A SIP-based Programming Framework for Advanced Telephony Applications

Wilfried Jouve, Nicolas Palix, Charles Consel, Patrice Kadionik

A SIP-based Programming Framework
for Advanced Telephony Applications

Wilfried Jouve[1], Nicolas Palix[1],
Charles Consel[1], and Patrice Kadionik[2]

[1] INRIA / LaBRI   [2] IMS / University of Bordeaux
Talence, France   Talence, France

Abstract. The scope of telephony is significantly broadening, providing
users with a variety of communication modes, including presence sta-
tus, instant messaging and videoconferencing. Furthermore, telephony is
being increasingly combined with a number of non-telephony, heteroge-
neous resources, consisting of software entities, such as Web services, and
hardware entities, such as location-tracking devices. This heterogeneity,
compounded with the intricacies of underlying technologies, make the
programming of new telephony applications a daunting task.

This paper proposes an approach to supporting the development of ad-
vanced telephony applications. To do so, we introduce a declarative lan-
guage over Java to define the entities of a target telephony application
area. This definition is passed to a generator to produce a Java program-
ming framework, dedicated to the application area. The generated frame-
works provide service discovery and high-level communication mecha-
nisms. These mechanisms are automatically mapped into SIP, making
our approach compatible with existing SIP infrastructures and entities.
Our work is implemented and has been validated on various advanced
telephony applications.

1 Introduction

In recent years, telephony has dramatically broadened its scope by integrating a
variety of forms of communications, including video, text, and presence events.
This broadened scope has also created a need for telephony applications to in-
voke ever more heterogeneous, non-telephony resources such as Web services,
calendars, and databases. This situation changes significantly the programming
of telephony applications: it now consists of coordinating heterogeneous entities
and exchanging arbitrary types of values. These entities may either be hard-
ware, like a phone terminal, or software, like a Web service. Because telephony
applications involve an increasingly wide range of potential entities, selecting
the appropriate ones is becoming a real challenge. To take up this challenge,
programmers are forced to encode properties, like location or functionalities, in
the entity references, using error prone, ad hoc techniques [25].

To address the extended scope of telephony applications, platforms based on
SIP (Session Initiation Protocol) have, by definition of the protocol, the potential

to provide a rich range of communication forms, namely, instant messaging [9], events [29] and sessions [33]. More specifically, instant messaging is a one-to-one interaction mode; it can, for example, be used to display information about missed calls. Event is a one-to-many interaction mode; it is the preferred mechanism to propagate information such as presence status. Finally, session is a one-to-one interaction mode with data exchanged over a period of time; it is typically used to set up a multimedia stream between users.

Software layers have been added on the top of SIP in an attempt to facilitate the development of applications (*e.g.,* JAIN [37], SIP Servlets [17] and Parlay [14]). However, these layers provide a limited level of abstraction in that they require programmers to have an extended expertise in the underlying building blocks, including the signalling protocol, the API to the protocol-support layer, network protocols and distributed system programming. Going beyond software layers, other approaches have introduced programming languages dedicated to telephony service creation. However, languages such as CPL [31], LESS [40] or VisuCom [23] have a scope limited to end-user services. Others approaches, like SPL [8], propose scripting languages that are restricted to express routing logic.

This paper

This paper proposes an approach to covering the broadened scope of telephony applications and raising the abstraction level for programming these applications. To do so, we provide the programmer with a declarative language, named *DiaSpec*, to define the entities involved in a telephony application area. An entity is characterized by its interaction modes, consisting of the SIP-native interaction modes, namely, messages, events and sessions. However, to cope with non-telephony resources, our approach also provides a command mode, that is an RPC-like mechanism, to invoke entity operations (*e.g.,* a database look-up operation). Entity declarations are passed to a generator, called *DiaGen*, that produces a programming framework in Java dedicated to the target application area. This framework provides the programmer with a discovery service for entities and programming support dedicated to interact with the declared entities. The programmer writes telephony applications using high-level, automatically generated methods, that are mapped into SIP-compliant operations.

The contributions of this paper are as follows.

– We introduce a discovery service integrated into a SIP-based programming framework, making explicit the properties needed to select the relevant entities for an advanced telephony application.
– We provide the programmer with high-level interaction modes, raising the abstraction level of SIP-native operations and introducing a uniform mechanism to invoke non-SIP resources.
– We have developed DiaGen, a generator of programming frameworks for advanced telephony applications that is SIP compliant, enabling applications to abstract over heterogeneous entities.

Outline

The rest of this paper is organized as follows. Section 2 introduces an example to motivate and illustrate our approach. Section 3 presents a methodology, and supporting declaration language, for the development of advanced telephony applications. In Section 4, we describe how our development methodology, and high-level programming support, are automatically mapped into a SIP infrastructure. Section 5 examines the tool chain implementing this mapping. We discuss the related works in Section 6 and conclude in Section 7.

2 Working Example

To motivate our approach, we consider an application that prevents close collaborators from missing each others calls. Currently, a call has to be repeated until the callee answers the phone. When the missed call is returned, the original caller may not be available anymore. To address this situation, our advanced service queues a call between close collaborators when the callee is unavailable and calls back both parties as soon as they become available. We name this application *presence-based automatic callback*. The key interest of our working example is to motivate the need to coordinate a variety of building block services and to illustrate how this need is addressed by the high-level programming support provided by our approach.

Let us now introduce the building block services of presence-based automatic callback. A location manager keeps track of the location of users and publishes this information. A presence manager updates the status of the users depending on their location and activities (*e.g.,* busy on the phone). A virtual phone manages the incoming calls of a user depending on its status: when available, the user gets forwarded a call, otherwise a call from a close collaborator is sent to the *pending call manager*. When both collaborators are available the pending call manager initiates a call between them. A virtual phone provides a useful mediator between the telephony platform and the user; it can serve other purposes beyond presence-based automatic callback. For example, it could define routing logic with respect to a user, regardless of his terminals.

For the sake of simplicity, we do not further detail the presence manager in the remainder of this paper. As a result, the user status is treated by the location manager. Figures 1 and 2 present use cases of presence-based automatic callback. In Figure 1, Bob calls Alice while she is out of her office. His call gets queued in the pending call manager. As shown by Figure 2, when Alice comes back to her office, both Bob and her are invited to initiate a conversation on the phone.

