Model-based Design and Generation of Telecom Services

A. Hartman¹, M. Keren², S. Kremer-Davidson², D. Pikus²,
¹IBM India Research Lab, Bangalore, India
²IBM Haifa Research Lab, Haifa, Israel
ahartman@in.ibm.com, {keren|shiri|dpikus}@il.ibm.com

Abstract

This paper describes a model based approach to the creation of services supported by the telecommunications infrastructure (Next Generation Networks or NGN), including landlines, mobile phones, the internet, and other devices. We describe a domain specific modeling language which enables designers to create such services without any knowledge of the underlying protocols. We also describe the tooling created to support the language – which enables the automatic generation of running services without additional code being written.

1. Introduction

With the commoditization of telephony services, telecom service providers are challenged to find new sources of revenue by generating new services and deploying them rapidly. A major stumbling block in the creation of new services is the need for specialized engineering professionals with intimate knowledge of telephony protocols and infrastructure, internet protocols, billing systems, and other technologies used to create services in the communications space.

This paper describes a model based approach to the creation of services supported by the telecommunications infrastructure which enables service architects to design such services without any knowledge of the underlying protocols. The approach is supported by tooling which facilitates the reuse of existing services, the use of templates, behavioral patterns, and pre-defined service elements to speed up the design and implementation process. The main contributions of this paper are the creation of a domain specific language (DSL) for telecom service design, together with tooling to support the language, and transformations which can transform a service design into a service implementation (including both structure and behavior) which can be deployed rapidly.

The paper begins with a survey of model based approaches to service design (Section 2), and then in Section 3 we describe a typical 3G telecom service which we will use as a running example to illustrate our contributions. Section 4 describes the telecom services definition language, and Section 5 describes the tooling we created to support service designers who use the language. In Section 6 we discuss the technical issues with implementation of the transformations, and we conclude with a section on future areas for research.

2. Model Based Development of Telecom Services

The goal of model-based development is to enable the development of applications that can be rapidly created and easily modified [GN04]. Although it is well-known that modeling can be an effective way to manage the complexity of service-oriented software development and although its widely used [PSS05], its adoption in the Telecommunications domain is not yet widespread.

When developing Telecom services one is required to have deep low-level knowledge of numerous protocols (e.g. SIP [SIP02], SDP [SDP98], Diameter [DBP03], HTTP [HTTP99], SOAP [SOAP07]) and then know how to synchronize between them. Several flavors of these protocols are in use, and several protocol extensions are defined for different signaling aspects. For example, in the IMS environment the SIP protocol is extended by IMS-specific SIP header fields and Diameter is extended by the use of IMS-specific Diameter applications (ex., Sh, Ro and Rf at application server side).

In IP telephony, several platforms and tools for service development are available which simplify the service development process to some extent. IBM’s WebSphere Application Server(WAS) [WAS], version 6.1 with IBM WebSphere IMS Connector [IMSCON], Oracle BEA WebLogic SIP Server [BEAWL], and Avaya SIP Application Server [AVAYA] are examples of such platforms. However providing basic support for telephony signaling and accounting within a telecom service requires the designer to have an understanding of SIP messages structure and SIP control flows (as described in RFC 3261 [SIP02] and other SIP related RFCs), Diameter base protocol (RFC 3588 [DBP03]) messages structure, and also the structure of messages for the relevant Diameter
applications (ex., 3GPP TS 29.328 [3GPP29328], TS 29.329 [3GPP29329], TS 32.299 [3GPP32299] and others). Moreover, developers also need to understand the interaction between the Diameter and SIP protocols.

Defining a common and understandable high level abstraction over all layers and protocols is a challenging task but once defined it will clearly simplify the process of Telecom services development.

The rest of this section discusses a number of papers and offerings related to model based development applied to services in general and specifically in the Telecom domain. All touch on some aspect that is relevant to our work, but there is no directly relevant prior art in the area of domain specific modeling and generation of telecom services.

