CIAO: A Component Model and its OSGi Framework for Dynamically Adaptable Telephony Applications

Gilles Vanwormhoudt, Areski Flissi

CIAO: A Component Model and its OSGi Framework for Dynamically Adaptable Telephony Applications

Gilles Vanwormhoudt
Mines-Telecom Institute
LIFL, UMR CNRS 8022
France - 59655 Villeneuve d'Ascq cedex
vanwormhoudt@telecom-lille1.eu

Areski Flissi
LIFL, UMR CNRS 8022
Lille University
France - 59655 Villeneuve d'Ascq cedex
Areski.Flissi@lifl.fr

ABSTRACT

In recent years, thanks to new IP protocols like SIP, telephony applications and services have evolved to offer and combine a variety of communication forms including presence status, instant messaging and videoconference. As a result, advanced telephony applications now consist of distributed entities that are involved into multiple heterogeneous, stateful and long-running interactions (sessions). This evolution complicated significantly applications development and calls for more effective solutions. In this paper, we explore the adoption of components for addressing this issue, focusing specifically on the management and coordination of the numerous and various sessions occurring in such applications. The paper presents CIAO, a domain-specific and hierarchical component model for SIP applications. CIAO combines three kinds of component that are Actor, SessionPart and Role and manage them dynamically in accordance with real SIP sessions. By using these features, we are able to break the complexity of SIP entities and provide flexibility for their development. CIAO is implemented above OSGi to experiment the building of concrete SIP applications and enable their dynamic adaptation.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Software Architecture—*domain-specific architectures, adaptable architectures*

Keywords

Component model, telephony, sessions, dynamism, SIP, OSGi

1. INTRODUCTION

Stimulated by the ubiquity of IP networks and the existence of new protocols like SIP (Session Initiation Protocol), telephony services have endorsed significant changes in recent years. They are now able to offer and combine a variety of communication forms including presence status, instant messaging and videoconference while managing other concerns like mobility, localization, security, etc. This situation changes and complicates significantly the development of telephony applications that mainly consist of distributed entities involved into multiple heterogeneous, stateful and long-running interactions (sessions). Designing such applications not only requires programming expertise but it also requires familiarity with the handling of complex sessions and interactions. Owning to the specific skills required for these applications, the development process becomes expensive in terms of time and resources, calling for more effective solutions. Regarding this problem, a component-based approach to develop telephony applications becomes an attractive feature as it promises several advantages in terms of design flexibility and evolution [4]. For the flexibility in design, components can help to decompose the complex behaviour required for sessions management into prepackaged and reusable units that can be further assembled for specific applications. For evolution, recent component models designed to deal with dynamic changes like OSGi [10], can help to accommodate new features or change behaviours after the deployment stage.

Main existing works on component-based development for SIP applications are [7, 5, 11] and mainly target server-side entities. In all these works, the multiplicity and dynamicity of sessions occurring in advanced telephony applications are not or poorly addressed by means of components. Not only this causes difficulties when the base of sessions to manage and coordinate is various and complex, it also hampers the modularity and adaptability of SIP applications relatively to sessions.

In this paper, we propose CIAO (Component for sIp ApplicatiOns), a domain-specific component model for developing any kind of SIP entities, either client, server or both. This component model is hierarchical and relies on three specific component types that are Actor, SessionPart and Role. Actor components are top level components that represent distributed SIP entities. They contain SessionPart components that provide actor behaviour related to particular kinds of session. SessionParts are themselves container for Role components which encapsulate one of the behaviour played by an actor when it participates to a particular session. Besides these specific components, the model is characterized by powerful mechanisms for components interaction and coordination and by a management of components which is dynamic and maps real SIP sessions. According to this management, SessionParts and Roles created for an Actor depend automatically on the session it participates

in at runtime. Advantages provided by this model are to achieve a better modularization between session-dependent parts of entities, to simplify the reasoning regarding sessions dynamicity and to improve the capacities for constructing SIP entities from reusable components.

In addition to this model, we also describe an implementation of the CIAO model as component framework over the OSGi platform. This implementation leverages the capacity of OSGi components for software adaptation. The resulting framework allows to adapt components of SIP entities at runtime with minimal interruptions. This feature are illustrated and discussed through characteristic examples of adaptation.

The rest of the paper is organized as follows. Section 2 gives background on SIP and the issues underlying the development of advanced SIP applications. It is followed by the description of CIAO component model in section 3. The implementation of CIAO components over OSGi is presented in section 4. Section 5 reports experiments, notably on dynamic adaptations. Section 6 comments on related works prior to conclude in section 7.

2. BACKGROUND ON SIP AND ISSUES

SIP is an application-layered signaling protocol standardized by the IETF that supports applications ranging from simple VoIP routing or instant messaging to sophisticated multimedia sessions involving multiple parties with presence management. For these applications, SIP provides a rich range of communication forms. Communications can be stateless for simple messages exchange, session-based to exchange messages over a period of time or event-based to propagate information like state-change of an entity. One main benefit of SIP is that it enables mixing all its communication forms in order to design advanced telephony applications.

Figure 1 shows the architecture of an advanced telephony application based on SIP to provide presence-based redirection[1] of invitation to a dialog combining voice and text. The architecture is composed of three SIP entities having distinct capacities and interacting in multiple ways. In the scenario depicted by this figure, Alice updates its status because she leaves its office for a meeting (1,2). Then, Bob calls Alice during her meeting and he receives a `BUSY` response (3,4). When Alice becomes available (5,6), a new invitation from Bob results in a redirect response including Alice's address to initiate a dialog (7,8). Thanks to the returned address, Bob can directly contact Alice to establish a successful dialog (9 to 18).

As illustrated by the previous example, lots of message exchanges between the involved SIP entities may be required to achieve the intended functionality. This profusion of messages complicates significantly the development of SIP entities. In [12], we identified three typical sources of complexity:

1) Complex messages flow within a session : Within a session, SIP entities exchange and handle SIP messages at each end. Several difficulties must be faced when handling these messages:

– Messages can be received at any time from the network. As a result, SIP entities must be prepared to react to any

[1]In SIP, redirection consists in directing the client to contact an alternate address.

