
10:     end if
11: else if sourceCS == UCM && self.isInitiator then
12:     if ccs.enum ∈ {no_Change} then
13:         ccs.addEvent(localSameCS)
14:     else
15:         ccs.addEvent(localChangeCS)
16:     end if
17: else
18:     /*sourceCS == UCM && !self.isInitiator */
19:     if Env_i.CS_i == null then
20:         ccs.addEvent(intiateInviteNeg)
            /* SE Controller uses this event to start the state
            machines (SMs) for negotiation and media transfer
            and passes the inviteNeg to the negotiation SM */
21:     else if ccs.enum ∈ {no_Change} then
22:         ccs.addEvent(remoteSameCS)
23:     else if ccs.enum ∈ {partyRemoved} then
24:         ccs.addEvent(removeParty)
25:     else
26:         ccs.addEvent(remoteChangeCS)
27:     end if
28: end if
29: return ccs
```

Figure 4. Algorithm to analyze CS.

B. Schema Analysis

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 2(b). The algorithm for analyzing a schema is composed of two sub-algorithms, these are (1) `analyze_CS` for analyzing CSs, shown in Figure 4, and (2) `analyze_DS` for analyzing DSs, shown in Figure 5. The input parameters to both algorithms include a schema from the SE Controller (CS_{i+1} or DS_{i+1}), a reference to the current environment of the SE (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 two schemas, including the event that triggers the transitions in the state machines for (re)negotiation and media transfer.

The *compare* function, line 2, in both algorithms computes the change between the new schema and current schema and stores them in the object `ccs`, for CSs, or `cds`, for DSs. The fields in this object include an enumeration that specifies the category of change, e.g., `initialCS` as shown on line 4 in Figure 4, among other information used during the execution of the state machines. Applying the `analyze_CS` algorithm to the input parameters CS_1 - the new schema

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

Figure 5. Algorithm to analyze DS.

for a connection, (Env_0) - containing - CS_0 the null CS schema, and the source of CS_1 is the UCI, results in a `ccs` object being generated with `initialCS` as the enumerated change. The `initialCS` change results in `analyze_CS` returning `initiateNeg` as the trigger event field in the `ccs` object to the SE Controller. The SE Controller uses the contents of the `ccs` or `cds` object to either (1) create the initial state machines for (re)negotiation and media transfer, and/or (2) send the object to the executing state machine to trigger the appropriate transition(s) and be used during the execution of the specified actions.

C. Negotiation

The state machine for (re)negotiation is created by the SE Controller when the `initiateNeg` is returned as an event trigger field in the `ccs` object from `analyze_CS`. The (re)negotiation state machine works independently of the media transfer state machine, both are implemented as threads. Once the (re)negotiation state machine is created all other objects returned from `analyze_CS` are sent directly by the SE Controller to the executing state machine. Table I shows the state machine for schema (re)negotiation. The table has 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. For example, transition 1 between the source state `NegReady` and the target state `NegInitiated` is triggered by the `initiateReNeg` event, assuming that the environment has the negotiating token (`hasNegToken` is true). As a result of the transition being triggered a negotiation block is added to the new control schema, `addNegBlock(`CS_{i+1}`)`, and a script to create the connection is generated, `genConnection_Script`.

In Table I we use the following notations 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(`CS_{i+1}`)` - block in the schema that keeps data associated with the negotiation process, e.g., sender's id, negotiation initiator's id.
- `genXXX_Script` - generates the XXX control script. Recall a control script may contain one or more script commands.
- `update(`CS_{i+1}`)` - updates the schema being negotiated based on changes such as, removal of a participant or removing self from the schema.
- `UCI.notify(`CS_{i+1}`)` - send a CS to the UCI to inform the user of the state of the negotiation.

The **Initial** and **Final** states are shown in bold. Note that transition 5 results in an action that replaces the current schema with the negotiated schema (CS_{i+1}). In Section IV we show examples of the control scripts for (re)negotiation in Table III.

D. Media Transfer

The media transfer state machine, shown in Table II, is similar in structure to the table for (re)negotiation. Although we do not show it in Table II the executing DS is updated for transitions 1 through 12, i.e., the entry $DS_i \leftarrow DS_{i+1}$ should be in the *Action* column. We use a similar notation for the guards and actions as shown below:

- `streamEnabled` - is a boolean that represents if the live stream (audio, video, audio-video) specified in DS_{i+1} is enabled, see the predefined types for CML in [4]
- `# streams` - represent the number of active live streams.
