H-0278 (H1002-004) February 6, 2010
Computer Science

IBM Research Report

Model-based Design and Generation of Telecom Services

A. Hartman
IBM Research
Bangalore, India

M. Keren, S. Kremer-Davidson, D. Pikus
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Model-based Design and Generation of Telecom Services

A. Hartman[1], M. Keren[2], S. Kremer-Davidson[2], D. Pikus[2],

[1]IBM Research, Bangalore, India
[2]IBM Research, Haifa, Israel
ahartman@in.ibm.com,
{keren|shirild|pikus}@il.ibm.com

Abstract. We describe 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 introduce a protocol-independent, domain-specific modeling language containing high-level entities that architects and developers can use in their service design to rapidly support complex service flows in their telecom services. We also describe the transformation and tooling created to support the language, which enables the automatic generation of running services (structure and behavior) without additional code being written. Examples of two typical telecom services are provided to illustrate how the modeling language and tooling are used for telecom service development. These examples show that the presented approach can simplify and speed up the development of innovative telecom services.

Keywords: Model-based development, domain specific language (DSL), Telecommunication services, service design, model transformation

1. Introduction

The Telecom industry has grown and changed rapidly during the last few years. New infrastructures and protocols are frequently introduced and telecommunication companies are required to adapt rapidly to change. With the increasing demand for more innovative services, the challenge of developing telecom services continues to grow. Moreover, telecom services typically require invoking a variety of external services that may be implemented in diverse ways (e.g., SIP services, web services) and accessed using different protocols (e.g., SIP[1], HTTP[2], SOAP [3]).

Today, only protocol experts are able to develop telecom services. In this industry, most developers do not use abstraction mechanisms in their service design. Thus, the services tend to be closely coupled to the protocol layer, making the service design protocol-dependent. The lack of separation between service logic and technical interfaces reduces the service design flexibility and slows development.

To overcome such problems, other industries (e.g., automotive, aerospace) have adopted both model-driven architecture and service-oriented architecture. This is because modeling is an effective way to manage the complexity of service-oriented software development and it enables the development of applications that can be rapidly created and easily modified [4].

Model-driven architecture (MDA) formalizes the evolution of model-driven development (MDD) by defining models at distinct levels of abstraction and defining transformations that map and manage the relationships between those models and various implementation technologies. MDD has many advantages, including: increased productivity, simplified maintainability, better consistency, improved communication, and the encapsulation and reuse of domain specific expertise. Our intention is to bring these advantages to telecom service development.

Some work has been done in the telecom industry to adopt model-driven architecture and service-oriented architecture [5, 6, 7], but is not widespread yet. The level of abstraction has not been raised sufficiently to disconnect from the protocols and allow non-telecom experts to develop quality telecom services rapidly.

For service development several platforms and tools are available that simplify the service development process to some extent but they still require deep knowledge of the protocols, technologies, and methods to synchronize between them all.

In this paper we describe a model-based approach to the creation of services supported by the telecommunications infrastructure. This approach allows non-telecom experts to develop telecom services. We expect the users of this tool to be familiar with UML[14] and the language constructs we define. They are not required to be familiar with telecom protocols, platforms or service development.

To achieve this, we defined a telecom service domain-specific language (TS-DSL), developed a transformation that generates runnable code, implemented the Telecom Service Creation environment tool and designed and developed several representative sample telecom services to validate our approach. To enable the complete definition of telecom service behavior, we have defined an action semantics over a subset of UML behavior entities along with our TS-DSL-introduced behavior entities.

TS-DSL also captures a set of concepts that are commonly used in telecom service development but are being reinvented time and time again in various proprietary formats. We define these entities and others in the Telecom Library. The library also contains a set of built-in reusable activities for commonly used behaviors. Details on TS-DSL and the telecom library are provided in Section 2.

To enable developers to transform their service models into runnable deployable code we defined a transformation that generates the code and other artifacts from the service design. The generated service does not require any modification to run. Details are provided in Section 3.

To ensure our language is applicable and simple to use, we implemented the telecom service creation environment tool (TSCE) on top of IBM Rational Software Architect (RSA)[15], which allowed designers to design telecom services using the TS-DSL and the Telecom Library and generate code from it. Details are provided in Section 4. Using TSCE, we designed and deployed several typical telecom services (chosen in consultation with telecom service domain experts). This provided initial indication that indeed the TS-DSL, telecom library and transformation greatly

improve the user experience and lower the expertise level required for developing telecom services. Details on these typical telecom services are provided in Section 5.

Section 6 summarizes the related work. Section 7 concludes and discussed on future directions.

2. Telecom Service Domain-Specific Language

When designing a service, the designer should focus more on its business logic and less on its relation to the low-level details of the specific telecom platforms or protocols. Model-based development enables rapid development and maintenance of applications [4]. Thus a suitable domain specific language for telecom services enables defining an abstract model built from telecom domain specific entities that can be transformed into real application code via transformations.

