Runtime Models to Support User-Centric Communication

Yingbo Wang, Peter J. Clarke, Yali Wu, Andrew Allen, and Yi Deng

School of Computing and Information Sciences
Florida International University
11200 SW 8th Street
Miami, FL 33199, USA
{ywang002,clarkep,ywu001,aalle004,deng}@cis.fiu.edu
http://www.cis.fiu.edu/

Abstract. The pervasiveness of complex communication services and the need for end-users to play a greater role in modeling communication services have resulted in the development of the Communication Modeling Language (CML). CML is a domain-specific modeling language that can be used to declaratively specify user-centric communication services. CML models are automatically realized using the Communication Virtual Machine (CVM). The dynamic nature of end-user driven communication results in communication models being updated at runtime. This paper focuses on CML runtime models in the Synthesis Engine (SE), a layer in CVM, which is responsible for synthesizing these models into executable control scripts. We describe how the CML models are maintained at runtime and how they can evolve during the realization of a communication service.

Key words: Communication Model, Model Realization, Model evolution, Runtime

1 Introduction

Electronic communications have become pervasive in recent years. The improvement in network capacity and reliability facilitates the development of communication intensive services and applications. These applications range from IP telephony, instant messaging, video conferencing, to specialized communication applications for telemedicine, disaster management and scientific collaboration [1,2,3]. Deng et al [4] investigated a new technology for developing and rapidly realizing user-centric communication services to respond to increasing communication needs. 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 a domain-specific modeling language, the *Communication Modeling Language* (CML), which is supported by an automated model realization platform, the *Communication Virtual Machine* (CVM). The time and cost of developing communication services can be significantly reduced by using the CVM platform for formulating, synthesizing and executing new communication services.

A key part of rapidly realizing communication services is that users can change CML models during execution. In addition, several models associated with the communication service being realized can exist at runtime. These issues raise the question of how to maintain CML runtime models and evolve them in a seamless manner without affecting the current executing communication services. In this paper, we focus on handling CML runtime models in the Synthesis Engine (SE) to address the challenges of model evolution as well as model execution. SE is a layer in CVM, which is responsible for transforming CML models into executable control scripts. These control scripts are executed by the User-Centric Communication Middleware (UCM), a layer below the SE in the CVM. The functionalities of SE includes: (1) maintaining and evolving CML models at runtime, (2) the parsing and interpretation of CML models, and (3) the generation of control scripts.

The rest of the paper is organized as follows. Section 2 introduces the CVM technology. Section 3 presents a motivating scenario form the healthcare domain. Section 4 describes the approach used to manipulate communication models at runtime. Section 5 presents the related work and we conclude in Section 6.

2 Modeling and Realizing Communication Services

In this section we introduce the technology to support the model creation and realization of user-centric communication services.

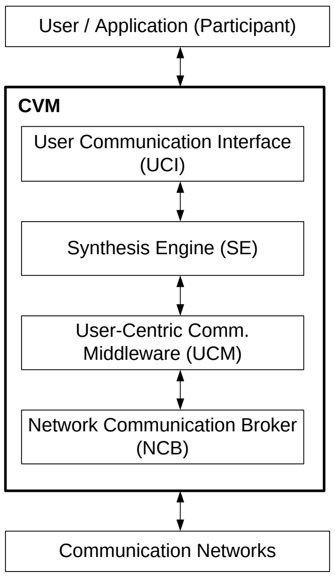
2.1 Communication Modeling Language (CML)

Clarke et al. [5] developed a language, *Communication Modeling Language (CML)*, for modeling user-centric communication services. There are currently two equivalent variants of CML: the XML-based (X-CML) and the graphical (G-CML). The primitive communication concerns that can be modeled by control CML include: (1) participant, (2) attached device, (3) connection, and (4) data, including simple medium and structured data, which can be transferred. Figure 1(a) 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.

A CML model is referred to as a *communication schema* or simply *schema*. A schema consists of two parts: the *control schema* (CS) part which specifies an instance of a topology (participant ids and the types of the exchanged media), and the *data exchange schema* (DS) part which specifies actual media (name or urls) to be exchanged across each connection. We refer the interested reader to [5] for more details.