- `UCI.notify(`DS_{i+1}`)` - 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 `cds` returned from the `analyze_DS` contains the information required by the state machine to trigger transitions and perform actions. For example, transition 1 in Table II 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 end of the communication. In next section we show examples of the control scripts for media transfer.

Table I
STATE MACHINE FOR SCHEMA NEGOTIATION.

Trans.	*Source_State*	*Target_State*	*Event*	*Guard*	*Action*
0	**Initial**	NegReady	initiateNeg || intiateInviteNeg		
1	NegReady	NegInitiated	initiateReNeg	hasNegToken	addNegBlock(CS_{i+1}) genConnection_Script
2	NegInitiated	WaitingSameCS		# remoteParty != 0	genSendCS_Script
3	WaitingSameCS	WaitingSameCS	localSameCS	# responses $<$ # remoteParty	
4	WaitingSameCS	NegComplete	localSameCS	# responses $==$ # remoteParty	genSendCS_Script
5	NegComplete	NegReady			$CS_{exe} \leftarrow CS_{i+1}$ UCI.notify(CS_{i+1})
6	WaitingSameCS	WaitingAnyCS	localChangeCS		update(CS_{i+1})
7	WaitingAnyCS	WaitingAnyCS	localSameCS	# responses $<$ # remoteParty	
8	WaitingAnyCS	WaitingAnyCS	localChangeCS	# responses $<$ # remoteParty	update(CS_{i+1})
9	WaitingAnyCS	NegInitiated		# responses $==$ # remoteParty	update(CS_{i+1})
10	WaitingSameCS	WaitingAnyCS	after 5 sec.	# remoteParty $>$ 1	update(CS_{i+1})
11	WaitingAnyCS	WaitingAnyCS	after 5 sec.	# remoteParty $>$ 1	update(CS_{i+1})
12	WaitingSameCS	NegTerminateInit	after 5 sec.	# remoteParty $==$ 1	
13	WaitingAnyCS	NegTerminateInit	after 5 sec.	# remoteParty $==$ 1	
14	NegReady	NegRequested	inviteNeg		notifyUCI_InviteNeg
15	NegRequested	NegTerminateRemote	UCI.rejectInvite		genRejectInvite_Script
16	NegTerminateInit	**Final**			UCI.notify(CS_{i+1}) genCloseConnect_Script
17	NegTerminateRemote	**Final**			
18	NegRequested	InviteAccepted	UCI.acceptInvite		genConnection_Script
19	InviteAccepted	WaitingConfirm			genSendCS_Script
20	WaitingConfirm	NegComplete	remoteSameCS		UCI.notify(CS_{i+1})
21	WaitingConfirm	InviteAccepted	remoteChangeCS		update(CS_{i+1})
22	WaitingConfirm	NegTerminateRemote	after 5 sec.		UCI.notify(CS_{i+1}) genCloseConnect_Script
23	NegReady	SelfRemoved	removeSelf	hasNegToken	genRemoveSelf_Script
24	NegReady	NegReady	removeParty	# remoteParty $>$ 1	update(CS_{i+1}) $CS_{exe} \leftarrow CS_{i+1}$ genRemoveParty_Script
25	NegReady	PartyRemoved	removeParty	# remoteParty == 1	genCloseConnect_Script
26	PartyRemoved	**Final**			UCI.notify(CS_{i+1})
27	SelfRemoved	**Final**			

IV. APPLYING SEMANTICS TO THE SCENARIO

In this section we describe a scenario from the healthcare domain and show how the synthesis process realizes a user-centric communication service.

A. Domain Specific Scenario

The authors have been collaborating with members of the cardiology division of Miami Children's Hospital (MCH) over the last 3 years to study the applications of the CVM technology in healthcare. One such scenario involves post-surgery consultation between Dr. Burke - heart surgeon, Dr. Monteiro - attending physician and Ms. Smith - attending nurse.

Scenario: After performing surgery on patient baby Jane, Dr. Burke returns to his office and establishes a live audio/video communication with Dr. Monteiro and Ms. Smith to discuss the post-surgery care for the patient. Dr. Burke dynamically creates the post-surgery medical record containing a text summary of baby Jane's vital signs and the echocardiogram (echo) of her heart captured during surgery and shares it with Dr. Monteiro and Ms. Smith while discussing the post-surgery care. After the discussion Dr. Burke terminates the communication. □

Figure 6 shows several of the G-CML models generated during communication for the scenario. Figure 6 part (a) shows the CS for the scenario and parts (b) and (c) show successive DSs used during the communication. We do not show all the fields in the schemas only those that help in the presentation. In addition, the various versions of the CSs used during termination of the communication are not shown. We assume the system is initialized with the null CS and null DS. The G-CML shown in Figure 6(a) represents a pre-defined schema that Dr. Burke loads into the user interface for novice users, see the screen shots in [4, page 1656].