Although conceptually simple, the presence-based automatic callback application relies on (1) SIP events to manage user presence, (2) SIP signalling to forward and initiate calls, and (3) a real-time transport protocol for multimedia sessions. The nature of these operations require programmers to dive into the API of the SIP platform. For example, SIP Servlet provides support for SIP methods and SIP dialog management but it offers low-level interface to manipulate message headers and body. JAIN SLEE supports a high-level interface to

a telephony platform. However, because it is protocol agnostic, it introduces an overly generic interface to cover a range of protocols besides SIP.

Programming advanced telephony applications also involves interacting with heterogeneous SIP entities; in our working example, these entities are hardware (*e.g.,* a phone terminal), software (*e.g.,* a virtual phone), and non-telephony (*e.g.,* a location sensor). Not surprisingly, this heterogeneity is often reflected in the application code impeding future evolutions.
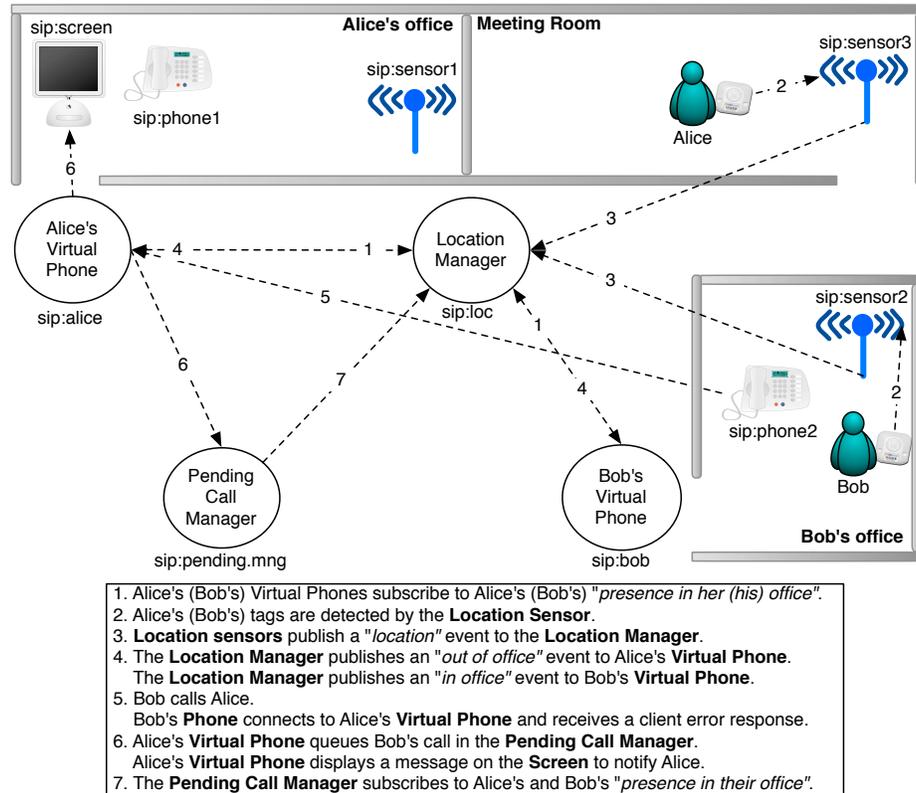

1. Alice's (Bob's) Virtual Phones subscribe to Alice's (Bob's) "*presence in her (his) office*".
2. Alice's (Bob's) tags are detected by the **Location Sensor**.
3. **Location sensors** publish a "*location*" event to the **Location Manager**.
4. The **Location Manager** publishes an "*out of office*" event to Alice's **Virtual Phone**.
   The **Location Manager** publishes an "*in office*" event to Bob's **Virtual Phone**.
5. Bob calls Alice.
   Bob's **Phone** connects to Alice's **Virtual Phone** and receives a client error response.
6. Alice's **Virtual Phone** queues Bob's call in the **Pending Call Manager**.
   Alice's **Virtual Phone** displays a message on the **Screen** to notify Alice.
7. The **Pending Call Manager** subscribes to Alice's and Bob's "*presence in their office*".

Fig. 1. Presence-based Automatic Callback: initial process

3   The DiaGen programming framework

Our approach is based on the notion of *taxonomy* that describes the distributed entities that are relevant to a given application *area* in the telephony *domain*. This description is enriched by attributes that further characterize possible variations. An area is introduced by an architect, whose task is to identify what kinds
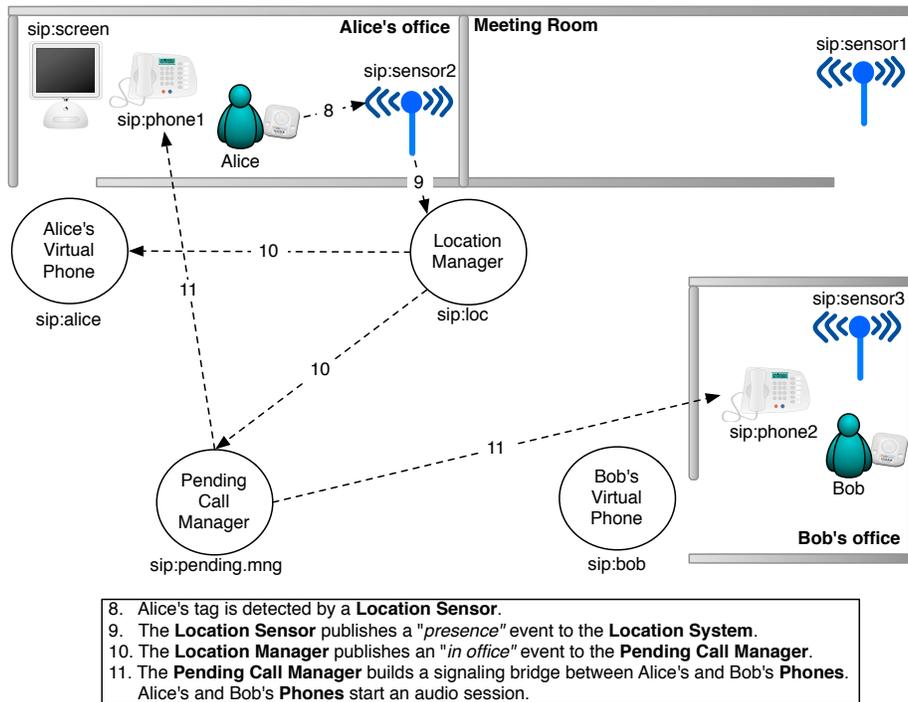
8. Alice's tag is detected by a **Location Sensor**.
9. The **Location Sensor** publishes a "*presence*" event to the **Location System**.
10. The **Location Manager** publishes an "*in office*" event to the **Pending Call Manager**.
11. The **Pending Call Manager** builds a signaling bridge between Alice's and Bob's **Phones**.
    Alice's and Bob's **Phones** start an audio session.