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

(a)


User / Application (Participant)
CVM
User Communication Interface (UCI)
Synthesis Engine (SE)
User-Centric Comm. Middleware (UCM)
Network Communication Broker (NCB)
Communication Networks

(b)

Fig. 1. (a) EBNF representation of X-CML. (b) Layered architecture of CVM

2.2 Communication Virtual Machine (CVM)

The Communication Virtual Machine (CVM) [4] provides an environment that supports the model creation and realization of user-centric communication services. Figure 1(b) shows the layered architecture of the CVM. 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)*, which provides a language environment for users to specify their communication requirements in the form of a schema using X-CML or G-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, independent of the underlying network configuration; (4) *Network Communication Broker (NCB)*, which provides a network-independent API to UCM and works with the underlying network protocols to deliver the communication services.

3 Motivating Scenario

The authors have been collaborating with members of the cardiology division of Miami Children's Hospital (MCH) to study the applications of the CVM tech-

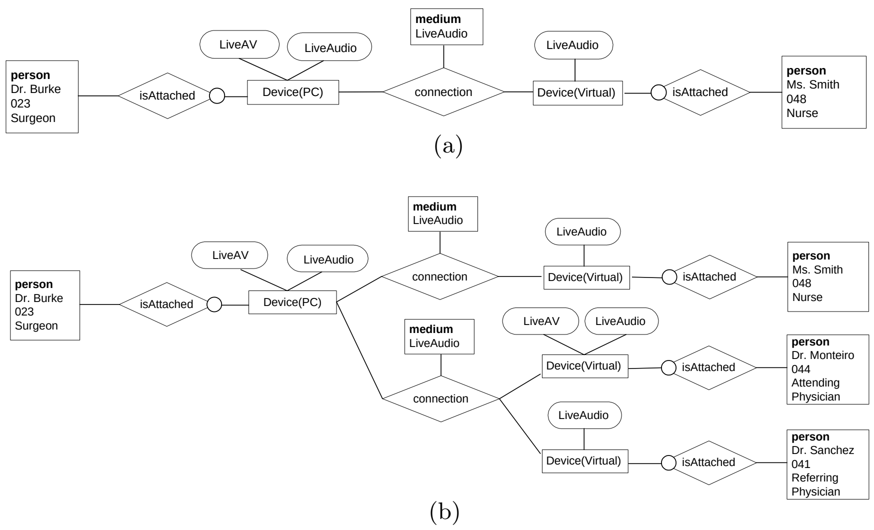


Fig. 2. G-CML models for the scenario. (a) Initial G-CML model of communication between Dr. Burke and Nurse Smith, and (b) G-CML model after all communication connections are established.

nology in the healthcare domain. One of the scenarios we reviewed is described below.

Scenario: After performing surgery on a patient, Dr. Burke (surgeon) returns to his office and establishes an audio communication with Ms. Smith (nurse) to discuss the post-surgery care for the patient. During the conversation with Ms. Smith, Dr. Burke establishes an independent video communication with Dr. Monteiro (attending physician) to obtain critical information for the post-surgery care of the patient. Dr. Burke later decides to invite Dr. Sanchez (referring physician) to join the conference with Dr. Monteiro to discuss aspects of the post-surgery care. Dr. Sanchez's communication device does not have video capabilities resulting in only an audio connection being used in the conference between Dr. Burke, Dr. Monteiro and Dr. Sanchez.

Figure 2 shows two of the three G-CML models created by Dr. Burke during the execution of the scenario. Figure 2(a) shows Dr. Burke's initial request for audio communication with Ms. Smith and Figure 2(b) shows the final G-CML model after Dr. Sanchez is added to the communication. Due to space limitations we do not show the intermediate G-CML model containing only Dr. Burke, Ms. Smith and Dr. Monteiro. A video clip of a similar scenario can be accessed at `http://www.cis.fiu.edu/cml/` that shows an interface for novice users.