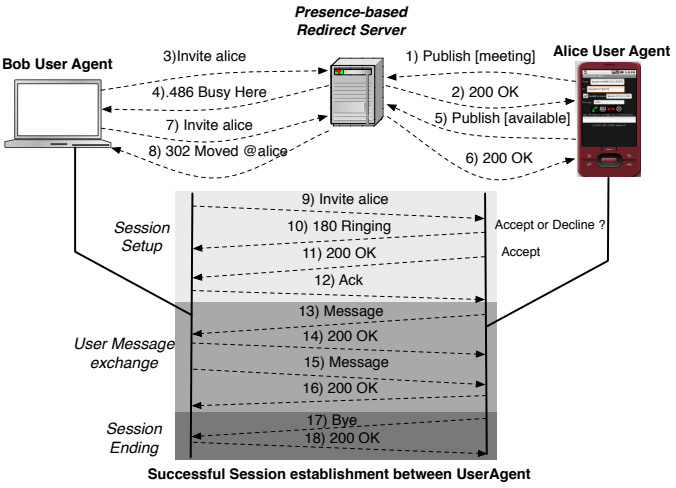

Figure 1: Example of Presence-based Redirection

received messages, including while others are processed;

– Interpretation of messages in the flow is generally state-dependent (see `OK` messages in the example). This requires the entities maintain a session state according to the exchanged messages for distinguishing between these interpretations;

– Messages flow of a session may contain and interleave messages that are related to distinct concerns as illustrated by colored groups of messages in the lower part of Figure 1.

– Message flow can be inverted during the session (see `BYE` for example). This implies that entities must be able to provide behaviours for handling session flow in both directions and to act both as requestor and responder.

Because of these difficulties, the behaviour of entities handling some parts of the session flow like the previous one is usually not easy to design.

2) Multi-branches session : A SIP session is not always restricted to a peer-to-peer conversation. There are some classical calling or presence scenarios which involve more than two peers in the same session. In these scenarios, one or several SIP entities generally act as facilitator between several participants to the session. For such entities, this entails to handle more that one conversation within the same session and to coordinate all the conversations in order to serve the communication partners properly. In our example, we initially assume a redirect behaviour for the server but we can easily imagine changing this behaviour to a proxy-like one in order to extend the status of a user according to its participation in an existing dialog[2]. Compared to the previous behaviour, adopting this change requires that the server manages and coordinates two communication paths within a session : one to the callee and one to the caller, switching back and forth being a client and a server at the same time. When a SIP entity is involved in more than one conversation within a session, the complexity for describing its behaviour is inherently increased.

3) Multi-sessions management : The need to handle multiple sessions is another situation that makes the development of SIP applications and their entities complicated. Two

[2]Proxy behaviour in SIP consists in relaying each request and response between user agents of a session.

cases may be distinguished for multi-sessions management. The first case is the one where the sessions managed by a SIP entity have the same type and typically occur concurrently. We generally encounter this case when designing SIP servers. In our example, the redirect server is an illustration of this case as it must be able to manage calling session coming from multiple requestors. Here, the difficulty is that multiple session states must be maintained separately and concurrently. The second case is the one where each session has a different type. This case can be found particularly in the development of rich user agents and rich servers. In our example, Alice's user agent illustrates this case as it must be able to manage several sessions related to registration, incoming invitation and status modification. For this case, an additional difficulty besides maintaining multiple states is that each session may require the handling of distinct set of messages resulting in a significant amount of messages to handle. Note that a combination of the two cases may sometimes be required for the design of a SIP entity as it is illustrated by the redirect server. For SIP entities, supporting multiple sessions makes their design more complicated because they must include characteristics for many states and behaviours.

As we can see, SIP applications have specific issues related to sessions management that make their development complex and lead to intricate structure of behaviour for involved SIP entities. Therefore, an adequate component model targeting these applications and their entities should provide abstractions to solve all these issues. In the next section, we present CIAO, a component model that includes appropriate abstractions for that purpose.

3. CIAO COMPONENT MODEL

CIAO only deals with the part of applications that is related to SIP but it is sufficiently general to address the needs of any SIP entity, that is client, server or both. For other parts of applications like UI or database, we assume that they are addressed by more classical components. CIAO components can be assembled and interact with these components to form a complete application.

3.1 Kinds of component

CIAO has three kinds of component which are Actor, SessionPart and Role. The first two are composite components while the last one is atomic. In our model, a composite encapsulates subcomponents as well as the logic for controlling them. An atomic component encapsulates computation logic. Each kind of component has a specific purpose in the model :

– *Actors* are composite components at the top-level. These components are so called because they represent SIP entities from the network view. They directly exchange SIP messages with other entities in the network. Actors are typically involved into one or several SIP sessions that may have the same nature or being different and exist concurrently. To handle these sessions, Actors encapsulate subcomponents which are SessionParts. Actors are also responsible of routing SIP messages from and to its SessionParts and coordinating them.

– *SessionParts* are composite components encapsulated by actors. These components aim to provide the behaviour of actors related to a specific session type. A SessionPart encapsulates Role components to provide this behaviour. Like Actors, SessionParts are also in charge of routing SIP messages from and to its Roles and coordinating them.

– *Role* are components at the lower level. These components are atomic and provides one of the actor behaviour involved in a session. This behaviour consists in handling incoming and outgoing SIP messages while realizing the relevant logic.

In CIAO, the Role kind introduced above is further refined in three sub-kinds to specify their capacities in terms of received and sent messages and help the analysis and correlation of communicating roles inside and between actors :
– *Client role* represents an asymmetric role type that sends requests and receives related responses.
– *Server role* represents an asymmetric role that receives request and sends related responses.
– *ClientServer role* represents a symmetric role that sends and receives both requests and responses.

