A Stepwise Approach to Developing Languages for SIP Telephony Service Creation

Nicolas Palix, Laurent Réveillère, Charles Consel, Julia Lawall

A Stepwise Approach to Developing Languages for SIP Telephony Service Creation

Nicolas Palix Charles Consel
Laurent Réveillère
INRIA / LaBRI / ENSEIRB
Department of Telecommunications
351 cours de la Libération
F-33405 Talence Cedex, France
{palix,consel,reveillere}@labri.fr

Julia Lawall
DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

ABSTRACT

Developing a SIP-based telephony service requires a programmer to have expertise in telephony rules and constraints, the SIP protocol, distributed systems, and a SIP API, which is often large and complex. These requirements make the development of telephony software an overwhelming challenge. To overcome this challenge, various programming languages have been proposed to develop telephony services. Nevertheless, none of these languages as yet has a formal semantics. Therefore, the reference implementation, which may not be available, becomes the only source of information for the programmer to understand the subtleties of the language. Furthermore, this situation makes it difficult for third-party developers to port the language to another runtime system or to provide another implementation of the runtime system.

This paper presents a semantics-based stepwise approach for designing and developing a scripting language dedicated to the development of telephony services. This approach enables critical properties of services to be guaranteed and captures expertise on the operational behavior of a service. We have applied this approach to developing the *Session Processing Language* (SPL) [3] dedicated to SIP-based service creation. A variety of services have been written in SPL for our university department.

Keywords

SPL, Domain-Specific Language, SIP, Telephony services

1. INTRODUCTION

A commonly cited motivation for adopting the SIP protocol [13] as the basis of IP telephony is its support for services. A SIP service manages calls on behalf of a person or group, enriching the functionalities of the calling process. Typically, a service analyzes call parameters (*e.g.,* caller, callee, and subject), modifies and enriches these parameters (*e.g.,* by adding the caller's picture), queries and updates various peripheral systems (*e.g.,* calendar and call log), and performs some action on a call (*e.g.,* accept, reject, or redirect). When the SIP protocol was first created, developing a SIP service required extensive knowledge of network protocols, distributed systems, and complex, low-level platform APIs. Recently, however, high-level languages have emerged for developing SIP services, including SER [8], MSPL [11], CPL [9], SCML [2], LESS [18], and CCXML [17]. These languages have succeeded in making it easier and faster to develop SIP services. However, none of these languages as yet has a formal semantics. This situation ultimately hinders the development of advanced services, as developers must resort to studying the reference implementation, which may not be available, to understand the subtleties of the language. Furthermore, this situation makes it difficult for third-party developers to port the language to another runtime system or to provide another implementation of the runtime system.

In recent work, we have developed yet another language for SIP services, the *Session Processing Language* (SPL) [3]. In contrast to the above-cited languages, SPL has been developed based on a formal semantics which defines static properties that should hold for verification purposes, and a dynamic execution model.

IPTCOMM 2007 New-York, USA

This paper

This paper presents a stepwise approach for designing and developing a scripting language dedicated to the development of telephony services. This stepwise process includes a formal specification of the semantics of the language and its interaction with its underlying execution environment. This process enables critical properties of services to be guaranteed and expertise on the operational behavior of a service to be captured. We have applied this approach to develop SPL which is dedicated to SIP service creation.

The contributions of this paper are as follows.

- **SIP Virtual Machine**. The analysis of the SIP protocol is the starting point of our stepwise approach. The level of abstraction of the protocol was raised to match the needs of service development. This process resulted in a domain-specific virtual machine, named the SIP VM. This virtual machine is centered around the notion of a session, consisting of a set of events and

a state, which structures the development of telephony services.

- **Session Processing Language**. The SIP virtual machine, combined with common programming patterns found in telephony services, forms the main ingredient in the design of SPL. The syntax of an SPL service reflects the SIP VM session structure. Each kind of session is represented by a block containing the declarations of the variables and handlers associated with the session.

- **Formalized semantics**. SPL introduces notations and abstractions that are specific to the domain of SIP telephony services, facilitating the development process and offering expressiveness. The static and dynamic semantics of SPL have been formally specified, enabling a precise definition of SPL's interaction with the SIP VM.

- **Simplified implementation**. The formal definition of SPL is a foundation for defining program analyses and serves as a documentation for both service programmers and platform developers. As an example, an SPL interpreter has been developed in one week by a programmer. The interpreter represents about 2,000 lines of OCaml code [10] and is a straightforward mapping from the dynamic semantics.

The rest of this paper is organized as follows. Sections 2 and 3 introduce the SIP virtual machine and SPL. Section 4 describes the static and dynamic semantics of SPL and its relationship with the underlying SIP virtual machine, and Section 5 presents an assessment of our approach. Finally, Section 6 presents related work and Section 7 concludes.

2. SIP VIRTUAL MACHINE

The first step of our approach consists of identifying the building blocks of telephony services, enabling the definition of abstractions and operations required by developers. These abstractions and operations define a domain-specific virtual machine for SIP, named the SIP VM, that raises the level of abstraction of the SIP protocol to match the needs of telephony service developers. This VM sits within a SIP application server, between the SIP platform and the SIP services. The SIP VM mainly introduces the notions of operations, events, and sessions.

2.1 Operations

A SIP service typically performs some computation, forwards a SIP request and returns a SIP response. Within these computations, the SIP VM distinguishes signaling operations, which send a SIP message over the network, from non-signaling operations, which perform arithmetic and other local calculations.

Signaling operations.