TS-DSL is a language for defining platform- and protocol-independent models of telecom services. The target audiences of this language are both telecom experts and those with less expertise who need to develop quality telecom services efficiently. These services typically require support for call management, instance messaging and interaction with external services (e.g., presence service, billing service).

The main challenges of this work were related to defining the right level of abstraction that is flexible enough for defining a wide range of applications in this domain. The abstraction should be powerful enough to describe all the required functionality of a telecom service. It should also be both simple and intuitive to allow non-experts to design and develop telecom services by hiding the low-level platform and protocol details.

TS-DSL supports design of both structural and behavioral aspects of the service. To formalize the behavior definition, it introduces an action semantics over a subset of UML behavior entities along with our TS-DSL behavior entities.

2.1 Telecom Service Structure

Service-oriented architecture (SOA) [16] is a commonly used architecture that separates functions into distinct services that interact typically over a network in order to allow users to combine and reuse them in the production of applications. Each service includes a specification (via interfaces) that defines how it expects other services to interact with it. In principle, each service can have multiple realizations. Actual service-to-service communication also involves data passing. The TS-DSL design was influenced by SOA concepts.

In TS-DSL, *TelecomService* is the top level entity representing the designed telecom service (see Fig. 1 showing part of the TS-DSL telecom service UML profile). Each *TelecomService* exposes a set of *TelecomServiceSpecifications* that define its interfaces. As TS-DSL focuses especially on defining a realization of the *TelecomService*, each *TelecomService* can have a number of realizations. We expect each model to contain a realization, named *ServiceImplementor* (extends UML's class) that contains all information needed for implementing the interfaces.

As in SOA, each *TelecomServiceSpecification* exposes the interface through which the service expects others to interact with it. *TelecomServiceSpecifications* expose a set of *ServiceOperations* (extending UML's operation). *ServiceOperation* includes a Boolean attribute called "isSynchronous" that specifies the synchronization required for the invocation process. Actual synchronization-related tasks are left to the model transformation. Although this characteristic is not typical in SOA, it is important in the telecom domain because in some cases, delicate protocol interchange-related functionality defined in the specification must be completed without interruption (e.g., in billing).

Each *TelecomServiceSpecification* also defines the set of notifications and errors it can send/receive throughout its lifecycle. To be able to precisely identify the semantics of the signal we introduce Notification and Errors both extend UML Signal.

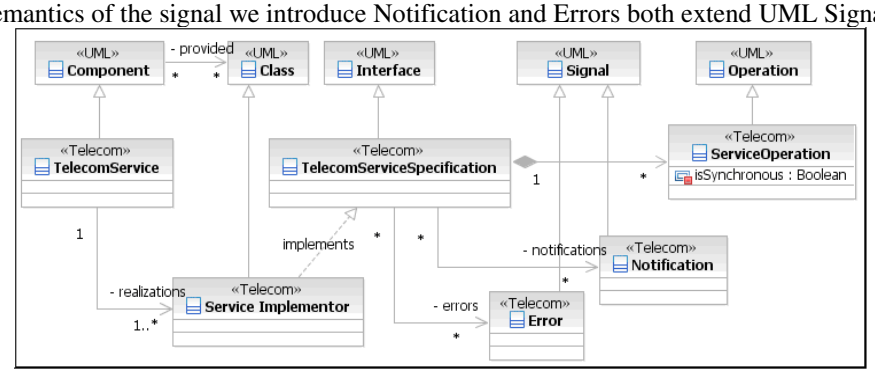

Fig. 1. Telecom service structure

2.2 Modeling Telecom Service Behavior

To define telecom service behavior we utilize the UML2 semantics for state machines and activities.

Each *TelecomService* is created with a main state machine. The state machine defines how the service state changes in response to events it receives or exceptions that are thrown during the service operation.

To reduce the complexity of defining and managing such a state machine, the *telecom service* state machine captures the service interaction with a single party. Thus it is not contaminated by events arriving from other parties that require handling in parallel to the main service flow.

A *TelecomService* state machine includes a set of states and transitions. Each state in the state machine can contain "do", "entry" and "exit" activities. An "entry" activity is launched on entrance to the state. An "exit" activity is launched just before leaving the state (due to the arrival of a signal that causes the state machine to move to another state). A "do" activity is launched to perform the actual logic in the state.

If a state includes one of these activity instances, it is responsible to launch it at the expected time and pass it to the event (and its data) that caused this state to be

processed. Note that if *Notifications* and/or *Errors* events are thrown from the activities and not caught by them, they will reach the state machine and may also cause state transition.

A transition between states in a *TelecomService* may be triggered by an event caused by either: a service invocation request (a request made to invoke one of the service's provided interfaces, i.e., a *ServiceOperation*) or the arrival of a *Notification* or *TelecomError* event. An empty transition between states is defined to allow moving from one state to the other with no specific event causing the trigger. Flow passes through this transition only after a source state behavior finishes. In all cases, constraints (guards) on the transition must be met for the transition to take place.