IBM's UML Profile for Software Services [PSS05] has been defined for Service-Oriented Architecture (SOA), and implemented in IBM Rational Software Architect [RSA]. The profile's aim is to cover a number of activities for service-oriented solutions through the development lifecycle and also to provide views to different stakeholders. This profile is used in the extended Rational Unified Process (RUP) for Service-Oriented Modeling and Architecture method (RUP for SOMA) that merges these two activities together. It provides general SOA support but does not focus on domain specific issues. Our work uses some of the concepts introduced here but our DSL main focus is on enabling to create telecommunication services without deep knowledge of specific telephony protocols details.

For Telecommunication service development, IBM provides a package of software capabilities named IBM Rational Unified Service Creation Environment (USCE)[USCE]. It includes Conallen’s SIP modeling toolkit [CNO08, CNL07]. This toolkit includes a set of domain extensions to the Rational modeling platform for the development of Session Initiation Protocol (SIP) services. Its Domain Specific Language (DSL) defines basic SIP Services constructs and enables the description of the interaction flow of client-service messages using UML sequence diagrams. This environment simplifies development of telecom services but developers still need to be familiar with protocol details and remain responsible for implementing the behavioral parts of the service.

Georgalas et. al. [GA05, AG07] have applied Model Driven Architecture (MDA) in several case-studies that demonstrate the advantages it offers in the process of designing, developing and integrating Operational Support Systems (OSS) in terms of improved quality and lower costs. In the later paper, they describe an Eclipse-based environment of bundled Model Driven Development (MDD) facilities that supports the automatic generation of domain-specific tools.

Harel’s Live Sequence Charts approach and the PlayEngine tool have also been used to model Telecommunication applications [CHK03].

Belaunde et. al. [MB05, MB08] define a DSL for voice interactions. The language is a dialect of UML and describes voice dialogs by state machines. In the later paper the initial approach is extended for composite service modeling. Our work takes a similar approach to theirs but tackles the more complex domain of NGN telecom protocols (e.g. SIP, Diameter) and extends the approach by introducing higher levels of abstraction and reusable service components.

Deubler et. al. [DMR05], define a system in terms of its services and introduce aspect-oriented modeling techniques for UML Sequence Diagrams and show their usefulness for asynchronous communications with web services and between system components.

Mittal et. al. [MCGM] have described a simple integration scheme for the orchestration of composite Telecom services from simpler components. But their work does not make use of modeling.

Important initiatives have been taken by Telecommunication Management Forum (TMF) [TMF] to create standards in the world of telecommunication that result in defining Business Process Framework (eTOM) and Information Framework that includes Shared Information/Data (SID) models. These standards provide the common abstracts and allow easier understanding between the different worlds of service providers and users. We have attempted to leverage this work in our domain-specific modeling language for Telecom services.

3. Example of a Telecom Service

Our work will be illustrated by the design and implementation of the “Meet Me Now” service (MMNS). The primary user of the MMNS has a cellular phone with a subset of his address book contacts designated as a buddy list. When the service is invoked, it sends an SMS message to all the members of a selected buddy list that are within a fixed distance from the user, and are available.

The implementation of this service assumes the existence of a buddy list service which specifies information on the users buddy list content, a location
service which specifies the geographical position of a cell phone user, a presence service which specifies whether a user is available or busy (similar to computer chat services), and an SMS Service capable of sending SMS messages.

4. A Domain Specific Language for Telecom Services

The Telecom Service Domain Specific Language (TS-DSL) is a language for defining models of telecommunications services which are independent of the specific architectures and protocols (e.g. IMS, SIP, Diameter, or SDP) which are used by the network. This language is intended for service designers who may not have telecom industry domain knowledge. It hides the internals of the platforms focusing on their provided functionality and provides high level building blocks for the design of services. Model transformations are responsible for closing the abstraction gap by transforming the service model into a deployable service. The TS-DSL includes both static and dynamic aspects of telecom services utilizing the power of UML2 and IBM’s “UML Profile for Software Services” [Apr05]- refining and extending them for telecom service domain.

In this Section we describe the elements of the TS-DSL and include some details of the process we expect designers to follow when specifying the behavioral aspects of the service.