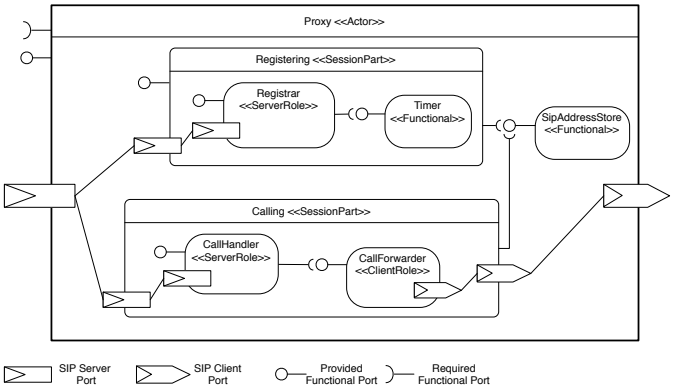

Figure 2: Example of CIAO Components corresponding to Proxy

Figure 2 shows a concrete Actor corresponding to Proxy entities and illustrates the hierarchical structure of such components. This Actor is composed of two SessionParts, one dedicated to sessions for user address registering, the other to sessions for user calling. These two SessionParts include themselves Roles for providing the respective behaviour. In the *Registering* SessionPart, the *Registrar* role provides the behaviour for registering/unregistered user agent address. In the *Calling* SessionPart, the *CallHandler* role handles the communication with the calling user agent and the *CallForwarder* role handles the communication with registered called user agents. *Timer* and *SipAddressStore* are not CIAO components but are functional ones. They are included in the Proxy architecture to provide both timing and usage address storage functionalities. The first one is needed to record the time period of each user address registration, so is contained in *Registering* SessionPart. The second one allows to store user address during registration and retrieve them during calling, so is contained in the Proxy Actor to be shared by its two SessionParts.

Using CIAO, SIP architectures that generally involve several SIP entities, can be built from a set of related actors[3]

[3]To preserve the interoperability with existing SIP systems, CIAO does not enforce that all entities in such architecture be necessarily based on CIAO. Indeed, Actors can interop-

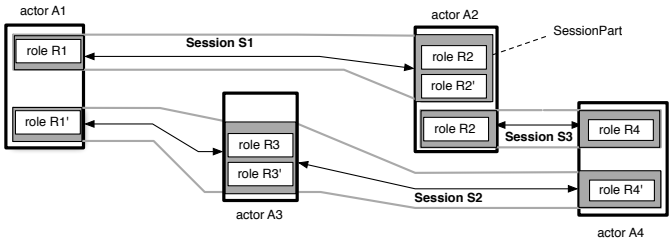

Figure 3: Illustration of Actors, SessionParts and Roles

where each actor communicates with other actors within sessions, playing dual and consistent roles. Figure 3 illustrates such an architecture and show capacities of the model. In this architecture, related actors A1 and A2 participate to a SIP session S1. Within this session, actor A1 plays a single role R1 and communicates with A2 that plays dual roles R2 and R2'. Figure 3 also shows a second SIP session S2 independent from S1 that involves actors A3 and A4 but also A1, already engaged in S1. For this session, we may observe that actor A1 plays a role R1' distinct from R1 played in S1. Actor A3 plays two distinct roles to communicate with A1 and A4 actors.

The combined use of Actor, SessionPart and Role components proposed by our model allows to deal with the issues identified above. The first issue can be managed by defining multiple roles for a particular session (e.g. A2), each role dealing with one session concern. The capacity for an actor to play multiple roles for distinct communication path (e.g. A3) of a session may be exploited to solve the second issue. Concerning the last issue, the solution is provided by the ability of an actor to participate in multiple sessions with separate roles (e.g. A2 and A4).

3.2 Component instanciation and life-cycle

Whereas some models treat component as configured instances, CIAO differentiates between component types and instances. The principle is that all CIAO components are type-level component and these components are instantiated at runtime to form the architecture of concrete Actors. For an actor, its architecture is always a tree of component instances where they can be multiple instances from a given inner component type. Figure 4 gives an example showing the instance tree of a Proxy actor currently engaged in two calling SIP sessions (represented by SessionParts with Call-Id @345 and Call-Id @127) and one registering SIP session (represented by SessionPart with Call-Id @56).

One key feature of CIAO is that the inner architecture of an actor is not fixed at instantiation-time but changes dynamically during its lifetime. Changes performed to an actor architecture aim to reflect its participation to real SIP sessions and provide all the states and behaviours required for these sessions. These changes are as follows: each time an actor is involved in a new SIP session, a corresponding SessionPart is created and added to the actor architecture. Conversely, each time a SIP session is terminated, the corresponding SessionPart and its roles are removed from the actor architecture.

The instantiation of SessionPart can be done explicitly on demand by means of a new-like construct or occurs implicitly on the basis of received request. For a receiving request, the decision to create a SessionPart instance depends if there already exists a session part matching the session identifier (Call-Id header) included in the request. If none exists, some rules attached to the Actor for SessionParts are evaluated to create a matching instance. A SessionPart may be instantiated more than once per actor. This corresponds to the situation where an actor participates to several sessions of the same type during its lifetime.

At the level of SessionPart, similar changes are performed for the inner roles. These changes consist in adding and removing roles to advance the actor state and behaviour inside the SIP session. Like session part instantiation, the creation of roles attached to a session can be done in two ways: either explicitly by means of a new-like construct or implicitly on the basis of a received request. In the latter case, rules attached to the SessionPart are evaluated to determine if a corresponding role must be created.

SessionPart and Role components have explicit states that are automatically setup during their lifecycle. SessionPart can be in one of the following session-related states: *Idle* (initial state), *Bound* (after binding with real session), *Started*, *Ended*, *Unbound* (terminal state). For a Role, there are three states, namely: *Idle*, *Active*, *Invalid*. All these states are accessible through a common interface provided by components so that they can be used to realize the actor behaviour.

3.3 Component interaction

CIAO components have features to interact with other components both at functional and SIP levels.

Functional Ports

For functional interactions, components may have functional ports which are access points to component functionality with a set of operations. In CIAO, functional ports can be of two kinds: provided and required. A provided functional port corresponds to a functionality provided by the component whereas a required functional port corresponds to a functionality required by the component. Interaction between components is only possible if required functional ports are bound to provided ports. Connection between ports can be dynamically set and unset at runtime.

By means of functional ports, a component can interact with other components to use or provide its functionality. For example, a Role (resp. SessionPart) component can use the functionality of its owning SessionPart (resp. Actor) to access a shared state or activate a common behaviour. Conversely, a composite like an Actor (resp. SessionPart) may consume functionality provided by its inner SessionParts (resp. Roles).

SIP Ports

For SIP interactions, CIAO introduces SIP ports and SIP messages routing among the hierarchy of components. SIP ports are bidirectional access points to the SIP network for components. They are used for sending and receiving SIP messages. Actor and SessionPart may have several SIP ports but roles only have a single one (see Figure 2). A SIP port is created dynamically each time a communication path is established for a role, that is an initial SIP request is sent to or received from the network. There are three kinds of SIP port in relation to the kinds of role defined in section 3.1 :

erate transparently with existing SIP entities.