To capture this in the profile, we introduce the following events (see Fig. 2):

- ***ServiceRequest*** (extends UML CallEvent)**:** A request to activate a service-provided operation arrived
- ***NotitificationEvent*** (extends UML SignalEvent): A notification arrived
- ***TelecomErrorEvent*** (extends UML SignalEvent): A TelecomError arrived
- ***TelecomSignalEvent***(extends UML SignalEvent): A telecom-related protocol signal arrived (for advance designers)

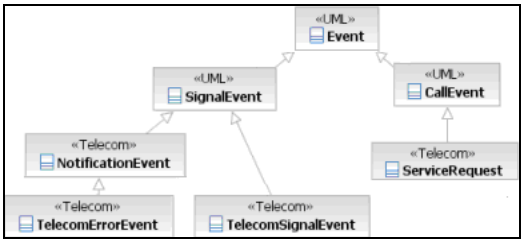

Fig. 2. Event types

We identified three main types of telecom services: a basic stateless service (i.e., no client information is stored for future interactions); a service that keeps the full state of the required interaction (e.g., call management); and a bidirectional service that keeps some data from the user and is able to interact with him when needed (e.g., subscribe-notify management).

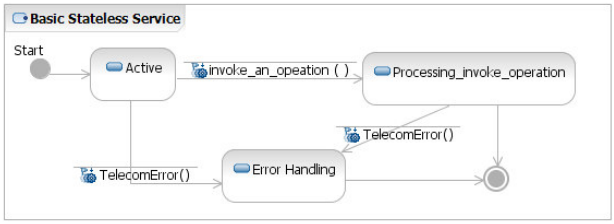

Fig. 3. Basic stateless service state machine

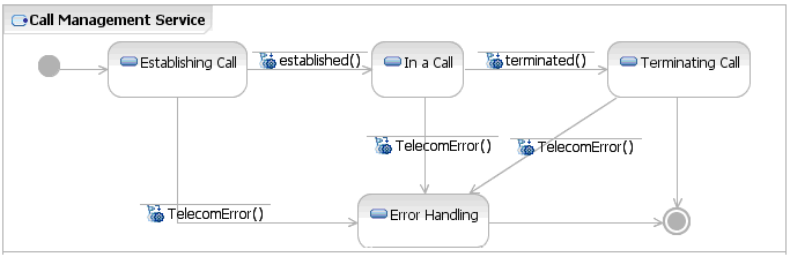

Fig. 4. Call management service state machine

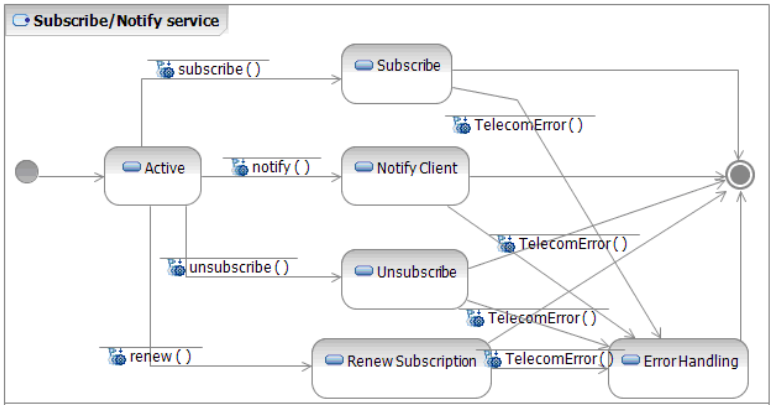

Fig. 5. Subscribe/Notify service state machine

TS-DSL enables the design of these service families and others. Figures 3-5 capture the state machines of these types of services. Note that transitions marked with "TelecomError" mean that an error event arrived. All other marked transitions represent service invocations.

We now focus on the UML activities used to define service behavior for each state. A state's "do", "entry", and "exit" activities are the TS-DSL's entry points to the activity flow.

Each UML activity includes actions and control nodes connected via control flow links. Data flow is defined via data links between the action's pins. The TS-DSL introduces a set of telecom-related actions to answer telecom domain-specific needs. Their intent is to simplify flow design for telecom service designers. The TS-DSL actions are defined below.

- ***AcceptServiceRequestAction*** (extends UML AcceptCallAction): Used as an initial flow node in a state's activities with an incoming transition caused by a *ServiceRequest* event. The action output pins pass data as parameters in the event. This action specifies that the client expects a service operation to be invoked. Since we differentiate between the specification and realization, the action does not dictate precisely what behavior is to be invoked, and therefore its return information is not used. Nevertheless, a designer will usually design the activity to invoke the related *ServiceImplementor* class operation.