Fig. 2. Presence-based Automatic Callback: a scenario

of entities are relevant and how they should interact. Application programmers then use the area definition to guide and structure the implementations of the services provided by the various entities. A DiaSpec area specifies not only what kinds of data can be exchanged between entities, as is declared by an interface, but also which kinds of entities can participate in a given interaction, and the mode by which the interaction should be carried out. DiaGen, the compiler for DiaSpec, translates these declarations to Java code that is constructed such that many of the expressed constraints are checked statically by the Java type checker. The generated Java code relies on SIP to carry out interaction between entities, as detailed in Section 4.

3.1 The DiaSpec Specification language

A DiaSpec area is expressed as a collection of *service classes*, each of which represents a set of entities sharing common functionalities. We refer to these entities as *services*. Associated with each service class is a collection of *attributes*, characterizing the non-functional properties of the associated services, and a collection of *interaction modes*, specifying how the associated services may interact with other entities. Finally, service classes and attribute values are organized hierarchically, permitting more specific instances to be used where less specific

instances are required, thanks to polymorphism. In the rest of this section, we describe the declaration of the attributes, interaction modes, and hierarchical relationships in more detail, using the fragments of our application area shown in Figure 3 and Figure 4.
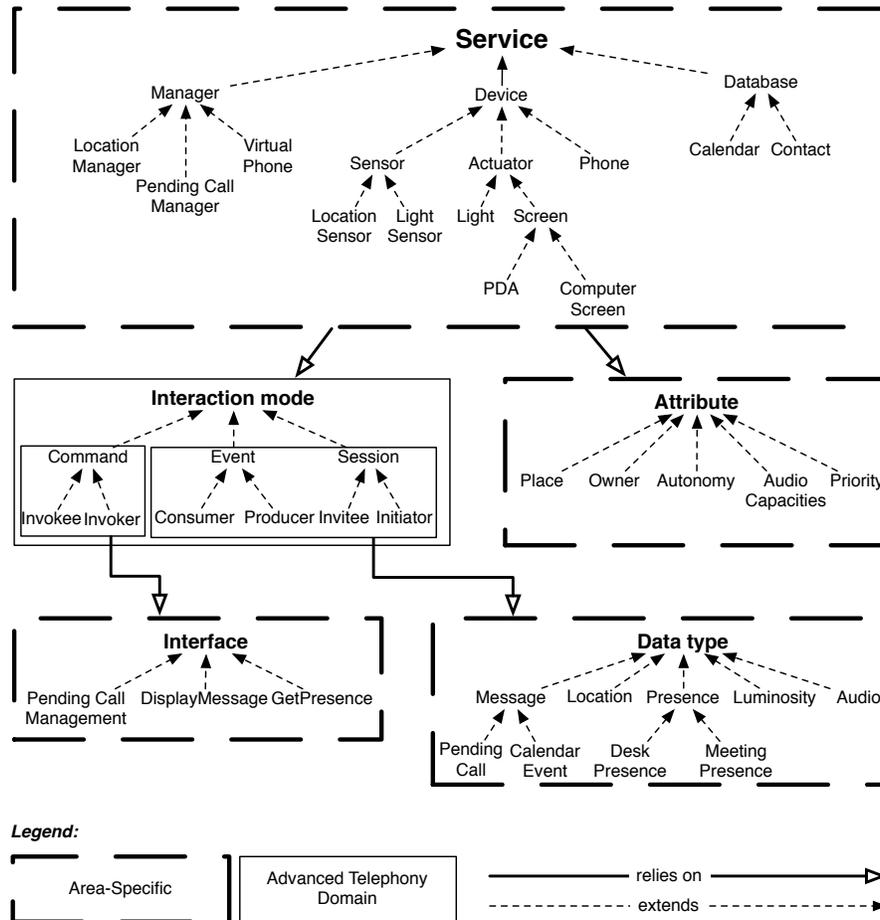

Fig. 3. Excerpts of hierarchies for the target application area

Attributes The attributes of a service class describe the range of entities that the service class corresponds to, in terms of their non-functional properties (*e.g.,* location). Attributes are specified in a parenthesized list following the name of the service class. For example, in Figure 4, the service class `Phone` introduces the specified attribute `AudioCapabilities`. An attribute is a Java data type, allowing

```
service   Service(Owner) {
}
service   Device(Location, Autonomy) extends   Service {
}
service   LocationSensor() extends   Sensor {
  provides event Location
}
service   Calendar() extends   Database {
  provides event Info
}
service   Phone(AudioCapacities) extends   Device {
  provides session Audio
  requires session Audio
}
service   Screen() extends   Device {
  provides command DisplayMessage
  requires command PendingCallManagement
}
service   Manager(Priority) extends   Service {
  provides command StartStop
}
service   VirtualPhone() extends   Manager {
  requires session Audio
  requires event DeskPresence
  requires command PendingCallManagement
  provides session Audio
}
service   LocationManager() extends   Manager {
  requires event Location
  provides command GetPresence
  provides session Presence
  provides event Presence
}
service   PendingCallManager() extends   Manager {
  requires event DeskPresence
  requires command DisplayPendingCall
  provides command PendingCallManagement
  binds session Audio
}
```

Fig. 4. Excerpts of an application area in DiaSpec

sub-classing between attribute values according to the Java class hierarchy. Figure 3 shows various hierarchies extracted from our application area. Attributes are used within a service implementation, in the service discovery process, both to characterize the implemented service and to specify the set of entities with which the service needs to interact.