4 CML Runtime Models

In this section we provide an overview of how a CML model (schema) is realized by the CVM and the process of synthesizing a schema in the SE. In addition,

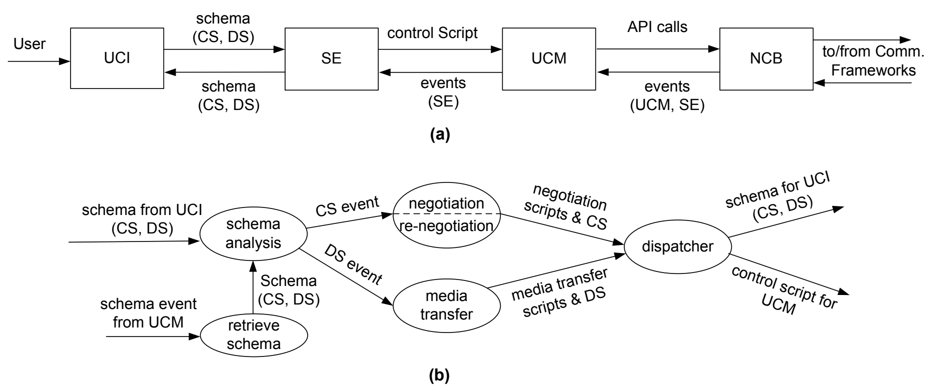


Fig. 3. (a) Execution of a schema in the CVM. (b) Execution of a schema in the synthesis engine (SE). CS - Control schema; DS - Data exchange schema

we describe our approach to handling the different CML models that may exist at runtime.

4.1 Overview of Model Realization

Figure 3(a) shows the process of realizing a CML model (schema) in the CVM. Each participant in the communication has a working CVM. UCI provides the environment for users to create new schemas or load schemas from a repository. SE accepts a schema from UCI or schema events from remote users via the UCM, handles the negotiation process, coordinates the delivery of media, and synthesizes control scripts. UCM is responsible for executing control scripts resulting in API calls to the NCB running on top of a communication frameworks such as Skype [6].

We limit the scope of this paper to the CML models that evolve and are maintained in the SE layer at runtime. Our approach to maintain and evolve the schemas at runtime in the SE involves three main processes: *schema analysis*, *(re)negotiation* and *media transfer* as shown in Figure 3(b). SE accepts a local UCI schema or a UCM event which contains a schema from the remote user, shown on the left side of Figure 3(b). The schema analysis process interprets the schema and generates events based on the runtime control schema (CS) or a data exchange schema (DS). There are two groups of events generated: (1) CS event - passed to (re)negotiation process and (2) DS event - passed to the media transfer process. These two processes work concurrently and both generate control scripts after processing their respective events. The dispatcher sends a control script to the UCM for execution or an updated schema to the UCI to be displayed to the user.

During the execution of a communication schema in the SE there may be several CML control models being manipulated at the same time. These CML models include: (1) the *executing schema* (may have several active connections) which supports the media transfer process to provide a communication service,

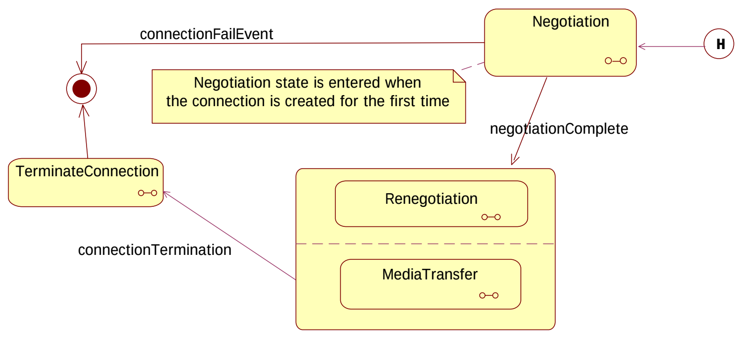


Fig. 4. State machine for a ConnectionProcessor.

(2) an *intended schema* that represents a user's request to change the executing schema (per connection), and (3) a *negotiating schema* providing a transition from the executing schema to the intended schema.