The meta-model for telecom services (see Figure 1) is derived from the UML Profile for Software Services and extends the profile with telecom specific constructs for sending and receiving notifications from other services, and for handling telecom specific errors. The service operation includes a Boolean attribute isSynchronous, which enables the designer to ignore the problems of synchronization, leaving the synchronization work to the model transformation.

In SOA a Service Specification can have multiple realizations. Our TS-DSL is focused at providing means to design a realization of the Service Specification within the defined Telecom Service.

The behavior of a Telecom Service is specified by state machines and activity diagrams. Each instance of TelecomService is created with a main state machine which specifies the service interaction with its client. See Figure 2 for an example of the Meet Me Now state machine. The designer then specifies a “do” activity for each defined state. The purpose of this activity is to catch the event (and its data) that brought the service to this state and pass the control and data to the implemented behavior. For example, in Figure 2 the transition with the invokeMeetMeNow service invocation event passes both the group name and inviter information to the “do” activity as its inputs. Note that we utilize the UML2 semantics for state machines and activity diagrams, as defined in the UML 2 specification.

The transition between states in a TelecomService may be triggered by one of the following:

1. **A Service Invocation request arrived**: meaning that a call was made to one of the service’s provided interfaces ServiceOperations.

2. **A Notification Signal arrived**: meaning that an external service/client sent a notification to the service that some event occurred.
3. **A TelecomError Signal arrived**: meaning that some unexpected situation occurred.

4. The activity of the previous state ended and no specific trigger was specified on the transition.

In all cases, constraints (guards) on the transition need to be met in order for the transition to be taken.

To capture this in the profile we introduce the following Events (see Figure 3):

1. **ServiceRequest** (extends CallEvent): indicating that a request to activate a service provided operation arrived

2. **NotificationEvent** (extends SignalEvent): indicating that a Notification arrived


A state’s “do”, “entry”, and/or “exit” activities can be seen as the entry points to the service activity flow. The flow can invoke a variety of actions and control nodes connected via control flow links. Data flow is defined via data links between the action’s pins.

Actions can invoke instance creations, operation calls, external service invocations, other activities, etc. Designers are expected to define activities that: implement invoked operations, specify reception event targets, and other activities required to design their service behavior.

It is important to point out that when a ServiceRequest arrives to the state machine for processing the target state “do” activity is expected to include an AcceptServiceRequestAction to catch it and start the Activity flow from it. This new action inherits from UML’s AcceptCallAction and overwrites some of the semantics of its parent class. It specifies to the model that the client expects the specification operation to be invoked but as we differentiate the specification and realization it does not dictate what exact operation is to be invoked, and thus its return information is not used. This action includes output pins containing the data passed by the event and the designer is responsible to dictate what behavior will be invoked using control flow and other actions. Typically, the designer will design the activity to invoke the ServiceImplementation related operation.

To simplify the definition of the model’s activity flow we added and modified some UML actions – see Figure 4.

1. **FreeFormAction** (extends OpaqueAction) - allows specifying snippets of Java code within an Activity. The Action input and output pins are treated as variables. The advantage of this Action is that it allows designers to enter their own code when they rather not use UML Actions for this.

2. **DecisionAction** (extends OpaqueAction) – replaces UML’s decision node by allowing to specify input to the decision. The decision constraint can use the action input pins as variables. We will consider removing this Action when RSM’s support for decision node will mature.

3. **Next** (extends OpaqueAction)- this action is used together with ForEach action (see below) to provide the next target in the loop iteration process.

4. **ForEach** (extends StructureActivityNode)- represents a iterative loop. It is defined together with Next (defined above) and InputList (define below) (see Figure 5).

5. **InputList** (extends InputPin) - used as a part of ForEach action indicating that the ForEach Activity must have a single input of type List. The ForEach with iterate on the members of this list.

6. **CreateTelecomElement** (extends CreateObjectAction)- this action is used to indicate that a ‘TelecomModelLibrary’ element will be created.

7. **InvokeTelecomOperation** (extends CallOperationAction) – this action is used to indicate that a ‘TelecomModelLibrary’ entity Operation is called.

8. **InvokeTelecomActivity** (extends CallBehaviorAction) – this action is used to
indicate that a ‘TelecomModelLibrary’ Activity (behavioral pattern) is invoked.

