

# Call/C: A Domain-Specific Language for Robust Internet Telephony Services

Claus Brabrand and Charles Consel

INRIA / LaBRI; 1, Avenue du Docteur Albert Schweitzer,  
Domaine Universitaire - BP 99; F-33402 Talence Cedex; France  
{brabrand, consel}@labri.fr

**Abstract.** The convergence of telecommunications and computer networks has added host of new functionalities to telephony services including Web resources, databases, *etc.* Making these rapidly evolving functionalities available to customers critically relies on developing a stream of new telephony services. Fortunately, this convergence has made the programming of telephony services as accessible as the programming of networking services. Yet, an undesirable effect of this new situation is that such a basic commodity as telephony is no longer dependent on thoroughly tested software, developed by certified programmers. Telephony is now exposed to bugs as found in ordinary software development and caused by common deficiencies such as lack of programming experience and insufficient domain expertise.

To reconcile programmability and reliability of telephony, we present a domain-specific language aimed to specify robust services. This language, named Call/C, offers high-level constructs that abstract over intricacies of the underlying protocols and software layers. Call/C makes it possible for owners of telephony platforms to deploy third-party services without compromising safety and security. This openness is essential to have a community of service developers that addresses such a wide spectrum of new functionalities.

## 1 Introduction

IP telephony materializes the convergence between telecommunications and computer networks. This convergence is dramatically changing the face of the telecommunications domain moving from proprietary closed platforms to distributed systems based on network protocols. In particular, a telephony platform is based on a client-server model and consists of a *signalling server* that implements a particular signalling protocol (*e.g.*, the Session Initiation Protocol [1]). A signalling server is able to perform telephony-related computations that include resources accessible from the computer network, such as Web resources, Web servers, databases, *etc.* This evolution brings a host of new functionalities to the domain of telecommunications. Such a wide spectrum of functionalities enables telephony to be customized with respect to preferences, trends and expectations of ever demanding users. These customizations critically rely on a proliferation of telephony services. In fact, introducing new telephony services is facilitated by the open nature of signalling servers, as shown by all kinds of servers in distributed systems. However, in the context of telecommunications, such evolution should lead service programming to be done by non-expert programmers, as opposed to developers certified by telephony manufacturers. To make this evolution worse, the existing techniques to program server extensions (*e.g.*, Common Gateway Interface [2]) are rather low level, involves crosscutting expertises (*e.g.*, networking, distributed systems,

and operating systems) and requires tedious session management. These shortcomings make the programming of telephony services an error-prone process, jeopardizing the robustness of a platform.

### **This Paper**

This paper presents a domain-specific language (DSL) [3, 4] named Call/C, aimed to ease the development of telephony services without sacrificing safety and security.

*Domain specific.* The design and development of Call/C is supported by a thorough analysis of the telephony domain. This study has included signalling protocols, signalling platforms, and service programming techniques.

*High level abstractions.* Call/C offers language constructs that abstract over low-level details of telephony services. In particular, it uses the notion of a session to prevent a service from being split in as many pieces as there are request/response interactions between a client and a server. Control and state management operations are automatically generated by the Call/C compiler. Also, some domain-specific types are introduced to capture aspects that are inherent to telephony-related computations.

*Safety and Security.* Unlike a general-purpose language (GPL), Call/C has semantic restrictions and includes domain-specific extensions. These features enable to determine safety and security properties that are far beyond the reach of GPLs.

### **Contributions**

The contributions of this paper are as follows.

- We have conducted a domain analysis that has led to the identification of major problems in the development of telephony services: low-level programming interfaces, unrestricted access to the platform, safety and security risks, and required cross-domain expertise (Section 2).
- We applied the session-centered paradigm to the development of telephony services (Section 3). This approach makes the control flow of a service explicit and enables its state to be managed automatically. This paradigm enables accurate program analysis to be performed on telephony services.
- We propose language design principles to facilitate the development of telephony services without sacrificing safety and security. They include high-level constructs, semantic restrictions, domain-specific extensions. These design principles are illustrated by Call/C, a DSL dedicated to the rapid development of telephony services (Section 4).
- We show that Call/C is expressive enough to describe a wide range of telephony services. We demonstrate that Call/C programs can be checked to determine critical safety and security properties. We argue that Call/C abstracts over the underlying signalling technology. As a result, a Call/C program can be mapped into different technologies, thanks to our retargetable compiler (Section 5).

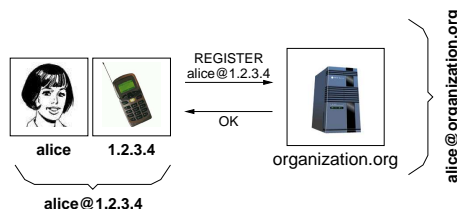


Fig. 1. User registration example.