While it is possible to have multiple connections in a CML model, we discuss the runtime CML model in SE based on a single connection because each connection operates independently of each other. Figure 4 shows the high-level state machine for the *ConnectionProcessor* that represents the behavior of a connection. The state machine in Figure 4 consists of four submachines (*Negotiation*, *Renegotiation*, *MediaTransfer* and *Terminate Connection*). The submachines *Negotiation*, *Renegotiation*, and *MediaTransfer* represent the behavior of the processes with similar names in Figure 3.

4.2 Schema Analysis

The schema analysis process compares a received CS/DS for a connection with the locally held CS/DS copy and produces specific events based on the results of the comparison. These events may trigger a transition into the negotiation/renegotiation process, media transfer process, or both (see Figure 3). Figure 5 provides a simplified algorithm of analyzing the CS to illustrate the idea of generating CS events. The algorithm takes the received schema and current schema as input. Based on the source of the received schema (either from UCI or UCM), the role of the local user (whether or not the initiator) and the current schema, it would generate different CS events (line 4, 12, 18, 24 in Figure 5). We have a detailed version of the algorithm that will be presented in a future publication.

4.3 Negotiation/Renegotiation

The *ConnectionProcessor* accesses the *SchemaAnalysis* subprocess to generate CS events and DS events. CS events always affect the negotiation of a new CS or the renegotiation based on an executing CS. DS events carry media transfer request supported by an executing CS. CS events include `initiateNegotiationEvent`, `receivedInvitationEvent`, `sameControlSchemaEvent`, `changeControlSchemaEvent`, and `terminateConnectionEvent`. These events trigger actions for

```
analyzeSchema_Control (receivedSchema, currentSchema)
/*Input: receivedSchema - new schema from the UCI or UCM
         currentSchema - reference to schema in the (Re)Negotiation process
if receivedSchema is from UCI then
    currentSchema ← receivedSchema
    generate initiateNegotiationEvent /*handled by Negotiation/Renegotiation */
else if receivedSchema is from UCM and currentUser.isInitiator then
    store receivedSchema in an internal recipient list
    if all schemas from remote participants are received then
        if  mergeSchemas(all schemas) = currentSchema then
            generate sameControlSchemaEvent /* the end of negotiation */
        else
            currentSchema ← mergeSchemas(all schemas)
            generate changeControlSchemaEvent /*another round of negotiation */
        end if
    end if
else if receivedSchema is from UCM and !currentUser.isInitiator then
    if currentSchema is null then
        update currentSchema to include local capabilities
        generate receivedInvitationEvent /*display the invitation */
    else
        if mergeSchemas(receivedSchema, currentSchema) = currentSchema then
            generate sameControlSchemaEvent /* the end of negotiation */
        else
            currentSchema ← mergeSchemas(receivedSchema, currentSchema)
            generate changeControlSchemaEvent /*the reply to a negotiation */
        end if
    end if
end if
```

Fig. 5. Algorithm to analyze control schema during negotiation.

creating control scripts and state transitions to handle a non-blocking three-phase commit protocol for schema negotiation [7]. Similarly, DS events trigger the transition of *MediaTransfer* submachine. Using part of the *negotiation* submachine as an example, the `initiateNegotiationEvent` will trigger an action `sendSchemaRequest`, which generates control scripts for sending an invitation, and move the submachine to the *waitingResponse* state. Only the receipt of the `sameControlSchemaEvent`, which indicates all invitees accept the invitation, can trigger the action `sendConfirmation` for creating control scripts to send confirmation of the negotiation and move the *negotiation* submachine from *waitingResponse* to the *negotiationComplete* state.