Figure 4: Extensions of OpaqueAction node

Figure 5: Types related to loop modeling extensions

Some of these features of the domain specific language are illustrated in Figure 6.

Figure 6: An activity diagram using a variety of TS-DSL Actions

Telecom Service design typically requires invoking a variety of external services that may be implemented in diverse manners (e.g. SIP Services, Web Services) and accessed using different protocols (e.g. SIP, HTTP, SOAP). The Service Structure Diagram below (i.e. Component Structure Diagram) is used to define the relationships between a new service and the external services it invokes (see Figure 7).

In order to invoke an external service from an activity, we use a ServiceInvocationAction. When creating such an action, the designer selects an operation from the provided interface of an external service. This will update the input and output pins of the action to match the signature of the chosen operation.

Figure 7: Meet Me Now Service Structure Diagram

For example, Figure 8 shows an activity with two ServiceInvocationAction instances which specify the invocation of the findLocation() operation of the Location Service and extractBuddyList operation of the BuddyList Service. The input and output pins of the action reflect the operation signature, thus for example the findLocation() action accepts a Party and returns a Location.

Figure 8: Invoking external services

This is our implementation of service choreography using service invocation points within the service behavior, and all the complexity of protocols, communication, and other low level details are hidden from the designer and implemented by the transformations.

5. Telecom Service Creation Environment

This section describes the tooling we implemented to support the language defined in the previous section, and the library of telecom service specific entities with which we populated the tool.

The Telecom Service Creation Environment (TSCE) is a set of plug-ins and transformations built on top of the Rational Software Architect, which is itself built on top of the Eclipse tooling framework. As such the tooling is open and extensible with the ability to add a
further level of customization when deploying it in a commercial context.

The TSCE supports the domain specific language (TS-DSL) introduced in the previous section with a UML profile, and provides a level of automation for implementing the service design methodology we have adopted. The modeling palette includes elements of the domain specific language, and many actions performed by the designer are accompanied by the automatic creation of the templates and diagrams required for completion of the model (see Figure 9). For example, when the designer creates a new service, the basic structure and state machine for the service is created based on one of the model templates, with an initial configuration that can be modified.

We also included a telecom service model library and some telecom specific structures that can be reused without modification. The model library comprises three packages, datatypes, business, and communications.

The datatypes package allows designers to use predefined data types that are widely-used in Telecom service models, quickly and efficiently. It is derived from accepted industry standards, including Shared Information Data (SID) models that relate to TMF (TeleManagement Forum) working on eTOM and SID evolving standards. Some attribute fields are extended and specialized according to naming convention in the IMS area. Examples of the datatypes that are included in this package include party, person, phone number, and URI. The library includes several commonly used notifications and error messages, and other types associated with the presence services defined in the Parlay-X specification standard which is used in IBM’s TWSS Presence server interfaces.

The business package contains entities that can be used to provide accounting, authorization and user profile management capabilities to the service. The entities defined in this package include Customer, Account, CreditProfile, etc. Other entities focus on accounting procedures and enable both session-based and event-based billing.

The communications package contains the basic entities required to manage calls and other modes of communication in an intuitive object oriented manner that is flexible, simple to use, and powerful. We include a description of two of the main package entities Call and CommunicationThread (as seen in Figure 10).
A **Call** is a stateful communication instance between participants. Once participants are connected, a media stream is established between them. The communication can be achieved through many channels supporting for example audio, video or other channels.

**Communication Thread** is a class responsible for organizing the runtime aspect of a Call. It points to a set of participants and specifies how they are grouped. It holds information on the thread initiator which is the end that initiated the Call. It also has an indication if the entire thread is active or suspended. Each Call is created with a single CommunicationThread instance. For most call management schemas, designers will not need to explicitly define behavior that accesses or manipulates the CommunicationThread during the call lifecycle.

The CommunicationThread abstraction enables, for example, services which intermittently interrupt a call with some information and are not full time participants in the call (very important in message traffic management). A Call can be created with two threads, the first with the regular set of participants and the second including not only all participants but also the service itself as a participant. For most of the call lifecycle the first thread is active and the second is suspended. But when the service wants to interrupt, it suspends the first thread and activates the second.