The SIP VM defines the operation `forward` to allow a service to forward a request. To use this operation, the service provides not only a pointer to the request, but also the service's current code pointer and state. When the corresponding response is received, the SIP VM restores the code pointer and state of the service, thus causing the execution of the service to continue from the point of forwarding the request.

After completing the processing of a request, a service has to return a SIP response. Typically, a service returns the response it received after performing a `forward` operation. However, some services must return a constant SIP response without forwarding any request, and thus must be able to forge responses. For example, a service that implements a black list must return a reject response code (*e.g.*, `486`) to any black listed caller. The SIP VM thus introduces abstractions and operations to manipulate SIP responses, including operations to forge a SIP response from the current request.

Non-signaling operations.

The SIP VM also defines non-signaling operations such as arithmetic and boolean operations. These operations allow services to perform various computations. In addition, the SIP VM includes an extension framework to enable a service to call external procedures. Such procedures may be available on the local machine or may be invoked via remote procedure calls.

2.2 Events

The SIP VM defines three kinds of events to which a service can react: *verbatim* events, *refined* events and *platform* events. When a SIP request is received, the VM generates the corresponding event. If the meaning of the SIP request is unambiguous (*e.g.*, `ACK`), then the request is transmitted as a verbatim event, *i.e.*, as is. Some requests are, however, context sensitive, and require interpretation. For example, the SIP request `INVITE` either initiates a dialog or, if used in the context of an existing dialog, modifies dialog characteristics. The VM thus refines a SIP `INVITE` request as either the event `INVITE`, in the former case, or the event `REINVITE` in the latter. A platform event notifies a service of events internal to the platform that are relevant to the service logic. For example, the platform event `unregister` indicates the expiration of a SIP user registration.

Table 1 lists the events provided by the SIP VM. The names of verbatim and refined events are noted in uppercase to indicate that they correspond to a SIP message. The corresponding handler code in a service must perform at least one signaling operation. The names of platform events are noted in lowercase to indicate that they do not correspond to a SIP message.

Concepts	Events		
	Initial	Medial	Final
Service	`deploy`		`undeploy`
Registration	`REGISTER`	`REREGISTER`	`unregister`

Dialog	`INVITE`	`CANCEL`
		`ACK`
		`REINVITE`
			`BYE`
			`uninvite`

Subscription	`SUBSCRIBE`	`RESUBSCRIBE` `NOTIFY`	`unsubscribe`

Table 1: Classification of SIP VM events

2.3 Sessions

The SIP protocol defines a session as a multimedia communication dialog. SIP requests and time-outs enable to create, confirm, modify and terminate a dialog. Therefore, they define a complete lifecycle for a dialog session.

In the context of the SIP VM, we generalize this notion of a session lifecycle to the context of a subscription, a registration, and a service. A session is thus created when subscribing to an event, registering a user, or deploying a service. It persists until un-subscribing to the event, un-registering the user or un-deploying the service. The events generated by the SIP VM correspond to specific points in the lifecycle of the various SIP concepts. The events are thus classified as initial (creation), medial (confirmation and modification) and final (termination), as shown in Table 1.

A service typically performs some computations before triggering a signaling action. For example, a load-balancing service for a company would compute the total time each employee has spent answering the telephone to decide who should take the next call. To allow services to manipulate session states throughout their lifecycle, the SIP VM attaches a state to a session. This state is saved and restored by the SIP VM across event notifications.

3. THE SESSION PROCESSING LANGUAGE

The second step of our approach consists of designing a domain-specific language based on the abstractions furnished by the SIP VM. SPL is an example of a language for developing SIP telephony services. In the rest of this section, we give an overview of SPL. A complete specification of the language is available at the SPL web site,

`http://phoenix.labri.fr/software/spl/`

3.1 Sessions

The structure of an SPL service reflects the SIP VM session structure. Each kind of session is represented by a block containing the declarations of the variables and handlers associated with the session. Sessions are organized into a hierarchy, with a service session at the root, the registration sessions created within the service session as its children, and dialog and subscription sessions at the leaves. A session at any level has access to all of the variables of its ancestor sessions.

As an example, Figure 1 shows the SPL service `sec_calls`, that implements a counter service. This service maintains a counter of the calls that have been forwarded to a secretary, whenever the SIP user associated with the service is unable to take the call. The counter is set to `0` when the user registers, incremented when a call is forwarded to the secretary, and logged when the user unregisters.

3.2 Handlers

For each event generated by the SIP VM, an SPL service may define a handler to describe how the service should react to the event. In the case of verbatim and refined events, the handler processes a complete SIP transaction from the request to the final response. Platform events do not correspond to a SIP request. Therefore, the handler for such an event only performs non-signaling operations. A handler may be designated as either `incoming` or `outgoing` to provide a specific behavior for messages received or emitted by the user associated with the service, respectively. By default, a handler treats both incoming and outgoing requests.

The `INVITE` handler in Figure 1 (lines 17-24) illustrates SPL transaction processing for incoming `INVITE` requests. In this example, the `forward` operation (line 18) is used initially to forward an incoming call to the original recipient.

```
1 service sec_calls {
2   local void log (int);
3
4   registration {
5     int cnt;
6
7     response outgoing REGISTER() {
8       cnt = 0;
9       return forward;
10    }
11
12    void unregister() {
13      log (cnt);
14    }
15
16    dialog {
17      response incoming INVITE() {
18        response r = forward;
19        if (r != /SUCCESS) {
20          cnt++;
21          return forward 'sip:secretary@company.com';
22        } else
23          return r;
24      }
25    }
26  }
27 }
```

Figure 1: The counter service in SPL

This operation yields control to the SIP VM, which resumes the handler when a response is received. This response is then stored in the variable `r` and checked in line 19. SPL offers domain-specific abstractions and operations for such processing of SIP responses: Comparing a response to `/SUCCESS` checks whether the response code of the response is in the range `2xx`, while comparing a response to `/ERROR` checks whether the response code of the response is greater than `300`. In the example, if the call was not accepted, the original request is redirected to the secretary (line 21) and the new response is returned to the caller. If the call was accepted, the success response that was stored in `r` is returned directly (line 23).