– *Client SIP Port* enables sending of SIP requests and receipt of SIP responses

– *Server SIP Port* enables receipt of SIP requests and sending of SIP responses

– *Client/Server SIP Port* enables both sending and receipt of SIP requests and responses.

In CIAO, the source and destination of SIP messages exchanged with the network are always Roles. More precisely, SIP messages are routed from and to the network following a unique path in the Actor's component hierarchy. This path starts from the root Actor to the appropriate role if the message is a received message. Conversely, the path starts from the emitting role to the root Actor if the message is sent. Along the path, the message is delivered to components through their SIP ports. This enables components to adapt the routing and eventually to perform computations in relation with the delivered messages.

When a component receives a message via one of its SIP ports, it is responsible of selecting the next component along the path and passing the message to this component. A routing component may use various strategies to make this selection. Currently, we use default strategies for sent and received messages. For sent messages, the strategy consists simply in selecting the outer component. According to this strategy, a message sent from a Role is submitted to its containing SessionPart then to the containing Actor and finally to the network. For received messages, the strategy for selecting the right component depends on the type of the message and the session it belongs. With this strategy, a message received from the network is first delivered to the root Actor. Then, the message is forwarded by the Actor to the SessionPart matching the message session identifier and type, eventually a new one if none matches. After that, the process continues with the selection of a matching role. This selection is done by the SessionPart on the basis of the message type and optionally some current state using its rules. At last, the message is forwarded to the selected Role so that it can realize the relevant logic. This is achieved by triggering one of its message handlers. Message handlers are special operations provided by Role to handle an incoming SIP request or response. These operations have a specific signature that matches the type of particular SIP message.

Figure 4 illustrates the receipt of an `INVITE` initial request by a Proxy Actor and its routing through the corresponding SessionPart up to the *CallHandler* role. This routing terminates by executing the *onInvite()* request handler provided by the role. The response emitted by the role is `301 Moved Permanently` indicating that the user is not available through the given SIP address but via another address provided in the response. As we can observe, its routing follows the converse way, through the same SIP Ports.

3.4 Component coordination

In some scenarios, the roles that an actor plays during a session or the multiple sessions it is involved in require coordination within the actor itself. Generally, this is due to the fact that some actions provided by some entities depend on actions provided by other ones. To fulfill these coordination needs, our component model offers two coordination mechanisms, one dedicated to SessionParts and one targeting Roles. These two mechanisms are grounded on the same principle : a composite instance provides the coordinating behaviour for its inner component instances. These principle aims to provide two benefits. Firstly, it permits to avoid explicit coupling between subcomponents for coordination purposes, improving therefore their reuse. Secondly, it permits to locate the coordinating behaviour in one place, facilitating its understanding and maintenance. In the following, we detail these two mechanisms which are both event-based.

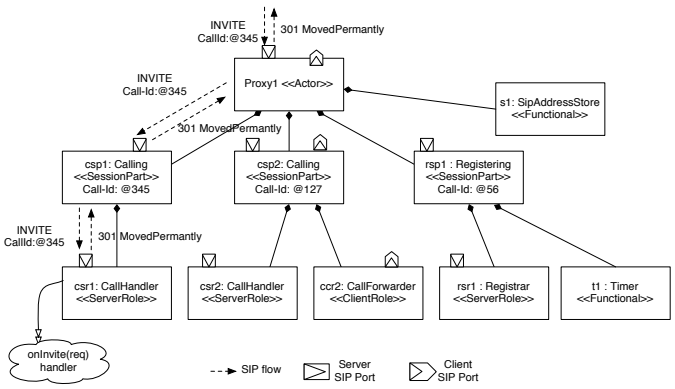

Figure 4: SIP Message routing among components

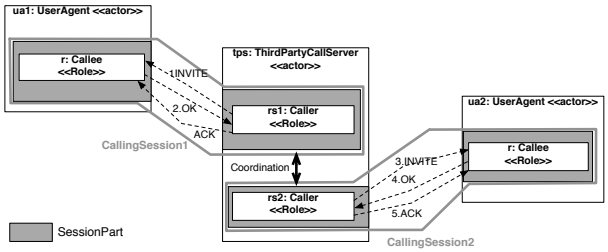

Figure 5: Third-party call as example of sessions coordination

SessionPart Coordination

The need to coordinate the behaviour of an actor across several sessions can be illustrated using the case of third-party call entities which are controllers setting up and managing a phone call between two or more other parties [4].

Figure 5 illustrates the basic idea behind third-party call entities using an example which only connects two users in a call. We can see that the server is represented by an actor which is involved into two calling sessions with user-agent actors. For these sessions, coordinating behaviour are required from the previous actor. The second calling session must be started only when the initial one has been acknowledged. It consists for the third-party entity in sending a second `INVITE` after the first *INVITE* was confirmed by a *200 OK*. Furthermore, when the second session is confirmed, the two sessions must be acknowledged by sending a ACK message to each user agent.

Within an Actor, the behaviour for coordinating its SessionPart is defined by means of callback operations related to specific SessionPart events. These coordinating callback operations are attached to the Actor component and they are automatically triggered at runtime each time the corresponding event related to a SessionPart occurs. The struc-

[4]Third party call entities are often used for operator services, click-to-dial and conferencing services.

Session event	Coordinating operation	Provided Parameters
Session creation	onCreatedSession	New session reference
Session deletion	onDeletedSession	Deleted session reference
Session start	onStartedSession	Started session reference
Session end	onEndedSession	Ended session reference
Session custom event	onCustomEventSession	Session reference, Objet reference for payload

Table 1: Kind of session events and their corresponding operations

ture of coordinating operations is similar to normal operations. Like an operation, it owns a body containing the set of coordinating actions and has provided parameters.

Table 1 gives the set of events dedicated to sessions coordination within an actor. They are related to the life-cycle of a session as well as its state from the actor point of view. For each event, this table also indicates the corresponding operations and provided parameters.

For this set of events, it is important to precise that only events related to session life-cycle are triggered automatically (creation, deletion). Those related to session state (started, ended) must be triggered explicitly by the developer. The reason why such event needs to be handled explicitly comes from the fact that determining the start or the end of a session is generally application-specific. Using the previous facilities, the behaviour for coordinating the two sessions as described can be done by defining an *onStartSession* operation for the Third-Party call Actor and by using provided session parameters to distinguish between second `INVITE` sending case and acknowledgments case.