4.4 Applying Runtime Model to Scenario

Three different intended CML models (schemas) are processed in the motivating scenario (see Section 3). This subsection describes how SE maintains and evolves the executing schema into an intended schema at runtime. Figure 6 shows the SE environment. The intended schema (shown at the top of the figure) is sent by UCI to SE for processing at runtime. This schema reflects that Dr. Burke (`A`) wants to invite Dr. Sanchez (`C`) to join the discussions with Dr. Monteiro (`B`). The executing schema is shown in ellipse labeled *SE Global Schema*, in which Dr. Burke has already established two independent connections, one with Ms. Smith (`D`)(handled by *ConnectionProcessor* C1 shown at the bottom of Figure 6) and the other with Dr. Monteiro (handled by *ConnectionProcessor* C2). Each

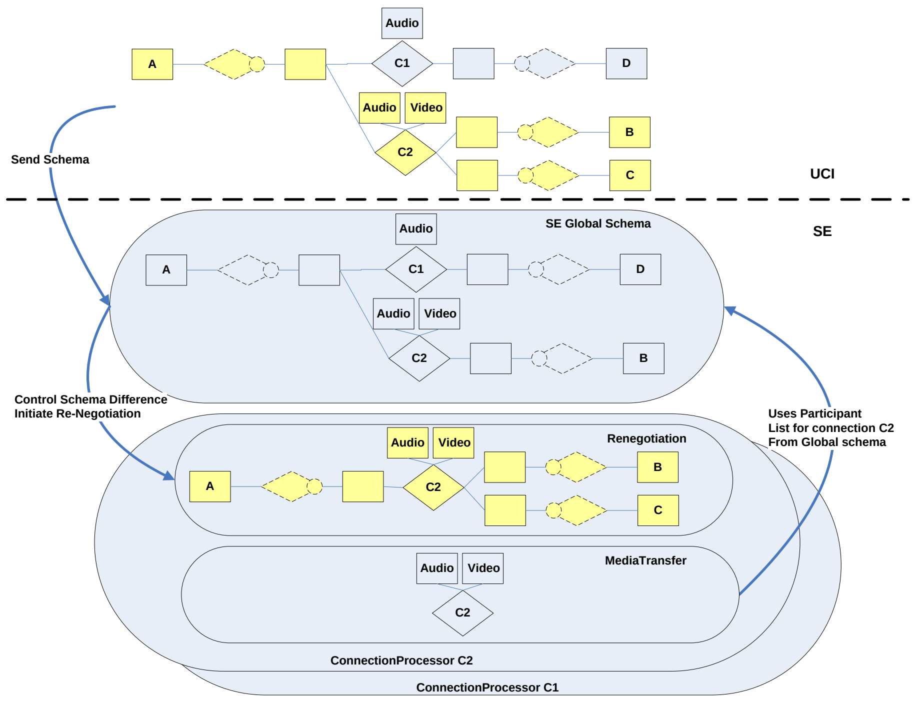


Fig. 6. CML model (control schema) being updated at runtime.

ConnectionProcessor contains two concurrent processes, one for renegotiation and the other for media transfer.

When SE accepts the intended schema from the UCI, it is decomposed into connection schemas and dispatched to the appropriate *ConnectionProcessor*. *ConnectionProcessor* C1 finds no in CS or DS, which means no change in the connection between Dr. Burke and Ms. Smith. For *Connection Processor* C2, it accesses the schema analysis process to compare the new intended CS (UCI schema in Figure 6) with the executing one (the SE Global Schema). The schema analysis process finds differences between these two CSs, which means the currently executing CS needs to be changed. It then generates `initiateNegotiationEvent` to trigger the *Renegotiation* process to initiate schema evolution. The negotiating schema is held in *Renegotiation* process. While the renegotiation is occurring, the *MediaTransfer* process in *ConnectionProcessor* C2 is still supporting the audio and video connection between Drs. Burke and Monteiro using the executing schema. When the renegotiation is complete, a confirmation from Dr. Sanchez's CVM triggers both processes in *ConnectionProcessor* C2. The *Renegotiation* process moves to the *Idle* state and waits for the next request to evolve the CS. The executing schema in ellipse labeled *SE Global Schema* is replaced by the negotiated schema.