As a component of a SIP application server, the SIP VM may receive requests at any time. To simplify SPL programming, the SIP VM never invokes a handler during the execution of another handler. Instead, events are queued by the SIP VM until the handler being executed yields control to the SIP VM, either due to a `forward` or a `return` operation. At this point, the SIP VM sends the request or response, respectively, out on the network and treats other pending requests. In the case of a `forward`, when a response arrives, the SIP VM resumes the handler containing the `forward` operation. The execution of a handler is thus atomic between two yield points, *i.e.*, two signaling operations.

3.3 Branches

The SIP protocol imposes a coarse-grained flow of control within each kind of session. For example, in a dialog, control may flow from `INVITE` to `ACK` and to `BYE`. To enhance expressiveness, SPL allows the programmer to refine the control-flow specification via a *branch* mechanism that passes control information from one handler to the next. This abstraction permits, *e.g.*, classifying a session as either personal or professional, and then introducing a logical personal or professional sub-thread across the remaining handler invocations of the session. As shown in Figure 2, the classification is done by adding the name of the branch to

the return from the current handler (*e.g.*, line 12) and the threads are implemented in each handler using the `branch` construct (*e.g.* lines 22-25).

```
1 service waiting_queue {
2  [...]
3   registration {
4    [...]
5     dialog {
6      [...]
7       response incoming INVITE() {
8         response resp = forward;
9        if (TO == 'sip:secretary@enseirb.fr') {
10         [...]
11         // Session tagged as "secretary"
12         return resp branch secretary;
13       } else {
14         [...]
15         // Session tagged as "private"
16         return resp branch private;
17       }
18       [...]
19     }
20     [...]
21      response incoming ACK() {
22       branch secretary {
23         // Specific treatment for secretary session
24         [...]
25       }
26       branch private {...}
27       [...]
28     }
29     [...]
30}
```

Figure 2: Inter-handler control flow

The branch construct is not only convenient for the programmer, it also increases the amount of information that is available to static verifications of SPL programs. Specifically, as the branch mechanism is a built-in language construct, it enables inter-handler control flow information to be determined accurately, before execution time. If this mechanism were not available, the programmer could simulate the same behavior using flags stored in the session state and conditionals in each handler testing the flag values. In this case, however, the control-flow information represented by the branches would not in general be apparent to the verifications associated with SPL (see *e.g.*, Section 4.1), which would result in a less precise analysis.

4. FORMAL SEMANTICS

As compared to general-purpose programming languages such as C, C#, and Java, two unique features of SPL are the session abstraction and the management of control and data across message forwarding. In this section, we describe both the static and dynamic semantics of SPL, focusing on the semantics of sessions, method invocations, and forward expressions. The complete semantics of SPL is available at the SPL web site.[1]

4.1 Static semantics

The purpose of the static semantics of a language is to specify what properties can be inferred about programs in that language, before execution. In the case of SPL, we present a static semantics that focuses on message forwarding. This static semantics can be enriched to consider other static properties, such as type safety.

[1] `http://phoenix.labri.fr/software/spl/`.

In SPL, when a handler needs to forward a message, it uses the `forward` expression. This forwarding yields control to the SIP VM, which sends the request. When a positive response is received (*i.e.*, a `2xx` response code), no additional forwarding of the request is allowed and the response should be returned as the result of the handler. This restriction prevents a service from forwarding a call to someone other than the party who accepted it. We now present an analysis that ensures that this restriction is satisfied for any program written in SPL.

4.1.1 Abstract Syntax

For conciseness, we focus on a subset of SPL related to forward operations, noted SPL_{fwd}. The abstract syntax of SPL_{fwd} is presented in Figure 3. We only consider handlers that return a `response` value and declarations that introduce variables of type `response`.

$$
\begin{array}{lcl}
H & ::= & \texttt{response}\ DIR_{opt}\ method\_name\ \texttt{\{}\ D^*\ S\ \texttt{\}} \\
DIR_{opt} & := & \texttt{None} \mid \texttt{Some(}DIR\texttt{)} \\
DIR & ::= & \texttt{incoming} \mid \texttt{outgoing} \\
D & ::= & \texttt{response}\ x\texttt{;} \\
S & ::= & id\ \texttt{=}\ \texttt{fwd}\ URI_{opt} \mid \texttt{return}\ E_R \mid \epsilon \\
 & \mid & \texttt{cond(}E_B\texttt{,}\ S_1\texttt{,}\ S_2\texttt{)} \mid S_1\ \texttt{;}\ S_2 \\
URI_{opt} & := & \texttt{None} \mid \texttt{Some(}URI\texttt{)} \\
E_R & ::= & id \mid \texttt{/ERROR} \mid \texttt{/SUCCESS} \\
E_B & ::= & id\ \texttt{==}\ E_R \\
URI & ::= & constant \mid id
\end{array}
$$