B. Synthesizing the Model

During the synthesis of the models for the scenario several control scripts are generated, these scripts are shown in the leftmost column of Table III. The first column in the table shows the user id of the SE on which the control script is

Table II
STATE MACHINE FOR MEDIA TRANSFER.

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

executed and the second column the transitions executed in the (re)negotiation and media transfer state machines. The table is divided into three sections, the initial negotiation between the participants (Dr. Burke (burke23), Ms. Smith (smith48) and Dr. Monteiro (monteiro41)), the media transfer between the participants and the final negotiation to close the communication. The media transfer section of Table III has two entries initiated by Dr. Burke corresponding to the two data schemas shown in Figure 6 parts (b) and (c).

The first row in Table III, under the label Negotiation, shows the actions taken by the SE on Dr. Burke's CVM after it receives the initial control schema from the UCI. These actions involve creating the negotiation state machine (transition 0 triggered by event `initiateNeg`) followed by establishment of the connection to start negotiation (transition 1 triggered by event `initiateReNeg`). The control script generated includes the `createConnection("C1")` and `addParticipant("C1", "smith48, monteiro41")` commands that informs the UCM to create a connection with id `"C1"` and add the participants with ids `"smith48"` and `"monteiro41"` to the connection. The entry "NA" in the table states that no control script is generated as a result of the transitions shown.

C. Limitations of Approach

The approach presented in the paper has several limitations with respect to the completeness of the semantics. These limitations include (1) the incomplete semantics for SE Controller, (2) details of the semantics related to the token used in negotiation, and (3) the details on updating the control schema. The SE Controller is the process that creates and destroys the negotiation and media transfer processes and requires the use of a priority queue to handle the request from the UCI and the SE events received from the UCM. In addition, the semantics will have to specify concurrency and synchronization details. We are currently still validating aspects of the SE in the CVM prototype and are currently working to define the correct semantics. We have defined the semantics for the operations related to negotiation token and the different updates that can be performed on the control script, however due to space restriction we could not provide the details in the paper.

The current version of CML can model various communication scenarios that involve multiple human participants. These include: a single connection with multiple participants, as shown in Figure 6, and multiple connections each with multiple participants. The current version of CML cannot model communication scenarios that involve explicitly defined workflows. G-CML suffers from the same problem associated with most graphical modeling languages, that is, there is reduced readability with models that contain a large number of nodes. The limitation with respect to scalability is based mainly on the services provided to the NCB from the communication frameworks such as Skype [7] or Smack [8]. However, this limitation is somewhat ameliorated since the NCB uses self-management principles to self-configure the communication frameworks based on user-defined policies [13].