Role Coordination

To illustrate the need of roles coordination, let us consider the case of a forking operation handled by SIP proxies. In a proxy, this operation occurs when it receives an invitation for a user which has registered more than one user agent. In that case, the proxy relays the invitation to all registered user agents by creating separate branches. When one registered user agent accepts the invitation, other invitations must be cancelled. Using CIAO, the behaviour for a forking operation may be achieved by coordinating *CallHandler* and *CallForward* roles attached to the running *CallingSession* SessionPart as illustrated in Figure 6. To achieve the forking operation, coordinating actions are required to create new *CallHandler* on the receipt of a request by the *CallHandler* and to activate or deactivate them depending on which user agent is the first to respond.

To deal with situations like the previous one, our component model includes a mechanism dedicated to the coordination of roles within a session. This mechanism follows the same design principles than the one designed for coordinating SessionParts of an Actor : the SessionPart acts as the coordinator of its role and relies on special operations to define the coordinating behaviour in term of actions.

Table 2 presents the set of role events that may captured in a SessionPart through the definition of coordinating operations. These events can be subdivided in two groups: one concerns the life-cycle of a role during a session while the other deals with message handling performed by a particular role.

All these operations allow to have a fine and complete control over the role events related to a session. Thanks to

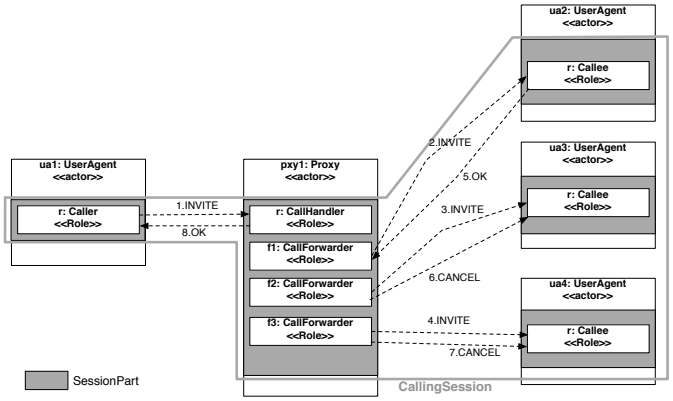

Figure 6: Forking as example of roles coordination

Role event	Coordinating operation	Provided Parameters
Role creation	onCreatedRole	New role reference
Role deletion	onDeletedRole	Deleted role reference
Role binding	onBoundRole	Bound role
Request reception by a role	onRequestReceived	Role reference, Request reference
Response reception by a role	onResponseReceived	Role reference, Response reference
Request sending by a role	onRequestSent	Role reference, Request reference
Response sending by a role	onResponseReceived	Role reference, Response reference

Table 2: Kind of role events and coordinating operations

such control, many scenario for coordinating roles, including the one discussed previously for proxy servers may be managed seamlessly. For this scenario, the expected behaviour can be achieved by defining two coordinating operations in the *CallingSession* SessionPart: an *onRequestReceived* operation to detect `INVITE` request handled by the *CallHandler* role and relay such request using appropriate *CallForwarder* roles ; an *onResponseReceived* operation to cancel unconfirmed branch of *CallForwarder* roles.

4. CIAO ABOVE OSGI

To experiment our component model for the development of real applications and validate its benefits, the proposed components and mechanisms must be implemented concretely. To this end, we have developed an implementation of CIAO on top of the OSGi component framework. The choice of OSGi were driven by the need to exploit its capacities for dynamic adaptation of applications.

4.1 Mapping CIAO components to Declarative Services

We use Declarative Services as a basis for implementing all the kinds of component included in CIAO. Declarative Services (DS) is a part of the R4 OSGi[10] specification that extends the service component model provided by OSGi Bundles. This extension offers a declarative model (based on XML descriptor) for managing multiple components within each OSGi bundle and for automatically managing the component lifecycle in response to bound services coming and going. In our implementation, each CIAO component of an application is implemented as a DS component and functional ports provided by CIAO components are specified as

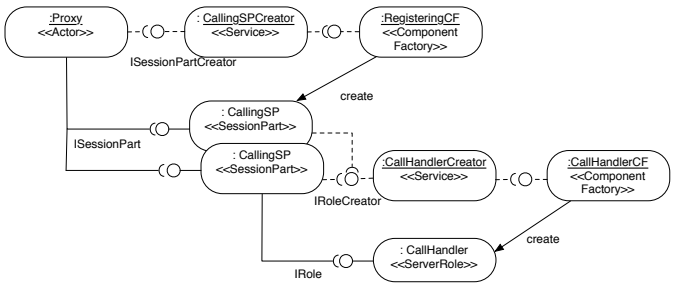

Figure 7: Architecture of the Proxy implemented with DS Components

separate provided services of the corresponding DS components. SIP ports are handled by the CIAO runtime.

By default, Declarative Services automatically creates instance of component when their service dependencies declared statically inside its descriptor are satisfied. This default behaviour is not well-suited to the dynamic nature of the CIAO component model. In CIAO, components, mainly internal ones, need to be created dynamically depending on SIP messages flow. Furthermore, multiple instances of a CIAO component may also be needed when handling multiple sessions at the same time. Owing to these specific needs, it was necessary to elaborate a specific management of component instances over DS.

To solve the previous problem, we rely on the Factory Component facility provided by DS and specific helper components added to the set of DS components derived from CIAO ones. A factory component is a DS component that can be used by other DS components to create and dispose instances of particular DS components. Such factory component are implicitly created if required. They are also automatically registered and can be referenced by other DS component to manufacture component instances. In our case, we declare a distinct factory component for each CIAO component included in an application.

In addition to factory components, we also introduce a correlated helper service component (called Creator) for each SessionPart and Role. The role of Creator components is to provide the logic for determining if a component instance have to be created for the handling of a particular SIP message. These helper components make directly use of the corresponding factory. They are also declared to provide a service that is referenced by the outer component and some properties for determining the appropriate component to instantiate on specific messages.

Figure 7 shows the architecture of DS components corresponding to a Proxy actor currently engaged in one registering session and two calling sessions. In this figure, components that are declared statically in descriptor files have underlined names. Others are instantiated dynamically for matching the sessions and supporting the intended behaviour. Dependencies that are automatically managed by the DS runtime are represented with dotted line. As we can observe, Actor and SessionPart components maintain a set of references to Creator components so that they can collaborate with them for instance creation during messages routing.