Figure 3: Abstract syntax of the SPL_{fwd} language

Abstracting an SPL program into its SPL_{fwd} counterpart is straightforward. This is illustrated by the counter service previously shown in Figure 1, whose SPL_{fwd} counterpart is displayed in Figure 4. In addition, we assume that standard program transformations such as *copy propagation* and *constant propagation* are performed to obtain the SPL_{fwd} program.

```
response Some(incoming) INVITE {
  response r;
  r = fwd None;
  cond (r == /ERROR,
        r = fwd Some(...); return r,
        return r)
}
```

Figure 4: Counter service in SPL_{fwd}

4.1.2 Semantic Rules

The semantic rules of SPL_{fwd} are described as inference rules with a sequence of premises above a horizontal bar and a judgment below the bar (see Figure 5). A judgment of the form $\tau_1 \vdash^D decs : \tau_2$ (rules 1 and 2) means that the evaluation of the declarations *decs* in the environment τ_1 returns a new environment τ_2. A judgment of the form $\tau_1 \vdash^S stmt : \langle \tau_2, \mathfrak{P} \rangle$ (rules 3 to 8) means that the evaluation of the statement *stmt* in the context of the environment τ_1 returns the new environment τ_2 and the boolean value $\mathfrak{P}$. This value becomes false as soon as an illegal use of `forward` is detected.

$$\frac{}{\tau \vdash^D \texttt{response}\ \ id : \tau[id \mapsto \mathsf{error}]} \quad (1) \qquad \frac{\tau \vdash^D D_1 : \tau_1 \qquad \tau_1 \vdash^D D_2 : \tau_2}{\tau \vdash^D D_1\ \texttt{;}\ D_2 : \tau_2} \quad (2)$$

$$\frac{\exists (x,\sigma) \in \tau.\ \sigma \neq \mathsf{error} \qquad \tau' = \tau[id \mapsto \bot]}{\tau \vdash^S id\ \texttt{=}\ \texttt{fwd}\ \ URI^? : \langle \tau', \mathsf{false} \rangle} \quad (3) \qquad \frac{\forall (x,\sigma) \in \tau.\ \sigma = \mathsf{error} \qquad \tau' = \tau[id \mapsto \bot]}{\tau \vdash^S id\ \texttt{=}\ \texttt{fwd}\ \ URI^? : \langle \tau', \mathsf{true} \rangle} \quad (4)$$

$$\frac{}{\tau \vdash^S \texttt{return}\ \ E_R : \langle \tau, \mathsf{true} \rangle} \quad (5) \qquad \frac{}{\tau \vdash^S \epsilon : \langle \tau, \mathsf{true} \rangle} \quad (6)$$

$$\frac{\tau \vdash^{E_B} E_B : (id, \sigma) \qquad \tau[id \mapsto \sigma] \vdash^S S_1 : \langle \tau_1, fwd_1 \rangle \qquad \tau[id \mapsto \neg\sigma] \vdash^S S_2 : \langle \tau_2, fwd_2 \rangle}{\tau \vdash^S \texttt{cond (}E_B\texttt{, }S_1\texttt{, }S_2\texttt{)} : \langle \tau_1 \uplus \tau_2, fwd_1 \wedge fwd_2 \rangle} \quad (7) \qquad \frac{\tau \vdash^S S_1 : \langle \tau_1, fwd_1 \rangle \qquad \tau_1 \vdash^S S_2 : \langle \tau_2, fwd_2 \rangle}{\tau \vdash^S S_1\ \texttt{;}\ S_2 : \langle \tau_2, fwd_1 \wedge fwd_2 \rangle} \quad (8)$$

$$\frac{\varnothing \vdash^D D : \tau_D \qquad \tau_D \vdash^S S : \langle \tau, fwd \rangle}{\vdash^H \texttt{response}\ \ dir^?\ \ method\_name\ \texttt{\{}\ D\ S\ \texttt{\}} : fwd} \quad (9)$$

$$\frac{(id, \sigma) \in \tau}{\tau \vdash^{E_R} id : \sigma} \quad (10) \qquad \frac{}{\tau \vdash^{E_R} \texttt{/SUCCESS} : \mathsf{success}} \quad (11)$$

$$\frac{}{\tau \vdash^{E_R} \texttt{/ERROR} : \mathsf{error}} \quad (12) \qquad \frac{\tau \vdash^{E_R} E_R : \sigma}{\tau \vdash^{E_B} id\ \texttt{==}\ E_R : (id, \sigma)} \quad (13)$$

Figure 5: The static semantics of SPL_{fwd}

A judgment of the form

$$\vdash^H \texttt{response}\ \ dir^?\ \ method\_name\ \texttt{\{}\ D\ S\ \texttt{\}} : \mathfrak{P}$$

(rule 9) means that the evaluation of the handler `response` $dir^?$ *method_name* `{` D S `}` returns the boolean value $\mathfrak{P}$. If $\mathfrak{P}$ is false, the handler contains an illegal use of `forward` and is rejected. If $\mathfrak{P}$ is true, the handler does not contain an illegal use of `forward` and is accepted.

A judgment of the form $\tau \vdash^{E_R} exp : \sigma$ (rules 10 to 12) means that the evaluation of the expression *exp* in the environment τ returns the status σ. The status is success if the expression is known to return a response code in the `2xx` class, error if the expression is known to return a response code not in the `2xx` class, and $\bot$ if the response code returned by the expression is not known. These status values form a partial order, with $\bot \sqsubseteq$ success and $\bot \sqsubseteq$ error. We use this ordering to determine how to merge environments in the static semantics of a conditional (rule 7).