The model library contains a set of predefined Activities that capture commonly used behavior patterns. These Activities can be invoked from the service flow (using `InvokeTelecomActivity` action) freeing the users from defining them from scratch. Among them are: creating and establishing a call, terminating a call, applying typical session based accounting, etc.

Other reusable structures provided with the tool include a media player and a unit convertor which can be seen in the bottom panel of Figure 9. The TSCE allows domain experts to introduce new reusable elements into the environment.

When creating a new service in the TSCE the designer may choose to do so using a template. Three service templates are provided for: a basic service, a service that requires call management (e.g. conference call), and a subscribe/notify service template

A set of wizard pages are used to configure the template instance. The new service is created with an initial structure including an initial state machine with states and transitions, an implementation class and package structure. The state machines of the three templates are shown in Figures 11-13.
6. Closing the Abstraction Gap - Transforming Designs into Running Services

One of the main challenges of this work was in defining the right level of abstraction that is both powerful enough to describe all the required functionality of a service but is also simple, intuitive, and allows designers not familiar with telephony protocol details (non telecom experts) to design and develop telecom services by hiding the low level protocol and architecture details. A further challenge is the generation of runnable service code, i.e. defining mapping rules and infrastructure that close the abstraction gap and transform the designed services into running code.

Numerous transformations exist that can generate the skeleton code of an application from a UML model. We chose to extend Rational Software Architect’s built in transformation from UML to Java and added our service specific rules to perform the task of generating both the static (e.g. SIP Servlet structure, sip.xml) and behavioral parts of the application.

The code behavior generation task includes the interpretation of UML behavioral semantics and closing its variation points in relation to telecom service behavior. We also tried to generate code that is efficient, readable, and easy to modify by the service developer.
We designed the UML behavior code generation part to be extensible and domain independent so it can be used for other domains. We implemented a core infrastructure that includes various abstract classes, event managing engines, and registries that enable correct and efficient behavior invocation. For example, we introduce an abstract Activity class from which each generated model Activity inherits. Transformation rules populate this class with:

1. A constructor that sets the activity input data and registers it as listener to events it expects to handle.
2. Methods for each action in its flow. Each method is responsible to extract the input data from its input pins, process the action related behavior, and set the resulting data on the output pins (for an example see figure 14).
3. Methods that implement the flow graph corresponding to the flow specified in the Activity.
4. Methods that initiate all data flow related entities and the dependencies between them.

When looking at the generated Activity code users see a structure resembling their designed Activity. This will simplify the understanding and enable more natural modifications if needed. An example can be seen in figure 14 that includes a MeetMeNow Activity and part of the generated Activity code that is responsible to invoke the CallBehavior Actions related Activities.

The design of the transformation enables the registration of new types of action generation rules that are responsible for the method behavior content. In our current implementation we added special telecom domain related support for invoking model library operations, invoking external services, sending notifications/errors, activating reusable elements, etc.

The transformation is built extensible to enable incorporating domain specific contributions to the generated code. We implemented a default domain extension for IMS (SIP oriented). Thus when generating code using this IMS domain, runtime code is packaged as sipplet (i.e. SIP servlet) that runs on SIP enabled Application Server (eg. IBM WAS, version 6.1).

The following is a detailed description of the implementation of the service behavior (generated using the IMS domain extension): All messages from the network arrive to the Sipplet. The Sipplet converts the message using registered Event Builders into a Service Request, Notification or Error depending on its content and context. The event is passed to an event dispatcher. All “living” behavioral units, (e.g. main state machine, live activities) register on their instantiation to the event dispatcher for particular events that they are interested in (i.e. can or expect to handle). Thus once a corresponding event arrives to the dispatcher it is passed for processing to an interested registered party. The actual algorithm
depends on the states of the currently running behaviors and the event type. Each behavioral unit that completes its lifecycle unregisters as an event listener from the dispatcher. When a first Service Request event arrives from a client it is passed to the only registered behavior at this time, the service main state machine. This causes a transition in the state machine from the start state to a new state. If the new state has a “Do” activity attached to it, a corresponding Activity instance is created and control is passed to it. Data is passed to the Activity and the Activity starting point is set to the corresponding AcceptServiceRequestAction. The action’s output pins are set with the data passed by the service request. From this point the activity can continue executing according to its specified flow. Typically now control will pass to a Service implementation operation for handling the service request. The Activity implementing this behavior is invoked. Note that a state may also have an “Entry” Activity that is to be invoked just before entering the state or an “Exit” Activity that is to be invoked just before leaving the state.