4.5 Discussion

Our approach for evolving models is similar to the typical control loop mechanism found in control theory [8]. Schema analysis plays the role of the observer and the *CommunicationProcessor* acts as the controller. Using CML for specifying user-centric communication services, the types of changes that could occur during runtime is predictable and enumerable resulting in a more stable system. Evolution of a currently executing schema into a new schema includes negotiating the schema and switching to the new negotiated schema with minimum effect on the existing services. There are however several remaining questions to be addressed: (1) If the schema evolution is unsuccessful due to an exception, how can the SE rollback to the previous schema? (2) How to keep track of the evolution history of the runtime models? and (3) How to effectively maintain consistency between the executing SE schema and the execution state of the lower layers in the CVM? These open questions would motivate future research in this area.

5 Related Work

Addressing the maintenance and evolution of runtime models in a constantly changing and interactive environment is a major research problem in the area of MDE. Depending on the problem at hand, models might need to evolve to be synchronous with the runtime application through dynamic adaptation, or the runtime system needs to be adapted as the input model evolves. We will see how these problems are addressed in the community.

In Prawee et al [9], the authors developed a framework for co-evolution of system models and runtime applications. As a system is described in the forms of ADLs models and then projected toward an implementation platform, dynamic system adaptation can cause the running system to be out-of-synchronous with its model. The proposed framework enables a system/model evolution and provides architects with consistent views of running systems and their models. We use a different methodology to adapt our CVM at runtime. New communication requirements are represented by an intended CML model and result in a model evolution which leads to the runtime environment adaptation.

Van der Aalst [10] use a generic workflow process model to handle dynamic change of executing processes. Since the change of an executing control flow is a more complicated process, whereby new tasks could be added, old ones replaced, and the order of tasks changed, the number of types of model changes that could occur during runtime becomes significant. How to keep track of different variants of the processes and decide on the safe states for migration is challenging. The paper proposed a generic process model with a minimal representative for each process family to give a handle to deal with these problems. We address similar problems in that we need to manage various CML models during runtime and perform a safe migration of an executing CML model into a new one. However since we are only limited to the communication domain, the types of possible model changes are fewer and ways of effecting the change could be more dedicated then the general workflow model update.

6 Conclusions and Future Work

In this paper we provided an approach that shows how the Synthesis Engine (SE), a layer in the Communication Virtual Machine (CVM), maintains and evolves runtime models during the realization of user-centric communication services. Three processes were presented that support these activities including: schema analysis, (re)negotiation and media transfer. In addition, a scenario from the healthcare domain was used to show how these processes can be applied during the execution of a communication service. Our future work involves investigating techniques for handling schema rollback, maintaining a schema history and ensuring the consistency of runtime models in the different layers of CVM.

Acknowledgments

This work was supported in part by the National Science Foundation under grant HRD-0317692. The authors thank the reviewers for their insightful comments.

References

1. Burke, R.P., White, J.A.: Internet rounds: A congenital heart surgeon's web log. Seminars in Thoracic and Cardiovascular Surgery **16**(3) (2004) 283–292
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. Deng, Y., Sadjadi, S.M., Clarke, P.J., Hristidis, V., Rangaswami, R., Wang, Y.: CVM - a communication virtual machine. Journal of Systems and Software (2008) (in press).
5. Clarke, P.J., Hristidis, V., Wang, Y., Prabakar, N., Deng, Y.: A declarative approach for specifying user-centric communication. In: Proceeding of CTS 2006, IEEE (May 2006) 89 – 98
6. Skype Limited: Skype developer zone (Feb. 2007) https://developer.skype.com/.
7. Rangaswami, R., Sadjadi, S.M., Prabakar, N., Deng, Y.: Automatic generation of user-centric multimedia communication services. In: Proceedings of IPCCC. (April 2007) 324–331
8. Muller, P.A., Barais, O.: Control-theory and models at runtime. In: Proceeding of 2nd International Workshop on Models@run.time. (Sept 2007)
9. Sriplakich, P., Waignier, G., Meur, A.F.L.: Enabling dynamic co-evolution of models and runtime applications. In: Proceedings of COMPSAC, Los Alamitos, CA, USA, IEEE Computer Society (2008) 1116–1121
10. der Aalst, W.M.P.V.: Generic workflow models: How to handle dynamic change and capture management information? In: COOPIS '99: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems, Washington, DC, USA, IEEE Computer Society (1999) 115