Finally, a judgment of the form $\tau \vdash^{E_B} bool\_exp : (id, \sigma)$ (rule 13) means that the evaluation of a boolean expression in the environment τ returns a pair of an identifier *id* and a status value σ. The only boolean expressions allowed by SPL_{fwd} are equalities that compare an identifier to a description of a response, and thus evaluating such an expression to true or false gives information about the response code stored in the identifier. The information that can be obtained is then represented as a pair of the identifier *id* and the corresponding status value σ.

The key points of the analysis are in the treatment of `forward` (rules 3 and 4) and the treatment of conditionals (rule 7). Because the response received from a `forward` is always stored in a variable, the environment τ is essentially a record of the effect of the `forward`s that have taken place. If there is any variable whose value is success or $\bot$, then some previous `forward` has or may have succeeded. In either case (rule 3) the `forward` is illegal and false is returned. Only if every variable has the value error (rule 4), do we know that all previous `forward`s have failed and the current `forward` is allowed. In both rules, the environment is updated by binding the identifier *id* to $\bot$, to indicate that the result of its `forward` has not been tested, and thus might be success. Finally, testing of the result of a `forward` takes place in a `cond` statement (rule 7), and uses rules 10 to 13. In rule 7, the "then" branch, S_1, is analyzed with respect to an environment reflecting the fact that the test is true, and the "false" branch, S_2, is analyzed with respect to an environment reflecting the fact that the test is false. The latter environment is constructed using the operator $\neg$, defined as $\neg$success = error, $\neg$error = success and $\neg\bot = \bot$. The treatment of the statements S_1 and S_2 yields the two new environments τ_1 and τ_2. Because either may be available at execution time, the static semantics merges them using the operator $\uplus : \tau \times \tau \to \tau$, defined as follows:

$$\tau_1 \uplus \tau_2 \quad = \quad \{(x, \sigma_1 \sqcap \sigma_2) \mid (x, \sigma_1) \in \tau_1 \wedge (x, \sigma_2) \in \tau_2\}$$

This operator ensures that if the response for some `forward` is unknown in either branch, or is considered to have different values in the two branches, then the value of the corresponding variable becomes $\bot$.

4.2 Dynamic semantics

The purpose of a dynamic semantics is to specify what happens when a program is executed. We now present the dynamic semantics of SPL, examining the relationship between the language constructs and the underlying virtual machine.

Sessions. SPL is designed around the concept of a *session*, encapsulating a set of variables and handlers. To identify sessions uniquely, each session is associated with a unique

label. To describe the position of a session in the hierarchy, a session is furthermore associated with an *address*, which is a sequence of the labels of the session and its ancestors. Information about a session is stored in a global state σ, mapping an address to a tuple containing some status information about the session, a *session environment* mapping the session variables to their values, and a list of the addresses of the sub-sessions:

$$\sigma \in \mathsf{state} = \mathsf{address} \rightarrow \mathsf{status} \times \mathsf{env} \times \mathsf{address\ list}$$

The semantics of SPL uses a set of functions that manipulate sessions: create_session, prepare_method_invocation, continue_session, and end_session.

The function create_session extends the global state with an entry for a new session and has the following type,

$$\begin{array}{l}\mathsf{create\_session}: \\ \quad \mathsf{program} \times \mathsf{state} \times \mathsf{address} \times \mathsf{method\_name} \rightarrow \mathsf{state}\end{array}$$

The entry's status information includes a flag true indicating that the session is live and a reference count 0 indicating that no handler is currently executing in the session. The entry's session environment is obtained by evaluating the session variable declarations in an environment binding the session variables of the ancestor sessions. Finally, the entry's list of sub-sessions is empty. The result is a new global state.

Once a session has been created, methods can be invoked within the session. Method invocation is initiated using the function prepare_method_invocation, of type:

$$\begin{array}{l}\mathsf{prepare\_method\_invocation}: \\ \quad \mathsf{program} \times \mathsf{state} \times \mathsf{address} \times \mathsf{method\_name} \times \mathsf{direction} \rightarrow \\ \quad\quad \mathsf{decl\ list} \times \mathsf{stmt} \times \mathsf{env\ list} \times \mathsf{state}\end{array}$$

This function extracts the declarations and body associated with the method, retrieves the sequence of session environments associated with the session and its ancestors, and increments the reference count, indicating that a handler is executing in the session. The declarations, body, and session environments are returned, with a new global state.

Execution of handler code manipulates the sequence of session environments of the current session and its ancestors, as obtained by prepare_method_invocation. When execution of the handler code completes, these environments must be reinserted into the global state, which is done by the function continue_session, having the following type:

$$\mathsf{continue\_session} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \times \mathsf{env\ list} \rightarrow \mathsf{state}$$

The behavior of this function depends on whether the session is still live. If so, continue_session updates the global state with the new session environments and decrements the reference count, indicating that execution of the current handler has completed. If the session is no longer live, continue_session additionally calls end_session to determine whether the session should be destroyed.

Due to the sharing of state between handlers and between sessions, the most complex part of session management is session termination. Termination of a session is requested either when execution of certain handlers returns an error code or upon invocation of a final method (see Table 1). Nevertheless, it is not always desirable to destroy the session immediately. One issue is that handlers of the current session or its sub-sessions may be waiting for responses. For example, a dialog session can be waiting for a response to a `REINVITE` request sent by one party when it receives a `BYE` request sent by the other party. When the `REINVITE` response arrives, some code may be executed by the `REINVITE` handler, which may refer to the session variables. Thus, a session is only destroyed when its reference count is 0, indicating that no handler is executing in the session. Another issue is that in some cases, a session may end from the point of view of SIP, but should persist at the SPL level. An example is a registration session, which terminates at the SIP level either on an explicit request or on expiration of a timer. Registration, however, is not necessary for existing dialogs or subscriptions to continue, and these dialogs or subscriptions may refer to the registration variables. Final methods are thus classified as *persistent*, meaning that the session no longer accepts sub-sessions but is not destroyed until all sub-sessions terminate, or *non-persistent*, meaning that the session and all sub-sessions terminate immediately, subject to the reference count constraint. For example, the method `unregister` for registrations is persistent, while the method `undeploy`, for services, is non-persistent to allow a system administrator to take down the system in a timely manner. Session termination is managed by the function end_session, which is invoked by continue_session whenever the session is not live.