Interaction modes The interaction modes associated with a service class describe how its services produce or consume data. DiaSpec supports three kinds of interaction modes: *commands*, *events*, and *sessions*. A command interaction amounts to a Remote Procedure Call (RPC), allowing a one-to-one interaction between distributed entities. An event interaction implements the standard publish/subscribe paradigm [27], in which an event publisher broadcasts an event notification to a collection of subscribers. A session interaction natively supports

negotiation of session parameters and exchange of a stream of data. The declared interaction modes strictly define how an entity can interact with other entities.

A interaction mode declaration specifies the interaction mode and indicates whether a service provides or requires the interaction. Such a declaration has either of the following forms:

> `provides` *interactionMode*
> `requires` *interactionMode*

In the former case, the service, acting as a server, must implement the interaction mode. In the latter case, the service, acting as a client, may use the required interaction mode.

The sub-term *interactionMode* specifies the mode of a given interaction. A mode is either `command`, `event`, or `session` combined with the name of a Java type characterizing the interaction. For a command, the Java type is an interface listing the relevant methods (*e.g.,* `DeskPresence`). For an event or session, the Java type indicates the type of the data that are exchanged. These Java types are, like the attributes, organized into a hierarchy. For example, Figure 3 presents the data type hierarchy, containing two sub-nodes to `Presence`, namely, `DeskPresence` and `MeetingPresence`. A service could subscribe to either event type or to the generic `Presence` event type.

To illustrate these declarations in more detail, we consider the `Location-Manager` and `LocationSensor` service classes, shown in Figure 4. The `Location-Sensor` service class provides a single event functionality and then, produces `Location` event (*i.e.,* Cartesian coordinates), to the `LocationManager` service class. The `LocationManager` service class in turn requires `Location` events. Based on this location, the `LocationManager` service can then provide information about user presence in a predefined set of places (*e.g.,* office rooms, meeting rooms or corridors), as a command (*i.e.,* `getPresence`), as a session of `Presence` data or as a `Presence` event (*e.g.,* `DeskPresence`).

Hierarchical relationships The DiaSpec description of a distributed environment of entities is structured as a hierarchy, as illustrated by Figure 3 for the definition of our application area. In a service class declaration, a sub-class relationship is specified using the `extends` keyword. Starting at the root node, the hierarchy breaks down the set of possible entities of this area into increasingly specific service classes. Each successive entry adds new attributes and interaction modes that are specific to the service class it represents. A service class furthermore inherits all the attributes and interaction modes of its ancestors.

In our approach, inheritance plays a decisive role in service discovery. Conceptually, a developer who wants access to a service class designates the corresponding node in the service hierarchy, and receives all of the services corresponding to the service classes contained in the sub-tree. This strategy implies that code that implements a service class should be associated with a node as low as possible in the hierarchy, to most precisely expose to users the variety of functionalities provided by the service. Yet, code that uses a service class should choose the

least detailed class of services that meets its needs. In doing so, (1) a service discovery request is more likely to be successful and to return a larger number of entities; (2) the resulting application only exposes the functionalities it requires, thus improving its portability, and making it forward compatible with future or refined versions of the requested service class.

3.2 Developing Advanced Telephony Services

To develop a new service in our approach, the programmer first determines the service class it should belong to. The declarations of the selected service class then provide the programmer with an area-specific design framework for implementing all the facets of the service, ranging from its operations to its deployment. This design framework is supported by a programming framework that is automatically generated from the DiaSpec specification. We assume that code is implemented in Java.

The interface provided to programmers For each DiaSpec specification, the generated framework provides an abstract class that contains methods and instance variables, corresponding to interaction modes and attributes, respectively. Definitions are generated for methods that solely depend on the information provided by the DiaSpec specification. For example, a method for publishing an event only depends on the type of the event, and thus is defined in the abstract class, whereas commands are represented by abstract methods that the programmer must implement in the service. This organization implies that the programmer can focus his development effort on the application logic. This use of abstract classes also triggers the generation of programming support in an Integrated Development Environment (IDE), such as Eclipse.

Definition of services To create a service, the programmer first extends the abstract class corresponding to the selected DiaSpec service class. In an IDE such as Eclipse, this action triggers the creation of a class skeleton (or class stub in Eclipse parlance) that the programmer must then fill in with the service code. We now describe the subsequent programming process, and the support that Eclipse provides.

Interaction modes. The interaction modes are represented by the methods and abstract methods of the abstract class. As examples, we use the implementation of a `PendingCallManager` service (Figure 5 and Figure 6).

Command. Commands are represented by abstract methods in the abstract class. From these declarations, Eclipse generates method stubs[1] for all of the commands that need to be implemented. As shown in Figure 5, for the pending call manager, these methods are `addPendingCall`, `deletePendingCall`, `start` and `stop` declared by the DiaSpec nodes `PendingCallManager` and `Manager` (`PendingCall-Management` and `StartStop` interfaces in Figure 4).

---

[1] We refer to *method stubs* as methods generated by an IDE, whereas stubs refer to client stubs created using an IDL compiler.

```
public class MyPendingCallManager extends PendingCallManager {

  public MyPendingCallManager(String uri) {
    super(uri);
    // TODO Auto-generated constructor stub
  }

  public void receive(DeskOffice event, Service source) {
    // TODO Auto-generated method stub
  }

  public void addPendingCall(Person caller, Person callee) {
    // TODO Auto-generated method stub
  }

  public void deletePendingCall(int pendingCallID) {
    // TODO Auto-generated method stub
  }

  public void start() {
    // TODO Auto-generated method stub
  }

  public void stop() {
    // TODO Auto-generated method stub
  }

}
```

Fig. 5. The MyPendingCallManager class skeleton (from Eclipse)

When a service needs to invoke a command from another service, it calls the associated method of this service, which embeds an RPC call. For example, the `MyPendingCallManager` service shown in Figure 6 may need to display a notification message on a screen. Line 22 of this service invokes the `display` method of a previously obtained screen.

Event. For a DiaSpec service class that provides an event interaction mode, the corresponding abstract class defines a `publish` method for each declared type of output event. Services implementing the interface definition invoke these `publish` methods to publish the corresponding event. In our Location Manager example, the produced event is a Presence event. Consequently, the Location Manager publishes an event whenever the location of a contact changes. This event will be received via an event channel by all services that have subscribed to it.