- ***AcceptNotificationAction*** (extends UML AcceptEventAction): Similar to AcceptServiceRequestAction. This action is used as an initial flow node in a state's activities with an incoming transition caused by a *NotificationEvent*. The action output pins pass the *Notification* instance data. This action is relevant to *the TelecomErrorEvent* that inherits from it.
- ***AcceptTelecomSignalAction*** (extends UML AcceptEventAction): Similar to AcceptServiceRequestAction, This action is used as an initial flow node in activities invoked after a telecom signal arrives.
- ***ServiceInvocationAction*** (extends UML CallOperationAction): Used to invoke an external service from within the activity flow. Telecom service design typically requires invoking a variety of external services that may be implemented in diverse ways (e.g., SIP services, web services) and accessed using different protocols (e.g., SIP, HTTP, SOAP). When creating such an action, the designer selects an operation from the provided interface of an external service.
- ***InvokeTelecomOperationAction*** (extends UML CallOperationAction): Used to invoke operations of telecom model library entities (to be further discussed in Section 2.3) from the activity flow. This action indicates that a 'TelecomModelLibrary' operation is called.
- ***CreateTelecomElementAction*** (extends UML CreateObjectAction): Used to indicate that a 'TelecomModelLibrary' entity is to be created.
- ***FreeFormAction*** (extends UML *OpaqueAction)*: Allows specifying snippets of Java code within an activity. The action's input and output pins are treated as variables used in this code. Although not telecom domain-specific, we found this action very useful in situations where basic general purpose computation (code snippets) is needed. This action reduces flow clutter, as it replaces chains of atomic UML actions to perform simple computations.

Other elements of activity were modified or extended inside the TS-DSL (e.g., ForEach, Next, InputList, DecisionAction) to allow the efficient manipulation of activity flows and their transformation into runnable code.

The TS-DSL entities in this section are intended to be used by both telecom experts and non-experts; they do not rely on any knowledge of low-level technical telecom details. Parts of these entities guide service designers in their use of Telecom Model Library entities (see section 2.3) for additional telecom domain-specific operations.

For telecom experts who want low level control over the telecom protocol-related events, we define the *TelecomSignalReception* (extends UML reception). In UML, receptions can be attached to a UML class indicating what behavior (e.g., an activity) to invoke if a particular signal is received (asynchronously). In TS-DSL, we allow designers to add TelecomSignalReceptions and specify what activity is to be invoked if a TelecomSignal arrives. Moreover, they can indicate if they would like the activity to be invoked either after the signal arrived or before it is sent. The latter is not supported in UML.

Telecom services logic usually requires collaboration with other services (maybe even from different providers) to make use of their functionality. But invoking

external services from telecom services is not a trivial task. This is mainly because external services may be implemented in diverse ways (e.g., SIP services, web services) and accessed using different protocols (e.g., SIP, HTTP, SOAP).
To hide these details in the telecom service design, we treat each external service as a component with a provided interface. The designer can use the *ServiceInvocationAction* (described above) to indicate what operation from its provided interface it wants to invoke.

2.3 Telecom Services Library

The Telecom Service Library (TSL) includes a set of entities that are intended to be used without modification, independent of the implementation platform and protocols.

TSL entities include: abstractions of concepts that are commonly used in telecom service development and often reinvented; an object-oriented abstraction over telecom communication flows; and a set of reusable activities capturing commonly used behaviors.

The TSL is divided into three parts: data types, business entities, and communication entities.

Data types allow designers to use predefined, widely used data types. The content is derived from industry standards, including shared information data (SID) models related to the TeleManagement Forum (TMF) [22] working on the eTOM and SID evolving standards. These standards provide common abstractions and terminology that allow easier understanding between the different worlds of service providers and users. In TSL, we extended some SID elements with additional attribute fields specialized according to naming conventions in the IMS area. Examples include Party, Person, Phone number, and URI. The data part also includes several commonly used *Notifications* (e.g. CommunicationEstablishedNotification), and *Errors* (e.g. AccessDeniedError).

The business entities can be used to provide accounting, authorization and user profile management capabilities to the service. For example: Customer, Account, and CreditProfile. Other business entities focus on accounting procedures and enable both session-based and event-based billing.

The communications entities are used to manage calls and other modes of communication in an intuitive, object-oriented manner that is flexible, simple to use and efficient. One of the main entities, *CommunicationThread* is described below.

CommunicationThread (see Fig. 6) is a class responsible for organizing a runtime aspect of a *Call*. Each *Call* contains 1 or more *CommunicationThread*s. Each *CommunicationThread* points to a set of participants and specifies how they are grouped. It also holds information on the thread initiator, and indicates if the thread is active or suspended (more then one thread can be active in a time). The *CommunicationThread* abstraction enables a variety of options such as services that intermittently interrupt a call with some information, yet are not full-time participants in the call.