The code in Figure 8 gives several descriptors used in the example. They correspond to the declarations of a Proxy Actor, the *CallingSessionPart* component with its factory and the corresponding *Creator* component.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="ProxyActor1">
   <implementation class="ciao.examples.proxy.ProxyActor"/>
    <service>
      <provide interface="ciao.examples.proxy.IProxyActor"/>
      <provide interface="ciao.framework.component.IActor"/>
   </service>
   <reference bind="addCreator" cardinality="1..n"
interface="ciao.framework.component.ISessionPartCreator"
name="sessionPartCreator" policy="dynamic" target="(sip.actor=ProxyActor)"
unbind="removeCreator"/>
</scr:component>

<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="CallingSPCreator">
   <implementation class="ciao.examples.proxy.CallingSPCreator"/>
   <service>
      <provide interface="ciao.examples.proxy.ISessionPartCreator"/>
</service>
   <reference bind="setFactory" unbind="unsetFactory"
interface="org.osgi.service.component.ComponentFactory"
name="SessionPartFactory" target="(component.factory=CallingSP.factory)" />
   <property name="sip.actor" type="String" value="ProxyActor"/>
   <property name="sip.request" type="String" value="INVITE:BYE"/>
</scr:component>

<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
factory="CallingSP.factory" name="CallingSessionPart">
   <implementation class="ciao.examples.proxy.CallingSessionPart"/>
   <service>
      <provide  interface="ciao.framework.component.ISessionPart"/>
   </service>
   <reference bind="addCreator" cardinality="1..n"
interface="ciao.framework.component.IRoleCreator" name="roleCreator"
   policy="dynamic" target="(sip.session=CallingSessionPart)"
unbind="removeCreator"/>
</scr:component>
```

Figure 8: Example of descriptors for a Proxy Actor

The previous architectural principles enables to benefit from the capacities of Declarative Services for all CIAO components. First, the modularity provided by DS allows to decompose an actor into multiple ways for their deployment in a running OSGi platform. For instance, it is possible to have a bundle containing an actor and all its inner components or having separate bundles for each SessionPart used by an actor. Secondly, thanks to the DS capacities for dynamic updates of components, all CIAO components can be installed, updated and started with minimal disruption to the running SIP entity. See section 5.2 for example of SessionPart and Role added dynamically.

4.2 Runtime and annotation frameworks

To enable the development and execution of SIP applications in accordance with our component model, we have implemented a runtime framework which is build above OSGi and JAIN-SIP frameworks. This framework defines a set of classes that provides certain core data structures and functionalities for managing the execution of CIAO components. Figure 9 shows the main classes of this framework.

Some CIAO runtime classes extend themselves the standardized JAIN-SIP API. To build an application, developers have to write Actor, SessionPart, Role, and Creator components by specializing respective classes and write XML descriptors as described previously.

Using CIAO implementation constitutes an improvement for the development of SIP applications but specializing the framework still requires deep knowledge of underlying programming rules. To simplify and speed up the coding task, we provide a Java annotations framework too. In our case,

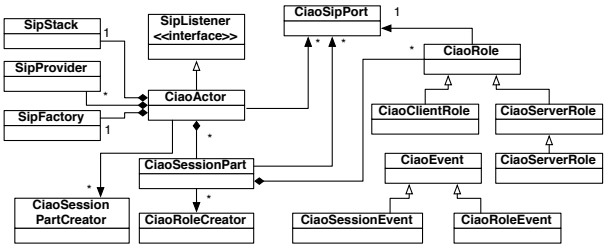

Figure 9: Main classes of CIAO Runtime Framework

we use annotations as a way of conveniently mapping the concepts of CIAO component model to user-defined classes or methods and automatically generate both the classes specializing the runtime framework and the descriptors.

A class with the `@actor` annotation will inherit from the `CiaoActor` class of the framework, whereas a class annotated with `@sessionPart` will inherit from *CiaoSessionPart* class. The three subkinds of role provided by the model are declined in corresponding annotations for classes: `@clientRole`, `@serverRole`, `@clientserverRole`.

An actor declares its sessions part with `@useSessionPart`(*type*, *method*, *condition*) annotation on its class. The mandatory *type* attribute determines the SessionPart class that has to be imported and activated. The mandatory *method* attribute provides the type of SIP request method (*e.g.* `INVITE` message) that triggers the activation of the SessionPart. The optional *condition* attribute is a string that refers to the name of a boolean method that describes some particular conditions for the activation of the SessionPart. The principle is similar for declaring roles of a SessionPart, thanks to `@useRole` annotations, except that it is used with a SessionPart class. Figure 10 gives an example of annotated code corresponding to some CIAO components used in the definition of *PresenceEnableUserAgent* actor.

```
@clientserverRole  // Declaration of role class
public class Callee {
  public void onInvite (Request req, ServerTransaction tx) { ... }
  public void onACK (Request req, ServerTransaction tx) { ... }
  public void sendBye () {...}
  ...
}
```

```
@sessionPart // Declaration of session part
@useRole(type=Callee.class,method="INVITE")
@useRole(type=Caller.class)
public class CallingSessionPartUA {
  public void onRequestReceive(Role r, Request req) {
    if (r.hasType(Callee.class) && isINVITE(req)) {
      from = req.fromHeader().getAddress();
      actor.getCallerHistory().put(from);
   } ...
}
```