For a DiaSpec specification that provides an event interaction mode, the corresponding abstract class also defines a `receive` abstract method for each declared type of input event. Such a method has an argument of the corresponding event type. From this declaration, Eclipse generates a method stub, which is to be filled in by the programmer. Lines 13-16 of Figure 6 show the method definition provided by the programmer for `MyPendingCallManager`, which implements the `PendingCallManager` service class that can receive `DeskPresence` events. The pending call manager may then subscribe to `DeskPresence` events from various sources. In Figure 6, line 11, it subscribes to all available `LocationManager` services in the building A by invoking `subscribe`.

```
1.  public class MyPendingCallManager extends PendingCallManager {
2.   private LinkedList<LocationManager> myLocManagers;
     [...]
3.   public MyPendingCallManager(String uri) {
4.     super(uri);
5.     priority.setValue(Priority.HIGH);
6.     owner.setValue(Owner.ADMIN);

7.     LocationManagerFilter filter = LocationManager.getFilter();
8.     filter.location.setValue(Location.BuildingA);
9.     myLocManagers = LocationManager.getServices(filter);
     [...]
10.    for (LocationManager myLocManager:myLocManagers)
11.      myLocManager.subscribe((IDeskPresenceEventInput)this);
12.  }

13.  public void receive(DeskPresence event, Service source) {
14.    PendingCall pendingCall = updatePendingCalls(
         event.getPerson(),
         event.isInOffice()
       );

15.    AudioSession session = bind(
         pendingCall.getCalleePhone(),
         pendingCall.getCallerPhone()
       );
       [...]
16.  }

17.  public void addPendingCall(Person caller, Person callee) {
18.    PendingCall pendingCall = addPendingCallToDB(caller, callee);

19.    ScreenFilter filter = Screen.getFilter();
20.    filter.location.setValue(callee.getDeskLocation());
21.    Screen calleeScreen = Screen.getService(filter);

22.    calleeScreen.display(pendingCall);
23.  }

24.  public void deletePendingCall(int pendingCallID) {
       [...]
25.  }

     // Updates pending calls with caller/callee presence.
     // Returns the next pending call where caller and callee
     // are in their office.
26.  private PendingCall updatePendingCalls(Person person, boolean isInOffice) {
       [...]
27.  }

     // Stores pending calls.
28.  private void addPendingCallToDB(Person caller, Person callee) {
       [...]
29.  }
     [...]
30. }
```

Fig. 6. A Pending Call Manager

Session. The code relevant to a session is similar to that of an event: services declared as session invitee lead to the creation of the `connect` and `disconnect` method stubs, whereas services in service classes declared as session initiator invoke these methods to receive a stream of data. Services in service classes

declaring session binder use the `bind` method to establish a session between two services that have session capabilities. In our example, the `Phone` services are invitees and the `PendingCallManager` service is an audio session binder. As such, the pending call manager can establish a session between the caller's `Phone` and the corresponding callee's `Phone` (line 15 of Figure 6).

Attributes. The programmer must initialize the values of the attributes in the constructor of the service. In doing so, the service is characterized, enabling other services to discover it. Such properties (*i.e.*, priority and owner) are initialized in the constructor of the pending call manager as shown in lines 5-6 of Figure 6. This constructor first invokes the constructor of the abstract class, via a call to `super`, to register the service in the SIP registrar. In our approach, this constructor always requires a URI as an argument, as this information is necessary to identify the service.

Service discovery The programming framework that is generated from a Dia-Spec specification provides the programmer with methods to select any node in a service class hierarchy. The result of this selection is a set of all services corresponding to the selected node and its sub-nodes, which we refer to as a *filter*. From a filter, the programmer can further narrow down the service discovery process by specifying the desired values of the attributes. Eventually, the method `getServices` or `getService` is invoked to obtain a list of matching services or one service chosen at random from this list, respectively.

Although a DiaSpec specification characterizes a telephony application area statically, our approach still permits services to be introduced dynamically within a filter. In doing so, we offer an alternative to the string-based discovery process by supplying selection operations generated from the DiaSpec specification, making the discovery logic safe with respect to the service class hierarchy.

As an example, consider the `MyPendingCallManager` service that should get the presence of people at their desk, in the building A. It obtains a filter by selecting the `LocationManager` node (line 7) and then sets the `location` attribute to limit the scope of Location Manager to those that are in the `BuildingA` (line 8). The operation `getServices` (line 9) then returns the location managers corresponding to this request, including its sub-nodes, if any. Finally, lines 10-11 iterate over all of the obtained location managers to subscribe to each `DeskPresence` event.

4 Mapping DiaGen into SIP

Despite the rich features of SIP, there is still a gap to fill between this protocol and the level of abstraction provided by DiaSpec. For example, SIP does not support an RPC-like mechanism, which is critical to command a wide range of entities like Web services. For another example, multimedia support mainly addresses major audio and video formats. However, advanced telephony applications involve other kinds of streams like locations and luminosity measurements.

In this section, we propose directions to extend the scope of SIP, leveraging on existing mechanisms and standards. The goal of the proposed extensions is

to provide the programmer with dedicated programming support by introducing service discovery and high-level interaction modes, consisting of command, event and session.

4.1 Message bodies

Targeting advanced telephony applications requires to keep pace with a constant flow of new devices and formats of exchanged data. To cope with this situation, SIP message bodies need to include data of arbitrary types. A number of proposals has been made to describe message body types for SIP messages. These proposals include body types for service description (*e.g.*, SDP [15]), service functionality invocation (*e.g.*, DMP [20]) and exchanged data (*e.g.*, CPIM [21]). Unfortunately, these proposals are not interoperable and force developers to create and manipulate numerous libraries to marshal and unmarshal message bodies.

Our approach abstracts over these issues by directly relying on the programmers' Java declarations. More specifically, Java *data types* are processed by the DiaGen compiler to automatically produce marshaling and unmarshaling support. DiaGen uses the SOAP protocol to encode values in SIP messages. This format is standardized, XML-based and widely used. Although verbose, SOAP is increasingly supported by embedded systems (*e.g.*, XPath offers an API for manipulating XML in embedded systems). Furthermore, some implementations are now written in C, providing high performance and low memory footprint. A key advantage of our approach is that we can develop new back-ends to DiaGen to target other data formats (*e.g.,* the eXternal Data Representation (XDR) data format [11]).