The TSL also contains a set of predefined activities that capture commonly used behavior patterns. These activities include creating and establishing a call, terminating a call and applying standard session-based accounting

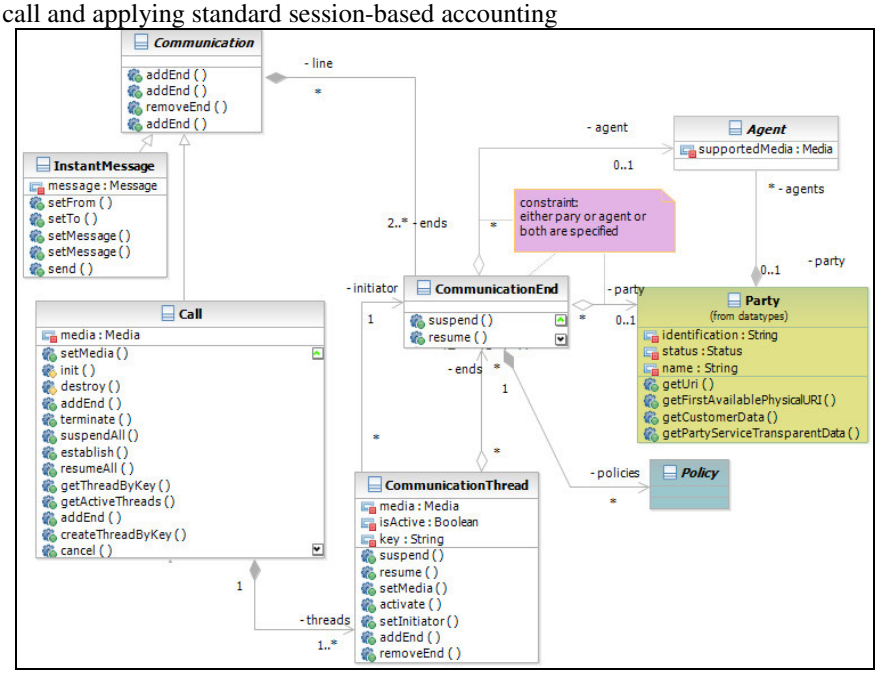

Fig. 6. Communication entities

3. Transformation to Runnable Code

One of the main challenges in this work was defining a transformation to runnable service code, i.e., defining mapping rules and infrastructure that close the abstraction gap. This task also includes support for the UML behavioral semantics and closing its variation points. We put additional emphasis on generating code that is efficient, readable and easy to modify. Note that we do not follow the OMG's MDA process of generating a domain-specific model before generating code, but rather separate the *transformation* into domain-independent and domain-specific parts, without generating an intermediate model. The reason for this is to allow rapid service development without going into low-level platform-specific details. This follows the practice advocated in Mellor and Balcer REF for embedded systems [23].

The transformation infrastructure has three main layers:

A. A generic platform- and protocol-independent layer capturing the structure and behavior-related classes (e.g., Activity abstract class, event managing engines). This layer was designed to be domain-independent so it can be reused by other domains.

B. A generic platform- and protocol-independent layer implementing the telecom model library APIs. This also simplifies service portability (e.g., the ability to run the service on networks with different protocols).

C. A platform- and protocol-specific layer implementing a well-defined specification. This layer's interfaces are used by the other layers hiding the protocol-specific implementation from them. As a result, migrating a service to another platform or protocol requires only replacing a single, well-defined library realizing the defined specification.

Numerous transformations exist that can generate the skeleton code (structure) of an application from a UML model. We extended IBM Rational Software Architect's built-in transformation from UML to Java and added our service-specific rules to generate both the special telecom model structure (e.g., SIP Servlet structure, sip.xml) and behavioral parts of the application.

For example, each model activity is transformed into a class that extends the abstract Activity class from layer "A". Transformation rules populate this class with:

1. A constructor that sets the activity input data and registers it as a listener to events
2. Methods for each action in its flow. Fig. 7 includes the code of the two invoke activity actions seen in the diagram on the left. Each extracts the data from the pins, creates the target activity, invokes it with the data and then sets the returned information in the output pin.
3. Methods that implement the flow graph corresponding to the activity flow .
4. Methods that initiate data flow-related entities and dependencies between them.

The generated activity code has a structure that resembles the source activity. This simplifies understanding and enables code modifications if needed.

One of the main simplification decisions we chose for the designers and decided to keep for coders was that the main service state machine represents the interaction of the service with a single client and not multiple clients. Nevertheless, at runtime, the service needs to interact with multiple clients simultaneously. To enable this, we create a context object for each client interaction that includes all interaction state information. When an event arrives 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 service does not consume too many resources and has reasonable performance.

The implementation of the TSL was complicated, especially in situations where method implementation could not 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* class "establish()" operation is invoked. The implementation of this operation requires managing numerous message exchanges between parties until both are connected. This operation requires interaction with the network to receive events. But network interaction is handled by the servlet, and this operation should not interfere with the regular servlet-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.