Method invocation. The semantics of SPL terms is described using a continuation-based abstract machine [1]. Execution in this machine is specified as a sequence of configurations, starting with a configuration representing the receipt of a message and ending with a configuration representing the returning of some information to the SIP VM. Intermediate configurations represent the execution of a term or the invocation of a continuation. A continuation is analogous to the stack used in the standard implementation of a procedural language. Whenever the semantics begins the execution of a term, it adds a frame to the continuation storing all of the information required to continue execution from the point of that term. This approach makes each configuration self-contained, and is used in the semantics of `forward`. The configurations used in the small step semantics of method invocation are as follows, where ϕ is the service code, *uri* is the destination of the request and s is a continuation:

- Method invocation:
  $\phi, \sigma \models_{\mathrm{mi}} \langle \mathit{method\_name}(\mathit{address}), \mathit{direction}, \mathit{uri} \rangle$
- Method continuation:
  $\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_env} \rangle \models_{\mathrm{mc}} s, \mathit{resp}$
- Handler body:
  $\langle \sigma, \mathit{address} \rangle, \mathit{envs}, \mathit{uri}, s \models_{\mathrm{h}} \mathit{decls}, \mathit{stmt}$
- Return to the SIP VM: value, σ

The semantic rules are described as inference rules, with a sequence of premises above a horizontal bar, and the current configuration and the next configuration in the execution sequence below the bar, separated by an arrow. We focus on dialog methods. The treatment of other kinds of methods is similar.

An initial `INVITE` method creates a new dialog session:

$$\frac{\begin{array}{c}\mathsf{create\_session}(\phi, \sigma, \mathit{address}, \texttt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', \texttt{INVITE}, \mathit{direction}) = \\ \langle \mathit{decls}, \mathit{stmt}, \mathit{envs}, \sigma'' \rangle\end{array}}{\begin{array}{l}\phi, \sigma \models_{\text{mi}} \langle \texttt{INVITE}(\mathit{address}), \mathit{direction}, \mathit{uri} \rangle \\ \quad \Rightarrow \langle \sigma'', \mathit{address} \rangle, \mathit{envs}, \mathit{uri}, \langle \texttt{INV}\ \ \phi \rangle \models_{\text{h}} \mathit{decls}, \mathit{stmt}\end{array}}$$

$$\frac{\mathsf{continue\_session}(\phi, \sigma, \mathit{address}, \mathit{envs}) = \sigma'}{\begin{array}{l}\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \_, \_ \rangle \models_{\text{mc}} \langle \texttt{INV}\ \ \phi \rangle, \texttt{/SUCCESS/}\mathit{resp} \\ \quad \Rightarrow \texttt{/SUCCESS/}\mathit{resp}, \sigma'\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, \mathit{address}, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', \mathit{address}, \mathit{envs}) = \sigma''\end{array}}{\begin{array}{l}\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \_, \_ \rangle \models_{\text{mc}} \langle \texttt{INV}\ \ \phi \rangle, \texttt{/ERROR/}\mathit{resp} \\ \quad \Rightarrow \texttt{/ERROR/}\mathit{resp}, \sigma''\end{array}}$$

The first rule initiates the method invocation by creating the session and extracting the handler code and relevant session environments. The rule produces (bottom line) a configuration causing execution of the handler code. The continuation in this new configuration is labeled `INV`, indicating that after executing the handler body, some work should be done that is specific to an `INVITE` method. The second and third rules describe the invocation of this continuation. In the second rule, the result of the handler is a success code, in which case it only remains to update the global state using continue_session. In the third rule, the result of the handler is an error code, in which case set_persistence is called to indicate that the session is no longer live and that its termination should be nonpersistent (indicated by false). These changes to the session status cause the subsequent call to continue_session to call end_session to destroy the session.

Invocation of a medial or final dialog method is similar, except without the use of create_session. At the end of a medial method, the session always continues, and thus the continuation rule for a medial method is analogous to the `/SUCCESS` rule for the `INV` continuation. At the end of a final method, the session always terminates, and thus the continuation rule for a final method is analogous to the `/ERROR` rule for the `INV` continuation. While the termination of a session due to an error code in an initial method is always nonpersistent, the termination of a session due to invocation of a final method depends on the persistence associated with the method itself. The third argument to set_persistence is thus adjusted accordingly.

Forward expressions. Evaluation of a `forward` expression causes control to leave the SPL service and return to the SIP VM, which then sends the request out on the network and treats other pending requests. When a response arrives, the SIP VM resumes the handler containing the `forward` operation. This coroutine-like relationship between SPL and the SIP VM requires that the `forward` operation provide to the SIP VM enough information to restart the handler execution. For this, we use continuations, following a standard strategy for implementing coroutines [5].