One of the main simplification decisions we chose for the designers was that the main service state-machine is to represent the interaction of the service with a single client and not multiple clients. This drastically reduces the complexity and increases the readability of the state-machine for the service designer. The problem is that at runtime the service needs to interact with multiple clients simultaneously. To overcome this we create a context object for each client interaction that includes all interaction state information. When an event arrives at the sipplet it is processed and dispatched to the correlated client interaction for processing. Each context runs in a separate task. To manage this and enable simultaneous client processing, we handle threading and synchronization in a scalable manner to ensure that the generated service does not consume too many resources and has reasonable performance. We do a careful analysis for task thread balancing.

The model library also posed some interesting implementation issues. Do we generate full behavior on each usage of a library element or is it possible to implement this as a standalone library and have the generated code invoke its functionality and get the same expected results. We decided to create a standalone library to hide the protocol specific parts from the generated code, simplifying service portability (e.g. ability to run the service on networks with different protocols) and simplifying the generated code structure.

This task was complicated by the numerous situations where method implementation cannot be independent of the runtime environment. Moreover some implementations require event based interaction with the network. For example, to establish a call, the library Call “establish()” operation is invoked. The implementation of this operation needs to manage numerous message exchanges between Parties until both are connected. To do this the operation requires interaction with the network to receive the target ends’ events. But network interaction is handled by the Sipplet and this operation should not interfere with the regular Sipplet client interaction. Moreover, the model library should not depend on a particular service implementation. To overcome this, each Call instance registers as a behavior entity to the core infrastructure event dispatcher for specific events; on arrival these events are passed straight to the Call instance and the service state is not modified.

This architecture allows service deployers the flexibility of deploying the service over networks which use different protocols for the required tasks, requiring only a different implementation of the model library jar, and minor or no code changes.

To prepare the service for deployment the transformation is required to generate a SIP deployment descriptor (sip.xml). This file contains the information necessary for deploying a SIP application module. To perform this task we need to identify which SIP message types can initiate the service under which constraints. This is done by traversing the state-machine, locating the set of Events that can be reached directly from the service start state or by following empty transitions. Then for each Event we extract its mapping information and query the model library for which SIP message types it maps to. This information and the constraints on the transitions pointing to them, are used to generate the sip.xml file.

7. Conclusions and Future Directions
This paper describes a model based approach for the design and creation of telecom services hiding the complexity and details of the underlying protocols and architectures from designers. Our approach radically simplifies service design, cuts down service time-to-market, makes the service design process accessible to designers not familiar with telephony protocol details, and simplifies maintenance.

We introduced the Telecom Service Domain Specific Language and the Telecom Service Creation Environment that exposes it and is crafted to fit the telecom service designer’s requirements. We illustrated the Language and Environment by
designing and implementing the “Meet Me Now” service.

In this process we were faced with many interesting research challenges. Most are related to defining the right abstractions (and mapping rules) that hide the low level protocol and architecture details from the designers but are powerful enough to provide a rich set of service functionalities in an intuitive manner. Others relate to defining the behavior semantics, closing some UML variation points in relation to telecom services and generating high quality behavior code and not only skeleton code.

In the future we plan to focus our research on other aspects of model based development of telecom services. This includes research on how to perform model based testing and model debugging from TS-DSL detail level. Additionally, we will look into providing round trip support, i.e. reflecting modifications to the generated code back to the design model.

8. References


[SOAP07] SOAP Specifications, http://www.w3.org/TR/soap/


[3GPP32299] 3GPP TS 32.299, Telecommunication management; Charging management; Diameter charging applications., http://www.3gpp.org/FTP/Specs/html-info/32299.htm