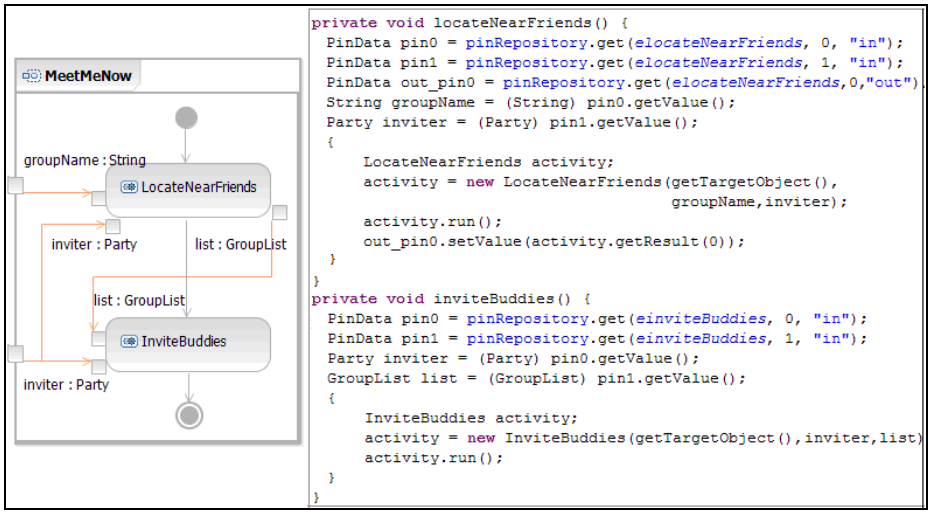

Fig. 7. Generated code segment

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

4. Telecom Service Creation Environment

To ensure TS-DSL can be used by none telecom experts and to ensure its applicability, we implemented the telecom service creation environment (TSCE). The TSCE provides an interface to the TS-DSL introduced in Section 2 and the transformation introduced in Section 3, and provides a level of automation for implementing the service design methodology we adopted.

The TSCE is implemented as set of plug-ins, UML profile, and transformations built on top of the IBM Rational Software Architect, which is itself built on top of the Eclipse tooling framework. As such, the tooling is open and extensible. It is designed in a way that enables the addition of a further level of customization when deploying it in a commercial context.

The telecom model library view (seen on the left of Fig. 8) is a tree view, exposing model library entities that can be used in a service model. The external services view exposes the set of external services used within the model. It also supports importing specifications of other services from the service registry to be used in the service.

The TSCE includes a set of service creation templates. This set is extensible to allow the introduction of additional templates. The basic TSCE has three templates, one for each family identified in Section 2.2.

The TSCE also exposes reusable structures provided with the tool, including a media player and a unit convertor that can be seen in the bottom panel of Fig. 8. New reusable elements may be added by domain experts.

5. Telecom Service Examples

We designed, implemented and deployed several typical telecom services using TSCE. In this section, we describe two of these services: the "Meet Me Now" service (MMNS) and the "Free Calls with Advertisement" service (FCWAS).

To ensure that the services we selected span a wide variety of requirements for telecom services, we worked with a number of sessions with telecom service development experts from within IBM.

These examples illustrate that indeed telecom service development can require only knowledge in UML and TS-DSL and not in underlining telecom protocols and platforms.

5.1 Meet Me Now Service (MMNS)

The primary user of the MMNS has a cellular phone where address book contacts are grouped into buddy lists. 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 where their presence information states that they are available. The implementation of this service uses a buddy list service, a location service, a presence service, and an SMS service.

The service was created in TSCE using the "basic" service family template (see Fig. 3). The service starts in the "Active" state. When an "invokeMeetMeNow" service invocation event arrives, the service collects the data and moves to the "ProcessingMeetMeNow" state for sending SMSs to the appropriate parties. If any error is raised, the flow is interrupted and the service moves to the "ErrorHandling" state. Otherwise, the processing finishes.

To handle the "invokeMeetMeNow" service invocation event, the designer adds a "do" activity to the "ProcessingMeetMeNow" state. This "MeetMeNow" activity is seen on the left of Fig. 7. This activity includes two activity invocation actions:

- "LocateNearFriends" activity includes the logic that determines what sub-group of buddies to send an SMS to.
- "InviteBuddies" activity, seen in Fig. 8, includes the logic that sends an SMS to the available buddies nearby. It uses two ServiceInvocationAction instances that specify the invocation of the findLocation() operation of the location service and the extractBuddyList operation of the BuddyList service. This illustrates the simplicity of invoking external services, where one only needs to make sure all input is supplied and treat it as a regular operation invocation.

During the MMNS Service definition, a structure diagram is built in parallel by the tool, capturing its relationship to external services used in the logic.