V. RELATED WORK

Defining operational semantics in the context of modeling languages like UML is not new. Butler et al [14] used

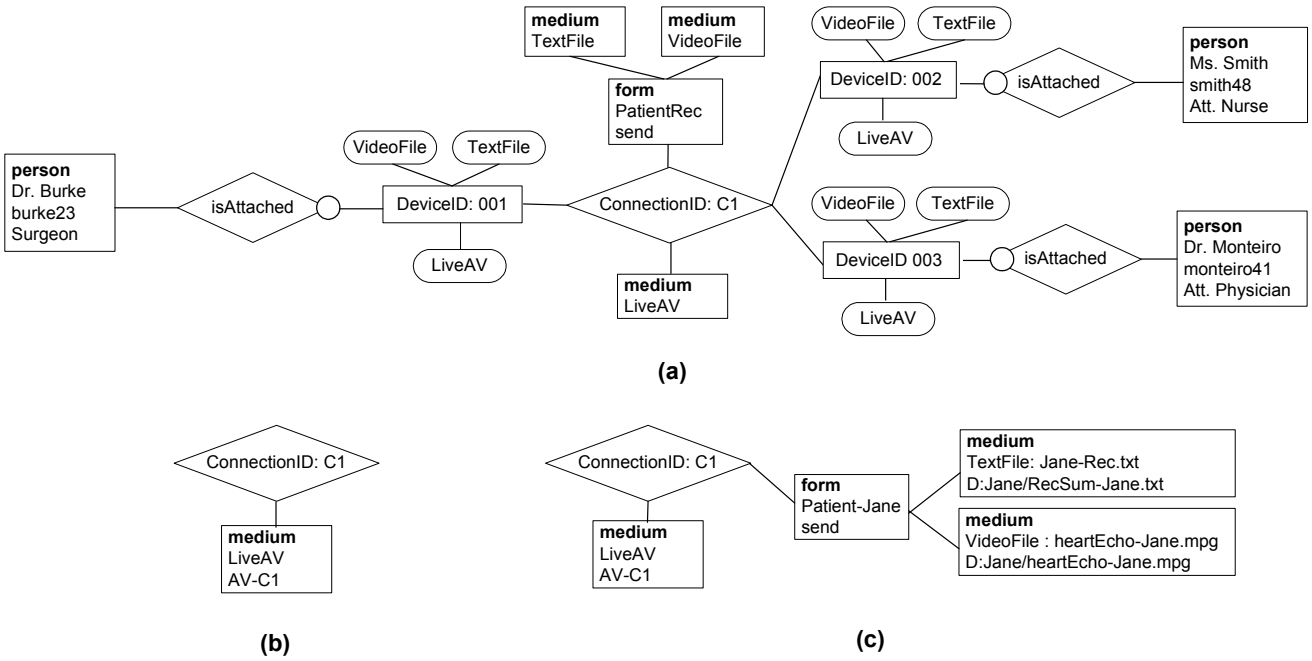

Figure 6. (a)The initial CS shown as a G-CML model for the scenario. (b) and (c) Successive models.

a similar approach to formalize stAC (a business process modeling language similar to BPEL4WS) by defining the system in terms of transition rules between configurations, and using activities as transition labels in configuration (state) transitions. However, their semantics are based on a semantic language (a variant of stAC) for which every language construct is mapped to a transition rule. The declarative nature of CML requires our semantics to include a set of schema analysis algorithms whose output triggers state transitions in the labeled transition systems (schema negotiation and media transfer state machines).

van Eijk et al. [15] studied the operational semantics of agent communication languages in multi-agent systems. These semantics are defined by transition rules that describe its operational behavior, giving rise to an abstract machine that interprets the language. However, the authors focus on defining a formal transitional system as a theoretical basis for their language, providing no details regarding the abstract machine used to interpret the language. Our operational semantics blend the formal definition with detailed algorithms and state machines in a practical manner, thus facilitating rapid model realization that conforms to specified behaviors.

Singh et. al [16] proposed a new ordering semantics for communication middleware and protocols allowing the application to provide a specification, which would be delegated and enforced by the underlying multicast layer. Our work has similar motivations, except that we have provided full executable semantics for the communication application specified in CML, not just a particular feature of it, say message ordering.

In the work by Deng et al. [4] the authors describe the syntax for two versions of CML and the detailed architecture of the CVM. The paper also describes a prototype that was developed to show the feasibility and practicality of the CVM technology. In this paper we provide additional details for the synthesis of CML models resulting in the generation of control scripts that are used by the UCM to realize a communication. These details include how the behavior of a user-defined communication scenario is determined based on a sequence of CML models created by the user. During the process of defining the semantics for the synthesis process we extended two major aspects of the CVM technology model. These extensions include (1) the syntax of CML to explicitly define the data schema, and (2) additional commands in the control script to support the data schema language extensions.

VI. Conclusion

In this paper, we define the semantics for a communication model (communication schema) written using a declarative communication modeling language (CML). The semantics for this model consist of schema synthesis and negotiation/communication management. An algorithm for CML schema analysis and state machines supporting CML operational semantics are provided. Our future work will focus on extending CML to include user defined workflows and the semantics to support such models.

Acknowledgment

This work was supported in part by the National Science Foundation under grant HRD-0833093.

Table III
CONTROL SCRIPTS GENERATED BY SE FOR THE HEALTHCARE SCENARIO.