```
@actor // Declaration of actor class
@useSessionPart(type=RegisteringSessionPartPUA.class)
@useSessionPart(type=PublishingSessionPartPUA.class)
@useSessionPart(type=CallingSessionPartPUA.class,method="INVITE",
condition="isUserAvailable"))
public class PresenceEnabledUA implements ICallControl,
IPresenceControl, ICallHistoryMngt {
   String contactName;
   List<SipAddress> callerHistory;
  public void clearHistory() {...}
  public void addHistory(SipAddress adr) {...}
  public void initiateCall(String sipAddress) {...}
  public void endCall() {...}
  public boolean isUserAvailable() { ... }
  public void onStartedSession(SessionPart session){
     getRole(PublishSession,PresencePublisher).
       setNewState(State.unavailable); }
...
}
```

Figure 10: Example of annotated code for components of presence-enabled UserAgent

5. EXPERIMENTATION

In this section, we report some experiments about the building of SIP entities using CIAO, their adaptation relatively to sessions and a capitalization of these experiments for providing a first set of reusable components.

5.1 Developed SIP entities

In order to validate the generality of our CIAO component model and the relevance of its structuring, we have developed four SIP entities[5] that target distinct SIP areas (telephony, presence) and map to classical categories of SIP agents (end-point, intermediary entity, client and/or server). In the following, we list these entities and indicate the issue(s) it was necessary to solve for each:

– *SipPhone* : The development of this entity required to manage two independent session types, one dedicated to call management and one dedicated to user address registration. Other addressed issues were mainly related to the complexity of flow during a call as it relies on state-dependent, asynchronous and inverted exchange of messages.

– *Proxy* : Through the development of this entity, we addressed the issue of handling multiple sessions of two distinct types (recording, calling) at the same time with the particular issue that the calling session is multi-branches and have a quite complex flow. For managing this issue, we used the mechanism for coordinating roles of a session as described in section 3.4.

– *Third-party Call Server* : In terms of issues, the design of this entity required the ability to initiate and manage as many calling sessions as parties as well as the ability to orchestrate all these sessions.

– *Presence Server* : Like a proxy, a presence server is an intermediary entity that accepts, stores and distributes SIP presence information from a source entity to subscribing entities. This kind of entity needs to manage several sets of sessions at the same time with the difficulty that the sessions in a set have distinct types (publish and subscribe-notify), map to distinct participants and need to be coordinated.

The following table presents the structuring of developed entities, so Actor components. For each Actor, we give its SessionPart components and its Role components[6]. For instance, the row for *Proxy Server* actor can be interpreted as follow: this actor has two *CallingSessionSvr* and *RegisterSessionSvr* SessionParts and play one *CallHandler* server role and one *CallForwarder* client role for the former and one *RegisterAcceptor* server role for the latter.

These experiments confirmed the generality of our proposal and the relevance of its structuring. They also highlighted that the model offer several advantages. First, it raises the level of abstraction as the developer can think about the behaviour of its actor at a higher level thanks

[5]Three of these entities are detailled in the first coauthor book on SIP Telephony in Java [2]

[6]The table also indicates if the actors participate to one or several sessions at the same time (written respectively [1] or [*] after the session type name).

Table 3: CIAO Components of developed entities

Actors	SessionParts [1|*]	Roles [owner SessionPart,c|s|cs]
SipPhone	CallingSessionClt[1], ProxyCallingSessionClt[1], RegisterSessionClt[1]	Caller[(1,2),cs], Callee[(1,2),cs], RegRequestor[3,c]
Proxy	ProxyCallingSessionSvr[*], RegisterSessionSvr[*]	CallHandler[1,s], CallForwarder[1,c], RegAcceptor[2,s]
3rd Party Call Server	CallingSessionClt[*]	Caller[1,cs]
Presence Server	PresencePublishSessionSrv[*], PresenceSubscribeSessionSrv[*]	PresencePublisRequestor[1,cs], PresenceNotifier[2,c], PresenceSubscriptionRecorder[2,s]

to SessionPart and Role. Second, the encapsulation of behaviour and state into multiple SessionPart and Role components with delimited scope contributes to increase the modularity of actors, making the maintenance and evolution of SIP entities easier. Third, automatic selection of SessionParts and Roles as well as automatic messages routing to roles allow to reduce the coding effort since number of controls and extra-state to ensure execution of the appropriate behaviour is minimized. Finally, our approach gives the ability to capture some recurrent behaviours into reusable components (see section 5.3).

5.2 Adaptation case studies

In addition to experiment CIAO for the development of SIP entities, we also tested the capacities offered by our implementation over OSGI for adapting SIP applications after their deployment, including at runtime. In our case, we tested the following adaptations using scenarios applied to the developed SIP entities:

– *Role addition to SessionPart* : This was tested by extending SipPhone Actor with a new functionality to exchange instant text message during a call. To get this extension, we added a *MessageHandler* client/server role to its *CallingSessionPartClt* so that `MESSAGE` requests and their responses can be exchanged during the session.

– *Role replacement in SessionPart* : This was tested by modifying the SipPhone so that it can handle redirected calls in addition to direct calls. For that purpose, we replaced the initial *Caller* role of *CallingSessionPartClt* with a richer *Caller* role.

– *Addition of SessionPart to Actor* : This was tested by adding the functionality of presence publishing and subscribing to ProxyServer Actor in order to get enhanced proxy. For this adaptation, we simply added the two SessionParts defined for the presence server (*PresencePublishSessionSrv* and *PresenceSubscribeSessionSrv*) to the deployed proxy. This required to change the target Actor in the descriptor of these components.

– *Replacement of SessionPart to Actor* : This was tested by changing the basic registration functionality of SipPhone and Proxy actors to secured registration with authentication. To apply these changes, we replaced each registering SessionPart contained in these entities with new ones encapsulating roles capable of handling flow with authentication SIP messages.

All the previous adaptations have been achieved by packaging required components (CIAO components and the related Creator components) in OSGI bundles with the appropriate descriptors and by deploying the bundles in an OSGI platform hosting the application to adapt. Thanks to the use of Declarative Services, CIAO components currently instantiated in the application are automatically notified of the installation of Creator components and can link to them. However, in case of replacement, such components must not be used immediately to create instance as they may be running sessions relying on replaced components. In this work, the replacement takes place when all related sessions are terminated. This strategy is intended to ensure a proper state and behaviour for these sessions until they finish. In particular, we exploit the states from the life-cycle of component (see section 3.2) to determine the correct moment of replacement. After replacement, bundles containing replaced components can be removed from the platform.

During these adaptation experiments, we meet some limitations related to the coordination of components. Indeed, when adding or replacing a subcomponent, it may be necessary to replace the composite if the adaptation also impacts its coordination behaviour. Currently, there is no way to adapt or extend the coordination behaviour of a composite without changing the composite itself. This limitation will be addressed in future works.

5.3 Towards a set of reusable components