The semantics of `forward` uses the following configurations:

- Expression: $\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_env} \rangle, s \models_{\text{e}} \mathit{exp}$
- Expression continuation: $\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_env} \rangle \models_{\text{ec}} s, \mathit{value}$
- Return to/from the SIP VM: value, σ

We consider the case where `forward` has no arguments, as illustrated by line 18 of Figure 1:

$$\frac{\mathsf{update\_envs}(\sigma, \mathit{address}, \mathit{envs}) = \sigma'}{\begin{array}{l}\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_envs} \rangle, s \models_{\text{e}} \texttt{forward} \\ \quad \Rightarrow \mathsf{forward}(\mathit{uri}, (\texttt{FORWARD}\ \ \mathit{address}\ \ \mathit{uri}\ \ \mathit{local\_env}) :: s), \sigma'\end{array}}$$

$$\frac{\mathsf{lookup\_envs}(\sigma, \mathit{address}) = \mathit{envs}}{\begin{array}{l}\mathsf{forward\_response}(\mathit{resp}, (\texttt{FORWARD}\ \ \mathit{address}\ \ \mathit{uri}\ \ \mathit{local\_env}) :: s), \sigma \\ \quad \Rightarrow \langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_env} \rangle \models_{\text{ec}} s, \mathit{resp}\end{array}}$$

The first rule initiates the forwarding operation. This rule uses update_envs to update the global state with the current values of the session variables and then passes the current continuation, as well as the address of the session and the values of local variables, to the SIP VM, which forwards the message. The second rule describes what happens when the response arrives. In this case, the SIP VM passes this information and the response back to SPL, which reconstructs the environment and applies the continuation to the response in order to continue the handler execution.

5. ASSESSMENT

In this section, we assess the benefits of developing a language like SPL using our stepwise approach.

5.1 Reasoning about programs

The formal definition of SPL serves as a foundation for defining program analyses. From the static semantics definition of SPL, we have implemented a type checker in OCaml [10] that consists of about 600 lines of code. Not only does the SPL type checker detect type mismatches, like the compiler of a general-purpose language, but it also checks SIP-specific properties. For example, it checks whether handlers manipulate SIP messages correctly with respect to the SIP protocol. Because SIP is a text-based protocol, message headers are represented as strings regardless of their intended types. To increase the type safety this situation, SPL provides the programmer with a typed interface to message headers.

Another example of a SIP-specific property is the constraint that message headers may either be read-only, write-only or both. The type checker makes sure that services manipulate headers in compliance with the protocol. A final example is the return type of a handler associated with an SPL event, which may either be void or response. The type checker detects whether each handler has the required return type.

In addition to type-related verifications, the static semantics of SPL can be used to check a variety of domain-specific properties, as illustrated by the analysis presented in Section 4.1, that verifies whether the `forward` operation is used appropriately by a service.

5.2 Capturing expertise

Several languages have been proposed to make SIP-based service creation easier and faster. However, they are only

defined informally, which makes it difficult for developers to understand the subtleties of the language or to implement the language and run-time systems.

In contrast, SPL is formally defined, making explicit in a high-level way an expertise in SIP-based telephony service creation and a model for run-time systems. In our experience, this repository of knowledge has proved to be extremely valuable for porting SPL from OCaml to Java. Furthermore, formalizing SPL has made it possible to thoroughly cover both language and run-time design issues, prior to implementing the language.

5.3 Making implementation simpler

The formal definition of SPL serves not only as a documentation but also as a foundation for implementing the language and its run time. Our first prototype implementation of the SPL interpreter was implemented in OCaml by a single developer in about one week. This implementation is a straightforward mapping of the dynamic semantics, where each inference rule is translated into an OCaml function. The case of the invocation of an initial `INVITE` method shown in Figure 6 illustrates the close relationship between the semantic rule and its OCaml counterpart. Note that the semantic rule presented in Figure 6-a contains more detail than the one presented in Section 4.2. The latter version was simplified for the sake of clarity; in particular, handling of branches was omitted.

The complete semantics of SPL consists of about 100 inference rules. Our implementation of the SPL interpreter written in OCaml is very close in size. In addition to this first prototype, a developer has used the SPL semantics as a reference to implement a Java version. While being much more verbose, the Java version only required two weeks to develop. Table 2 shows the sizes of both the OCaml and Java versions of the SPL interpreter.

	OCaml	**Java**
Development time	1 week	2 weeks
LOC	2 175	5 786
# of words	10 855	17 040
# of functions/methods	82	36
# of modules/classes	6	45

Table 2: Ocaml and Java Implementations

A prototype implementation of the SIP VM and the SPL interpreter has been developed and has been integrated to the SIP-based telephony system of our university. A variety of services have been written in SPL for our university's Department of Telecommunications. In addition to traditional services such as black listing or conditional redirections, SPL has proved its usability to define advanced services. For example, an SPL service has been deployed that allows a secretary to manage incoming calls, using a waiting queue, depending on a shared agenda, availability of the personnel, and information about the call such as the name of the caller, if known. In these experiments, SPL has demonstrated its usability and ease of programming. In addition, our implementations based on the formal semantics described in Section 4 have proved their robustness while showing no significant performance penalty.

6. RELATED WORK

Various approaches have been proposed to develop SIP-based telephony services. The SIP Express Router platform [6] relies on a restricted configuration language to define the message routing logic. Its API also offers hooks to extend the core platform with modules written in C. However, analyzing such modules to ensure that a service respects the SIP protocol automaton can be very challenging. The Microsoft Live Communications Server [11] introduces a dedicated language for coarse-grained dispatch of SIP messages. Services that require advanced functionalities can shift the processing of a message to a C# program that can access the platform through a powerful API. Consequently, programmers must choose between expressiveness and simplicity. JAIN SIP [15] and SIP Servlet [7] are the standard Java interfaces to a SIP signaling stack. They provide a powerful solution for developing SIP services. However, programmers still have to deal with protocol intricacies.