4.2 Service Discovery

In the advanced telephony domain, new services appear and disappear over time (*e.g.*, SIP phones are switched on/off). Fortunately, SIP provides a way of handling this dynamic behavior via the registration process that supports mobility. To do so, SIP entities register their SIP URI with the registration server. This server associates SIP URIs and network addresses of the entities (*i.e.*, IP address and port). When communicating, SIP entities have their network address looked up from their URI by the proxy server. Although the SIP URI is a useful building block to provide service discovery, it does not take into account attributes that are needed to refine entity selection (*e.g.,* a device location or its rendering capabilities).

To circumvent this limitation, approaches consist of associating the characteristics of an entity with its SIP URI. This is either done by introducing parameters to the SIP URI [39] or by directly encoding characteristics of an entity in the string of its URI [25]. These approaches lack abstraction and are error-prone.

To solve this issue, our approach provides the programmer with an abstraction layer over SIP URIs. This layer corresponds to the hierarchy of service

classes introduced earlier; this hierarchy characterizes the entities of an application area. To discover one or more entities of a service class, the programmer makes a request consisting of the service class name, refined by values assigned to attributes. The discovery process produces a list of SIP URIs that matches the request.

To map our discovery process into SIP, we introduce a service broker. This component is notified by the registration server every time a REGISTER request is received. When notified, the service broker sends an OPTIONS request to the newly registered service to collect the name of its service class and its attributes. The service broker is also invoked when an application requests services. To do so, a request consists of the name of the service class and attribute values. When issued by the operation `getServices`, all registered services matching the request are returned. Alternatively, the operation `getService` can be used to obtain a unique, randomly chosen, service from services matching the request.

4.3   Interaction modes

Once discovered, a service is invoked to perform specific tasks. To do so, our approach provides the programmer with dedicated programming support and high-level interaction mechanisms. It extends and builds upon SIP operations.

Session SIP supports multimedia sessions that are described using SDP. Negotiation of SDP session parameters is based on the offer/answer model [32]. Once negotiated, the session is launched and a stream of data gets transmitted. To deliver multimedia streams, the Real-time Transport Protocol (RTP) is widely used [34]. Today, the combination of SDP and RTP is widely used to deal with multimedia sessions from negotiation to streaming. However, this combination falls short of addressing other kinds of streams. Yet, emerging applications combine telephony with sensors streaming a variety of measurements to track people's location, to detect intrusion, to remotely monitor devices, *etc.*

To address these new situations, our approach consists of generalizing SDP to any Java data type, while re-using its negotiation model. Specifically, Java data types are used by DiaGen to generate a codec that takes the form of a serializer, allowing RTP to transmit streams of any data type. In doing so, our approach leverages existing technologies and APIs such as JMF [4].

Event SIP event notification is mostly used for presence [29, 30, 36]. DiaSpec helps specifying a wide variety of event types that are not supported by available packages. To do so, we do not create a new event package for each new event type. Instead, we define a unique *generic* event package that supports arbitrary data types. As described previously, new message body types are automatically introduced by DiaGen.

The implementation of this event package is compliant to the event framework defined in existing protocol specifications [28, 26]. Using the SUBSCRIBE request, event consumers subscribe to both a service class and a data type (*e.g.*,

Location event of location sensors). A notification server manages these subscriptions. Event producers publish events to the notification server that notifies corresponding event consumers using respectively the SIP methods PUBLISH [26] and NOTIFY [28].

4.4 Interoperability with external services

Any SIP-compliant resource is natively supported by DiaGen generated programming frameworks. However, extended forms of telephony application rely on increasingly many non-SIP resources. To integrate these resources, the programmer mixes SIP with other technologies, complicating the development process. To resolve this issue, we propose a uniform SIP-centric development approach to interoperating with external resources. In this approach, non-SIP resources are wrapped to become SIP compliant. For example, we developed a X10 gateway to convert SIP messages, representing commands to appliances, into X10 commands [5]. For another example, we wrapped a MySQL database to convert commands to MySQL requests.

Wrapping non-SIP compliant services A great number of services from databases, to lights, to cameras is interfaced via the command interaction mode; it is used to control or query the status of these entities. Examples of commands include databases accessed via Web Service invocations, lights via X10 primitives and cameras via UPnP operations. However, despite various attempts (*e.g.,* the DO message [38]), SIP does not propose a standardized format to represent and transport commands.

The DiaGen framework extends the use of the standardized SIP MESSAGE request, initially defined for instant messaging, to handle RPC-like interactions [22, 7]. To differentiate instant messaging exchanges from DiaGen commands, we use the content type header whose value is set to `application/soap+xml`. The return value of a command is included in the response message. Both command invocation and returned value are represented as SOAP messages. Existing tools are used to automatically generate (un-)marshaling procedures for Java data types.

The key benefit of our approach is that SOAP is a de facto standard, making our generated programming frameworks inherently interoperable with existing Web services, without requiring a dedicated gateway [24, 10].

Interoperability with existing SIP services Our generated programming frameworks supports native SIP entities, ranging from SIP phones to instant messaging clients. Because of its momentum, SIP should soon be spreading to other areas, making SIP-compliant devices that are not directly related to telephony (*e.g.,* SIP webcams and displays). This situation should motivate further the need for dedicated programming support.

To deal with native SIP entities, DiaGen needs to convert SIP message body types into Java data types, and conversely. These conversions concern the SDP

format for service registration and session establishment, the Presence package for events, the CPIM message body for instant messaging, and DTMF digits contained by INFO messages. The DiaSpec specification imports these SIP native message bodies as data types to either create richer data types or reuse them directly. Then, developers manipulate these legacy bodies as any other new data types. The DiaGen framework provides automatic conversion to interact with native SIP services. The service broker annotates these services as legacy services at registration time. From these annotations, a dedicated service proxy is generated; it performs the appropriate conversion automatically.

5 Implementation

The information provided in a DiaSpec area has been designed to enable verifications and programming support generation, throughout the life-cycle of a telephony application: from design, to development, to deployment, to run time. To do so, we have developed DiaGen, a tool chain that processes DiaSpec declarations and telephony applications.