During the development of targeted applications, we were able to identify and define a first set of reusable Role and SessionPart in the area of calling, location and presence management. We can cite *Caller*, *Callee*, *RegAcceptor* as examples of roles that we were able to reuse in more than one application. *PresencePublishRequest*, *PresenceNotifier* and *PresenceSubscriptionRecorder* roles are not specifically tied to the Presence server application and therefore have the potential to be reused in other presence-enabled applications. SessionParts are also reusable. For example, redirect servers can easily reuse the SessionPart for registering in Proxy (*RegisterSessionSrv*) since they need the same functionality. This set of components is a first step and as the number and types of applications developed with our approach will grow, we expect that more reusable Roles and SessionParts will be discovered and that finally a rich base of telephony-related components will be available. The existence of such base should improve significantly the quality of telephony applications but also accelerate their development. It is one of our perspective to construct such a base.

6. RELATED WORKS

Over the years, several approaches for simplifying the development of SIP applications have been proposed. Most of these approaches are not component-based and can be subdivided in two main categories which are : 1) domain-specific language (DSL) like LESS[13] or SPL[8] to program specific kind of SIP entities by means of high-level concepts hiding the intricacies of SIP; 2) generic APIs or frameworks like JAIN-SIP [6] or PJSIP [9] which enables the programming of unrestricted SIP applications but have limited support for breaking the complexity of involved behaviours. Compared to these approaches, CIAO raises the level of abstraction in a direction similar to DSL but without sacrificing the generality. In addition, CIAO allows to increase the modularity of SIP entities making their maintenance and their evolution easier than with API or frameworks

Besides the previous approaches, we also found a few component-based approaches for developing SIP-based applications. Compared to our work, all these approaches only

focus on server-side component while CIAO offers a component model for developing both client and server entities. Another important difference with CIAO is that these approaches do not correlate components with the multiple sessions to manage but with features included in the application instead. This difference has some consequences on how components are organized and managed at runtime by respective approaches. Lastly, none of these approaches supports adaptation at the level of components like in our implementation of CIAO above OSGI. Hereafter, we present these approaches and discuss more specific differences with CIAO.

SIP Servlet [7] is a JAIN[7] specification of component and container for building SIP convergent applications similar to HTTP Servlet. SIP Applications are essentially composed of SIP Servlets which are Java components performing answering incoming request but also proxying them or creating new ones. Applications are hosted by a container which handles low-level functionalities and provides a set of high-level services to managed servlets. Compared to CIAO, SIP Servlet is not a hierarchical and dynamic component model. Its components are monolithic and are only instantiated once during the application life-cycle. Interaction between components for routing and handling messages from one or several sessions is possible but it must be done explicitly and is more restricted than in CIAO.

SLEE [5] is another JAIN specification for developing communication services based on SIP and others protocols occurring in the area of telephony. This specification defines both a model of high-level component for designing these services and a standardized environment for hosting and executing them. The atomic element defined by SLEE is the Service Building Block, a software component written in Java that send and receive events and perform computation on the receipts of subscribed event and its current state. A service is typically composed of a tree of SBBs. The SLEE environment instantiates such SBB tree when relevant events from the network are triggered and delegates the events to interested SBB instances. As we can observe, SLEE is also a hierarchical and dynamic component model like CIAO. However, the hierarchy of instantiated components is dedicated to a single session while several sessions are dynamically reflected in a CIAO hierarchy. As a result, interactions between sessions are more difficult to manage with the SLEE approach.

ECharts for SIP Servlet [11] is a state-based language that provides component-like entities (called Machines) translated into SIP Servlets. Such entities encapsulate state and logic for handling SIP signaling and have connectable ports to interact with SIP and non-SIP environments. Applications are specified by a top-level Machine that can contain nested machines. Machines are instantiated in response to initial request and the instances interact for handling subsequent messages. Compared to other models, ECharts is very close to SLEE in the sense that it instantiates a complete hierarchy of components for one session and therefore can be compared similarly with CIAO.

More generally, outside the previous works, the use of components for developing telecommunications applications have not been explored a lot. In this area, we may cite the work on DFC [3] which has inspired the previous SIP-based component approaches. The work described in [1] also adopts components but specifically targets the development of voice-based applications accessible to web browsers.

[7]JAIN is an industrial initiative to standardize the development of telecommunication systems in Java.

7. CONCLUSION

To tackle some issues arising from the design of advanced telephony applications based on SIP, we have proposed an component-based approach that enables involved entities to be constructed as actors playing roles in multiple sessions. Proposed approach have been experimented to develop various SIP entities with an increase in modularity and adaptability relatively to sessions.

In future works, we plan to optimize the instantiation process by recycling component instances instead of deleting them when a session is terminated. We also plan to identify recurrent coordination patterns between components (such as pipe-and-filter, delegation, forking patterns) and provide these patterns as reusable behaviour for components. Facilitating adaptation of components regarding coordination is also a perspective of this work. A last perspective is to study the typing rules underlying our components, particularly those relative to SIP Ports, in order to enable rigorous checking of components assembly.

8. REFERENCES

[1] R. Akolkar and al. Reusable dialog component framework for rapid voice application development. In *Proceedings of CBSE'05*, volume 3489 of *LNCS*, pages 306–321. Springer, 2005.

[2] A.Meddahi and G. Vanwormhoudt. *Telephonie SIP : Concepts, usages et programmation en Java.* Hermès-Lavoisier, 2012.

[3] G. Bond and al. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technologies*, 4(1):83–123, 2004.

[4] G. Bond and al. Experience with component-based development of a telecommunication service. In *Proceedings of CBSE'05*, volume 3489 of *LNCS*, pages 298–305. Springer, 2005.

[5] JCP. JSR 22: JAIN SLEE API Specification. 2004.

[6] JCP. JSR 32: JAIN SIP API Specification. 2006.

[7] JCP. JSR 289: SIP Servlet 1.1 Specification. 2008.

[8] N. Palix, C. Consel, L. Reveillere, and J. Lawall. A stepwise approach to developing languages for SIP telephony service creation. In *Proceedings of IPTComm'07*, 2007.

[9] B. Prijono. PJSIP - Open Source SIP Stack, http://www.pjsip.org/. 2011.

[10] The OSGi Alliance. Osgi service platform core specification, release 4.3. 2011.

[11] T.Smith and G. W. Bond. ECharts for SIP Servlets: a state-machine programming environment for VoIP applications. In *Proceedings of IPTComm'07*, 2007.

[12] G. Vanwormhoudt and A. Flissi. Session-based role programming for the design of advanced telephony applications. In *Proceedings of DAIS'11*, volume 6723 of *LNCS*, pages 77–91. Springer, 2011.

[13] X. Wu and H. Schulzrinne. Handling feature interactions in the Language for End System Services. In *Feature Interactions in Telecommunications and Software Systems VIII*, 2005.