High level scripting languages such as CPL [9], LESS [18] and CCXML [17] have emerged for developing SIP services. Some approaches have been proposed to verify properties of services written in these languages. For example, detection of interaction between features has been explored in the context of CPL [12, 20] and LESS [19]. However, none of these languages has been formally defined. Therefore, language implementations and verifications rely on informal specification found in the available documentation, making them subject to variation.

Semantics-based methodologies for language development have focused on general-purpose languages, designed to be universal [14, 16]. The second author and Marlet have proposed an approach for developing languages dedicated to a specific application domain [4]. This approach uses the denotational framework to formalize the basic components of a language. The semantics definition is structured so as to stage design decisions and to integrate implementation concerns. Following this methodology, our stepwise approach is based on the definition of a SIP virtual machine centered around the notion of a session. This virtual machine, combined with common program patterns found in telephony services, forms the main ingredient in the design of a language for developing SIP telephony services.

7. CONCLUSION

In this paper, we have argued that the scope of the telephony domain make developing services an overwhelming challenge. To take up this challenge, we have proposed a stepwise approach for designing and developing a scripting language dedicated to the development of telephony services.

Following our approach, we have defined a SIP VM that provides a high-level and portable interface to SIP platforms. This SIP VM is centered around the notion of a session that structures the development of a service. We have furthermore designed a scripting language named SPL that offers high-level notations and abstractions for service development. This language hides the subtleties of a SIP platform, making service implementations more concise than their GPL counterparts, without sacrificing expressiveness.

The static and dynamic semantics of SPL have furthermore been formally specified, enabling a precise definition of its interaction with the SIP VM. The formal definition of the SIP VM and SPL is a foundation for defining program

$$\frac{\begin{array}{c} address = \langle service, rid, did \rangle \\ \mathsf{lookup\_branches}(\sigma, \mathsf{parent}(address)) = \langle branch \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle branch \rangle, \mathtt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, direction, \mathtt{initial\ INVITE}) = \langle m, decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle m, \langle rq, headers \rangle \rangle \rangle \end{array}}{\begin{array}{c} \langle \mathtt{initial\ INVITE}(rid, did), direction, rq, headers \rangle, \phi, \sigma \models service \\ \Rightarrow \tau, r, \langle \mathtt{INITIAL\_INVITE}\ \phi \rangle \models decls, stmt \end{array}}$$

6a. Semantic rule definition

```
let interpret message phi sigma service =
  match message with
      (I_INVITE(rid, did), direction, rqid) ->
        let address = [service;(Reg, rid);(Dial, did)] in
        let branch = lookup_branches(sigma, parent(address)) in
        let sigma' = create_session(phi, sigma, address, branch, Some(If.UNINVITE)) in
        let (m_par, decls, stmts, envs, sigma'') = prepare_handler_invocation(phi, sigma', address, direction, If.INVITE) in
        let tau = (sigma'', address) in
        let rho = (envs, (m_par, rqid), []) in
          spl_handler_body tau rho ([]::[[T_INITIAL_INVITE(phi)]]) (decls, stmts)
```

6b. OCaml implementation

Figure 6: Initial INVITE method invocation

analyses and serves as a documentation for both service programmers and platform developers. As an example, an SPL interpreter has been developed in one week by a programmer.

A variety of services have been written in SPL for our university department. In these experiments, SPL has demonstrated its usability and ease of programming. Its robustness has been a key factor in expediting service deployment.

Acknowledgement

This work has been partly supported by the European Commission under the IST Integrated Project AMIGO.

8. REFERENCES

[1] M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

[2] J.-L. Bakker and R. Jain. Next generation service creation using XML scripting language. In *Proceedings of The IEEE International Conference on Communications 2002*. IEEE, 2002.

[3] 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.

[4] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, Sept. 1998.

[5] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 293–298, Austin, TX (USA), Aug. 1984.

[6] iptel.org. *SER Developer's guide*, September 2003.

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

[8] J. Kuthan. SIP Express Router (SER). *IEEE Network Magazine*, July 2003.

[9] J. Lennox and H. Schulzrinne. CPL: A language for user control of internet telephony services. Internet Engineering Task Force, IPTEL WG, November 2000.

[10] X. Leroy. The Objective Caml System Release 3.09, 2006.

[11] Microsoft. *Live Communications Server Application API*, 2005.

[12] M. Nakamura, P. Leelaprute, K. Matsumoto, and T. Kikuno. On detecting feature interactions in the programmable service environment of Internet telephony. *Computer Networks (Amsterdam, Netherlands: 1999)*, 45(5):605–624, August 2004.

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

[14] D. A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[15] Sun Microsystems. The JAIN SIP API specification v1.1. Technical report, Sun Microsystems, June 2003.

[16] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[17] W3C. *CCXML: "Voice browser call control: CCXML version 1.0."*. http://www.w3.org/TR/ccxml/.

[18] X. Wu and H. Schulzrinne. Programmable end system

services using SIP. In *Proceedings of The IEEE International Conference on Communications 2002*. IEEE, 2003.

[19] X. Wu and H. Schulzrinne. Handling feature interactions in the language for end system services. In S. Reiff-Marganiec and M. Ryan, editors, *FIW*, pages 270–287. IOS Press, 2005.

[20] Y. Xu, L. Logrippo, and J. Sincennes. Detecting feature interactions in CPL. *Journal of Network and Computer Applications*, December 2005.