5.1 The DiaGen Compiler

Figure 7 shows the main processing steps of our approach: area compilation and service compilation. Initially, a DiaSpec area specification is passed to the area compiler, which (1) verifies the consistency of the area, (2) generates an area-specific programming framework. Using the generated framework, programmers then develop various services forming the telephony application. Each service is then passed to the service compiler, which (1) checks that Java classes implementing the service conform to the DiaSpec specification and (2) invokes a Java compiler to generate a class file.
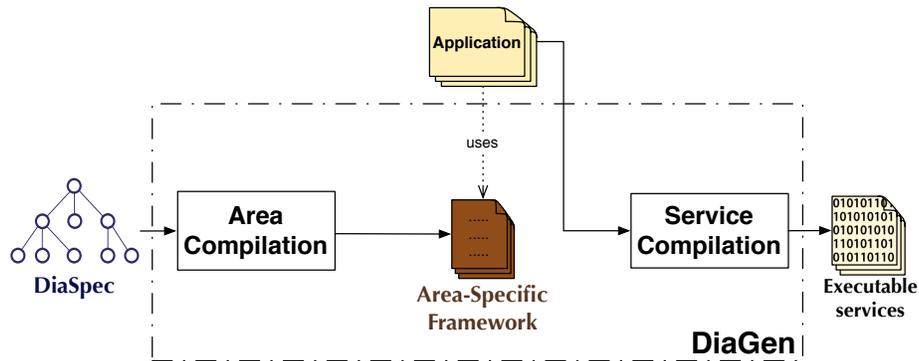

Fig. 7. Overview of our processing chain

DiaGen parses and analyzes a DiaSpec specification, and generates Java code. To do this processing, we use the tools JastAdd [13, 16] and JastAddJ [12]. JastAdd is a meta-compiler providing advanced support for constructing modular and extensible compilers. JastAdd has been used to develop an extensible Java 5.0 compiler named JastAddJ. JastAddJ makes it possible to easily introduce new language constructs into Java, and to implement new static analyses. We use JastAddJ to introduce DiaSpec declarations, to analyze them, and to generate Java classes and interfaces.

5.2 The DiaGen Architecture

The OpenSER SIP server is used as a SIP proxy server, a SIP registration server and a SIP location server [3] in the DiaGen architecture. The DiaGen services, the service broker and the notification server act as SIP entities. As such, they register to the OpenSER server. OpenSER is configured to send a raw copy of every REGISTER message to the service broker (except its own) to notify it of a newly registered service. Upon receipt of this message, the service broker sends a OPTIONS request, as described in Section 4, to collect the service class and its attributes. DiaGen services, the service broker and the notification server use the Java API of Jain-SIP for signalling purposes [1]. Java data types are serialized in SOAP using kSOAP [2]. kSOAP is a SOAP Web service library for resource-constrained Java environments such as J2ME applications.

5.3 Scenarios

We are collaborating with a telecommunications company on a project aimed to develop scenarios combining telephony with home automation. This project uses an ADSL modem as a gateway to enable homeowners to control their home remotely and to allow home residents to interact with heterogeneous services. The first step of this project was to characterize the set of entities (*e.g.*, devices and servers) that would be required to carry out a number of existing and future scenarios. This phase led to an environment description written in DiaSpec, enabling this knowledge to be shared among area experts and application programmers. DiaSpec was then passed to DiaGen to generate a framework customized with respect to home automation. Various scenarios are now being developed to validate the approach, involving entities such as SIP phones, entry phones, PDAs, video cameras, lights, doors, luminance sensors, Web services, Google calendar and digital media receivers. These scenarios include the Entrance Manager and the Surveillance Manager. The Entrance Manager connects entry phones to homeowners' cell phones for an audio session and allows the homeowners to remotely open the door using their cell phones. When an intrusion is detected, the Surveillance Manager sends a text message via SMS to the homeowner, sends an audio message to the police phone, records the intrusion scene using video cameras and sends the recorded video to the homeowner via email.

6 Related work

Several research projects have used SIP beyond the telephony domain. TinySIP is a modified version of SIP for use by wireless sensors in a hospital environment [22]. Berger *et al.* leverage the SIP event model to determine user location in ubiquitous computing environments [7]. However, their approach has a limited scope. Moyer *et al.* argue for the use of SIP in wide-area secure networked devices [25]; however, this proposal has not been studied further. In contrast with our approach, these works rely on existing, low-level programming frameworks (*e.g.*, JAIN SIP, SIP Servlet). As a result, the programmer needs to manage the intricacies of the underlying technologies, such as protocols, network layers, and signalling.

SPL is a domain-specific language (DSL) whose goal is to ease the development of robust telephony services [8]. Although SPL provides high-level programming abstractions, it is still close to SIP and only addresses the routing logic.

Rosenberg *et al.* have emphasized the need for programming support that is dedicated to Internet telephony services [31]. They propose CPL, an XML-based scripting language for describing and controlling call services. Wu *et al.* propose another XML-based language, LESS, to program end-system services [40]. Services written using CPL and LESS scripting languages are mostly limited to coarse-grained processing and dedicated operations. VisuCom is an environment to graphically create telephony services [23]. It support the development of routing services with respect to non-telephony resources like agendas and phone directories. VisuCom includes various verifications. Examples of errors detected in services include call loss, incorrect state transitions, and unbounded resource usage.

These high-level languages target non-programmers and provides static verifications. However, they focus on end-user services and call processing. As a consequence, they lack expressivity to program advanced telephony applications as targeted DiaSpec.

Existing, general-purpose middlewares (*e.g.,* CORBA, DCOM) are highly flexible and support a large number of features to ease the development of distributed applications in a wide range of application areas. This genericity can, however, be a burden when it comes to address requirements from specific domains such as telephony, as adaptation code must be developed to match the application's needs [6, 19]. Moreover, associated programming frameworks force developers to use unsafe and generic structures of code. In contrast, the DiaGen approach, by generating programming support, provides developers with structures of code typed with respect to the target area, as captured by the DiaSpec specification.

CINEMA is a SIP-based infrastructure that enables multimedia collaboration via IP phones, instant messaging and e-mail, among other forms of communication [18]. In CINEMA, service development relies on low-level programming support (*e.g.*, SIP Servlet) or end-user oriented languages (*i.e.*, CPL and LESS). In contrast, our approach provides high-level programming support and targets

a wide range of applications involving a variety of telephony and non-telephony resources. Continuing the work on CINEMA, Shacham *et al.* address the use of heterogeneous devices in ubiquitous environments [35]. They introduce location-based device discovery and customization for session interactions using SIP. Unlike DiaGen, they focus on a single interaction mode (*i.e.*, session). Moreover, they do not show how to interface and integrate these devices in distributed applications.