SE user's id	*Trans.*	*Control Scripts*
Negotiation:		
burke23	0, 1	createConnection("C1"); addParticipant("CS_1", "smith48, monteiro41")
burke23	2	sendSchema("C1","burke23","smith48, monteiro41", "CS_1, null")
smith48	0, 14, 18	createConnection("C1"); addParticipant("C1", "burke23, monteiro41")
smith48	19	sendSchema("C1","smith48","burke23", "CS_1, null")
monteiro41	0, 14, 18	createConnection("C1"); addParticipant("C1", "burke23, smith48")
monteiro41	19	sendSchema("C1","monteiro41","burke23", "CS_1, null")
burke23	3, 4, 5	sendSchema("C1","burke23","smith48, monteiro41", "CS_2, null")
smith48	20, 5	NA
monteiro41	20, 5	NA
Media Transfer:		
burke23	0, 1	enableInitiatorMedia("C1", "LiveAV"); enableRecieverMedia("C1", "LiveAV"); sendSchema("C1","burke23 ","smith48, monteiro41", "CS_2, DS_1")
smith48	0, 1	enableRecieverMedia("C1", "LiveAV"); enableInitiatorMedia("C1", "LiveAV")
monteiro41	0, 1	enableRecieverMedia("C1", "LiveAV"); enableInitiatorMedia("C1", "LiveAV")
burke23	8	sendForm("C1", "Patient-Jane", "D:Jane/RecSum-Jane.txt"); sendForm("C1", "Patient-Jane", "D:Jane/heartEcho-Jane.mpg"); sendSchema("C1","burke23 ","smith48, monteiro41", "CS_2, DS_2")
smith48	10	NA
monteiro41	10	NA
Negotiation:		
burke23	23, 27	sendSchema("C1","burke23","smith48, monteiro41", "CS_3, null"); closeConnection("C1")
smith48	24	removeParticipant("C1","burke23")
monteiro41	24	removeParticipant("C1","burke23")
smith48	23, 27	sendSchema("C1","smith48","monteiro41", "CS_4, null"); closeConnection("C1")
monteiro41	25, 26	removeParticipant("C1","smith48"); closeConnection("C1")

REFERENCES

[1] R. P. Burke and J. A. White, "Internet rounds: A congenital heart surgeon's web log," *Seminars in Thoracic and Cardiovascular Surgery*, vol. 16, no. 3, pp. 283–292, 2004.

[2] FEMA, "DisasterHelp," http://www.disasterhelp.gov/start.shtm (May 2008).

[3] Cyberbridges, "Center for Internet Augmented Research and Assessment," http://www.cyberbridges.net/archive/summary.htm (Nov2007).

[4] Y. Deng, S. M. Sadjadi, P. J. Clarke, V. Hristidis, R. Rangaswami, and Y. Wang, "CVM - a communication virtual machine," *JSS*, vol. 81, no. 10, pp. 1640–1662, 2008.

[5] P. Lasserre and D. Kan, "User-centric interactions beyond communications," Alcatel Telecommunications Review, Quarter 1, 2005.

[6] P. J. Clarke, V. Hristidis, Y. Wang, N. Prabakar, and Y. Deng, "A declarative approach for specifying user-centric communication," in *CTS*. IEEE, May 2006, pp. 89 – 98.

[7] Skype Limited, "Skype developer zone," Feb. 2007, https://developer.skype.com/.

[8] Ignite Realtime, "Smack api 3.1.0," Jan. 2009, http://www.igniterealtime.org/.

[9] Y. Wang, P. J. Clarke, Y. Wu, A. Allen, and Y. Deng, "Runtime models to support user-centric communication," Models@runtime Workshop in conjunction Models 2008.

[10] R. Rangaswami, S. M. Sadjadi, N. Prabakar, and Y. Deng, "Automatic generation of user-centric multimedia communication services," in *IPCCC*, April 2007, pp. 324–331.

[11] The Eclipse Foundation, "Eclipse modeling framework," http://www.eclipse.org/modeling/emf/ (Jan. 2009).

[12] G. D. Plotkin, "A Structural Approach to Operational Semantics," University of Aarhus, Tech. Rep. DAIMI FN-19, 1981, citeseer.ist.psu.edu/plotkin81structural.html.

[13] A. A. Allen, S. Leslie, Y. Wu, P. J. Clarke, and R. Tirado, "Self-configuring user-centric communication services," in *IEEE ICONS*, 2008, pp. 253–259.

[14] M. J. Butler and C. Ferreira, "An operational semantics for StAC, a language for modelling long-running business transactions," in *COORDINATION 2004*, 2004, pp. 87–104.

[15] R. M. van Eijk, F. S. de Boer, W. ven der Hoek, and J.-J. C. Meyer, "Operational semantics for agent communication languages," in *Issues in Agent Communication*. Springer-Verlag: Heidelberg, Germany, 2000, pp. 80–95.

[16] G. Singh, "Exploiting application semantics in communication middleware." [Online]. Available: citeseer.ist.psu.edu/487418.html