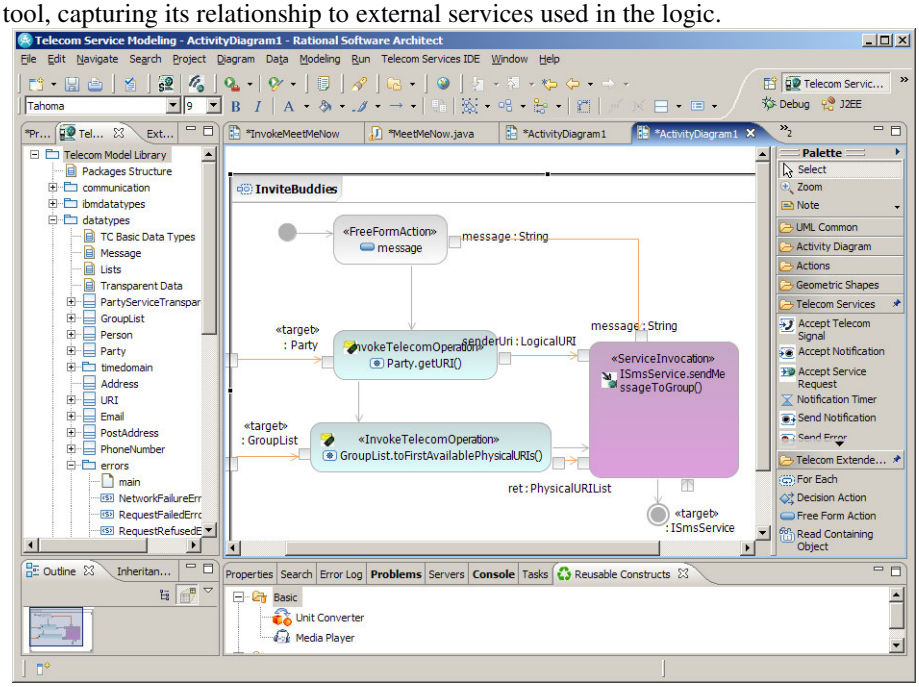

Fig. 8. Telecom service creation environment

5.1 Free Calls with Advertisement Service

The “Free Calls with Advertisement” (FCWAS) service is a different telecom service, where the emphasis is on call manipulation requirements, in contrast to the external service requirements in MMNS. This service allows users to place free-of-charge calls that are interrupted from time-to-time with an audio advertisement. It allows registered “gold” users (users who are registered with higher privileges) to place calls with a better quality line (codec) than regular users.

This service design depends heavily on entities defined in the TSL communication part. In the service logic, a *Call* object is created, with two ends pointing at the call initiator and the specified target. To allow the service to interrupt the call from time to time with an advertisement, a new *CommunicationThread* is added in which the service enrolls itself as an additional end to the call. When an invoked timer sends an event indicating that the time has come to play an advertisement the *Call*'s active threads are switched (i.e. the new *CommunicationThread* status is set to active and the original *CommunicationThread* state is set to suspended) to allow the service to take control over the call and play the advertisement. When the advertisement ends, the threads state is restored to allow the normal continuation of the call. Fig. 9 displays the “Play advertisement" activity that is responsible for switching the threads and playing the advertisement.

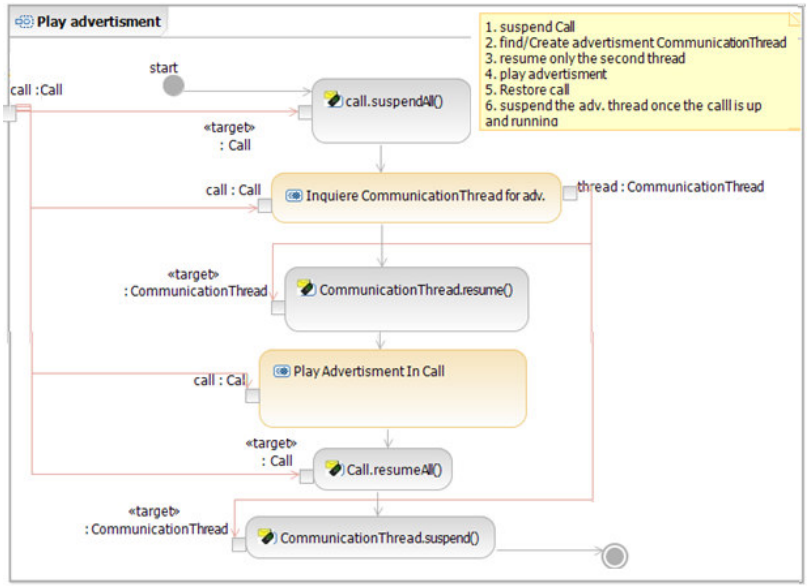

Fig. 9. FCWA "Play advertisement" activity

6. Related Work

In IP telephony, several platforms and tools are available for service development that simplifies the service development process. The IBM WebSphere Application Server (WAS) [10] version 6.1 with IBM WebSphere IMS Connector [11], Oracle BEA WebLogic SIP Server [12], and Avaya SIP Application Server [13] are examples of such platforms. However, they all require the architect to have an understanding of SIP (RFC 3261 [1] for basic signaling support within a telecom service; Diameter base protocol (RFC 3588 [9]) message structure, and message structure for the relevant Diameter applications. Moreover, developers need to understand the interaction between the Diameter and SIP protocols.