7   Conclusion

In this paper, we argued that the broadened scope of the telephony domain makes the development of services an overwhelming challenge. To take up this challenge, we proposed a two-step approach to developing applications. First, declarations define the entities involved in a telephony application area and how they interact with each other. Then, these declarations are processed by a generator to produce a programming framework dedicated to the target application area. Generated frameworks provide high-level programming support, including service discovery and communication mechanisms. They are SIP compliant and are automatically mapped into a SIP infrastructure. A generated framework forms a programming layer, abstracting over underlying technologies.

Our approach has been implemented and validated on advanced telephony applications, running over a SIP infrastructure and invoking SIP entities. Currently, we are extending the scope of our approach to a wide variety of devices and software components, to assess the generality of SIP as a general-purpose *software bus* to coordinate distributed entities.

Acknowledgement

References

1. JAIN-SIP, JAVA API for SIP Signaling, https://jain-sip.dev.java.net.
2. kSOAP 2, http://ksoap2.sourceforge.net.
3. OpenSER - the open source SIP server, http://www.openser.org.
4. SUN microsystems, Java Media Framework API (JMF), http://java.sun.com/products/java-media/ jmf/.
5. X10 communication protocol. http://www.x10.org.
6. S. Apel and K. Bohm. Towards the development of ubiquitous middleware product lines. In *ASE'04 SEM Workshop*, volume 3437 of *Lecture Notes in Computer Science*, pages 137–153, Linz, Austria, 2005.
7. S. Berger, H. Schulzrinne, S. Sidiroglou, and X. Wu. Ubiquitous computing using SIP. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 82–89, New York, NY, USA, 2003. ACM.

8. L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, and L. Réveillère. Language technology for Internet-telephony service creation. In *IEEE International Conference on Communications*, June 2006.

9. B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. Session initiation protocol (SIP) extension for instant messaging. RFC 3428, IETF, 2002.

10. W. Chou, L. Li, and F. Liu. Web service enablement of communication services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 393–400, Washington, DC, USA, 2005. IEEE Computer Society.

11. M. Eisler. XDR: External Data Representation Standard. RFC 4506, IETF, May 2006.

12. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM Press.

13. T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

14. R. Glitho and A. Poulin. A high-level service creation environment for Parlay in a SIP environment. *ICC 2002. IEEE International Conference on Communications*, 4:2008–2013 vol.4, 2002.

15. M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, IETF, 1998.

16. G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

17. Java Community Process. *SIP Servlet API*, 2003. http://jcp.org/en/jsr/detail?id=116.

18. W. Jiang, J. Lennox, S. Narayanan, H. Schulzrinne, K. Singh, and X. Wu. Integrating Internet telephony services. *IEEE Internet Computing*, 6(3):64–72, 2002.

19. W. Jouve, N. Ibrahim, L. Réveillère, F. Le Mouël, and C. Consel. Building home monitoring applications: From design to implementation into the Amigo middleware. In *ICPCA '07: IEEE International Conference on Pervasive Computing and Applications*, pages 231–236, 2007.

20. S. Khurana, P. Gurung, and A. Dutta. Device Message Protocol (DMP): An XML based format for wide area communication with networked appliances. Internet draft, IETF, 2000.

21. G. Klyne and D. Atkins. Common Presence and Instant Messaging (CPIM): Message format. RFC 3862, IETF, 2004.

22. S. Krishnamurthy and L. Lange. Distributed interactions with wireless sensors using TinySIP for hospital automation. In *PerSeNS'08: The 4th International Workshop on Sensor Networks and Systems for Pervasive Computing*, Hong-Kong, China, 2008. IEEE.

23. F. Latry, J. Mercadal, and C. Consel. Staging Telephony Service Creation: A Language Approach. In *Principles, Systems and Applications of IP Telecommunications, IPTComm*, New-York, NY, USA, July 2007. ACM Press.

24. F. Liu, W. Chou, L. Li, and J. Li. WSIP - Web service SIP endpoint for converged multimedia/multimodal communication over IP. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 690, Washington, DC, USA, 2004. IEEE Computer Society.

25. S. Moyer, D. Marples, and S. Tsang. A protocol for wide area secure networked appliance communication. *Communications Magazine, IEEE*, 39(10):52–59, Oct 2001.

26. A. Niemi. Session initiation protocol (SIP) extension for event state publication. RFC 3903, IETF, 2004.

27. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, New York, NY, USA, 1993. ACM Press.

28. A. B. Roach. Session Initiation Protocol (SIP)-specific event notification. RFC 3265, IETF, 2002.

29. J. Rosenberg. A presence event package for the session initiation protocol SIP : Session Initiation Protocol. RFC 3856, IETF, 2004.

30. J. Rosenberg. A watcher information event template-package for the Session Initiation Protocol (SIP). RFC 3857, IETF, 2004.

31. J. Rosenberg, J. Lennox, and H. Schulzrinne. Programming Internet telephony services. *IEEE Internet Computing*, 3(3):63–72, 1999.

32. J. Rosenberg and H. Schulzrinne. An offer/answer model with the Session Description Protocol (SDP). RFC 3264, IETF, 2002.

33. Rosenberg, J. et al. SIP : Session Initiation Protocol. RFC 3261, IETF, June 2002.

34. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, IETF, 2003.

35. R. Shacham, H. Schulzrinne, S. Thakolsri, and W. Kellerer. Ubiquitous device personalization and use: The next generation of IP multimedia communications. *ACM Trans. Multimedia Comput. Commun. Appl.*, 3(2):12, 2007.

36. H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr, and J. Peterson. Presence Information Data Format (PIDF). RFC 3863, IETF, 2004.

37. Sun Microsystems. The JAIN SIP API specification v1.1. Technical report, Sun Microsystems, June 2003.

38. S. Tsang, S. Moyer, and D. Marples. SIP extensions for communicating with networked appliances. Internet draft, IETF, May 2000.

39. A. Vaha-Sipila. URLs for telephone calls. RFC 2806, IETF, 2000.

40. X. Wu and H. Schulzrinne. Programmable end system services using SIP. In *Proceedings of The IEEE International Conference on Communications 2002*. IEEE, 2003.