The efficient development of telecom application has been addressed by several authors using the model-driven and service-oriented approach. Georgalas et al. [5, 17] 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 latter paper, they describe an Eclipse-based environment of model-driven development (MDD) facilities that supports the automatic generation of domain-specific tools. However, in different from us they do not provide telecom-specific runtime support.

Live Sequence Charts approach and the Play-Engine tool have also been used to model telecommunication applications [18], where they take advantage of use cases

and sequence diagrams as scenario-based requirements describing the behavior of such applications. Their work allows validation of the behavior on the model level and connecting existing components, but it does not have code generation support.

Belaunde et al. [19, 20] 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.

For telecommunication service development, IBM provides a package of software capabilities named IBM Rational Unified Service Creation Environment (USCE). It includes Conallen's SIP modeling toolkit [6, 7]. This toolkit includes a set of domain extensions to the IBM Rational modeling platform for the development of SIP services. Its DSL defines basic SIP service 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 coding the behavioral parts of the service, in different from our approach.

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 from designers. Our approach simplifies service design, cuts down design and development time, makes the design process accessible to designers not familiar with telephony protocol details and simplifies maintenance as the changes are done in a higher level of abstraction.

We introduced the telecom service domain-specific language, telecom library, a transformation to runnable code (including behavior) 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 two representative services.

During this process, we were faced with many research challenges. Most are related to defining the right abstractions (and mapping rules) that hide the low level protocol and platform 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 - not only skeleton code.

We received positive feedback from service developers who developed services with IMS-related protocols. We plan to continue the industrial validation to allow service designers to develop their services with TSCE.

In the future we plan to focus our research on other aspects of model-based development of telecom services. This includes extensions for model-based testing

and debugging at the TS-DSL level. We will do this by defining an extension to the generic model execution engine [21] and using its simulator.

8. References

1. RFC 3261, SIP: Session Initiation Protocol, http://www.ietf.org/rfc/rfc3261.txt
2. RFC 2616, Hypertext Transfer Protocol-- HTTP/1.1, http://www.ietf.org/rfc/rfc2616.txt
3. SOAP Specifications, http://www.w3.org/TR/soap/
4. Gernosik, G., Naiburg, E.: The Value of Modelling, http://www.ibm.com/developerworks/rational/library/6007.html, 2004
5. Georgalas, N., Azmoodeh, M., Ou, S.: Model Driven Integration of Standard Based OSS Components, Proc. of the Eurescom Summit 2005 on Ubiquitous Services and Applications, 2005, Heidelberg, Germany
6. Conallen, J.: Introduction to SIP Modeling Toolkit, http://www.ibm.com/developerworks/rational/library/07/0807_conallen/
7. Conallen, J., Olvovsky, S.: Model-based Development and Testing of Session Initiation Protocol Services, IBM Rational Software Development Conference 2008, Orlando, FL
8. RFC 2327, SDP: Session Description Protocol, http://www.ietf.org/rfc/rfc2327.txt
9. RFC 3588, Diameter Base Protocol, http://www.ietf.org/rfc/rfc3588.txt
10. WebSphere Application Server, version 6.1, http://www-01.ibm.com/software/webservers/appserv/was/
11. WebSphere IP Multimedia Subsystem Connector, http://www-01.ibm.com/software/pervasive/multisubcon/
12. Oracle BEA WebLogic SIP Server, http://www.oracle.com/technology/pub/articles/dev2arch/2006/02/communications-platform.html
13. Avaya SIP Application Server, http://www.avaya.com/gcm/master-usa/en-us/products/offers/sip_application_server.htm
14. Unified Modelling Language (UML2.0), OMG, June 2004, http://www.omg.org/technology/ documents/formal/uml.htm.
15. IBM Rational Software Architect, http://www.ibm.com/developerworks/rational/products/rsa/
16. Service-oriented architecture (SOA) definition, http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html
17. Achilleos, A., Georgalas, N., Yang, K.: *An Open Source Domain-Specific Tools Framework to Support Model Driven Development of OSS*, ECMDA-FA 2007: 1-16
18. Combes, P., Harel, D., Kugler, H.: Modeling and Verification of a Telecommunication Application using Live Sequence Charts and the Play-Engine Tool, ATVA, 2005
19. Presso, M.J., Belaunde, M.: Applying MDA to Voice Applications: An Experience in Building an MDA Tool Chain. ECMDA-FA 2005
20. Belaunde, M., Falcarin, P.: Realizing an MDA and SOA Marriage for the Development of Mobile Services. ECMDA-FA 2008.
21. Kirshin, A., Dotan, D., Hartman, A.: A UML Simulator Based on a Generic Model Execution Engine , MoDELS Workshops 2006 - LNCS, Springer, 2007
22. The TeleManagement Forum (TMF), http://www.tmforum.org
23. Stephen J Mellor & Marc J Balcer Executable UML: A Foundation for Model-driven Architecture (2nd Edition) Addison Wesley 2002.