## 2 Domain Analysis

It is only recently that the domain of telephony has actually converged with computer networking. As a result, the domain of telephony may now be analyzed from both a networking and a distributed system perspective. In this section, the presentation of this domain focuses on the main concepts and functionalities of a signalling platform. These aspects are then used to investigate existing approaches to the development of telephony services. Benefits and shortcomings of these approaches are assessed. The section concludes by listing requirements that should be used to develop telephony services without compromising the robustness of the telephony platform.

### 2.1 IP Telephony Infrastructure

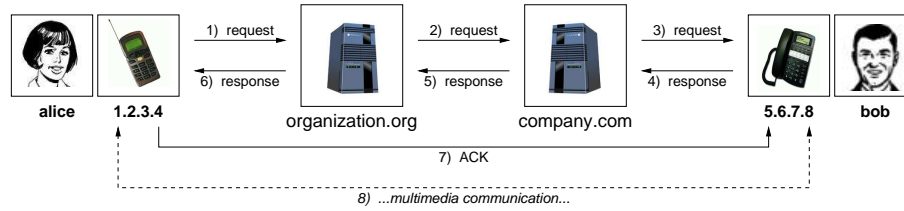
There has been three major IP-based signalling protocols in use over the years, namely, H.323 [5], the Media Gateway Control Protocol, MGCP [6], and the Session Initiation Protocol, SIP [1]. However, SIP, is becoming *the* protocol of choice for the telecommunications sector and beyond (*e.g.*, multimedia applications, alert facilities, instant messaging (IM), ...).

SIP is a lightweight distributed signalling protocol based on a client/server model. It aims to address the key stages of a session between clients: initialization, establishment, and termination. It also provides support for locating clients.

**Components** The clients are end-user devices connected to an IP network and capable of issuing (outgoing) calls and reacting to incoming calls. Devices of a client have varying multimedia capabilities, and range from simple hand-held telephones to sophisticated IP-telephony-enabled PCs, to automated Interactive Voice-Response (IVR) systems.

The servers are capable of relaying signalling call requests/responses to clients, as is commonly done by some servers on the IP network.

**An Example Interaction** Let us present the workings of a telephony platform by examining the signalling server interactions typically involved in processing a call. Suppose a user `alice` has a communication device associated with IP address `1.2.3.4`. Other clients are thus able to reach her directly using the address `alice@1.2.3.4`. To abstract away from physical IP addresses, the signalling server provides a domain name service mapping a symbolic address into a physical address. Moreover, the signalling server enables a symbolic address to be associated with multiple physical IP addresses, accounting for different devices, locations and forwarding addresses, used by a client. A registration facility is offered



**Fig. 2.** Session setup example.

by the local signalling server to dynamically allow the user to map his/her symbolic address to a specific physical address. Thus, *alice* can now *register* herself by contacting her local signalling server, say *organization.org*, and give it the IP address of her device. She can then be reached on her device via the symbolic address *alice@organization.org*. This registration process is depicted in Figure 1.

Suppose that *bob* has registered his phone, 5.6.7.8, at *company.com*, and is consequently reachable at *bob@company.com*.

Figure 2 shows the main steps and the server interactions involved in setting up a communication session with *caller* *alice* and *callee* *bob*.

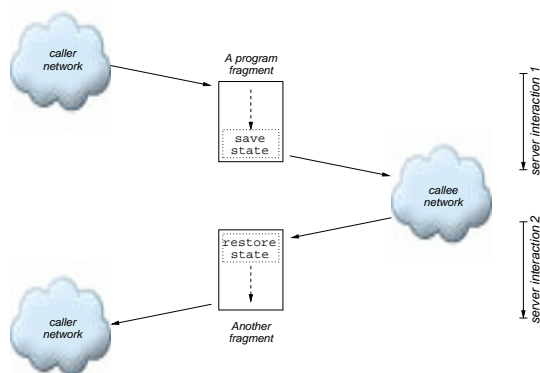
First, a session invitation request is sent by Alice to her local signalling server; it includes information such as Alice's IP address and Bob's symbolic address. This request is relayed, via a series of signalling servers, to Bob's local server. This server looks up Bob's currently registered physical IP address and forwards the request to this device (with IP address 5.6.7.8). When Bob accepts the call, an OK response is relayed back to Alice containing Bob's IP address and specific contextual information, such as his terminal capabilities and network features. This contextual information may be used to parameterize a multimedia communication application, once the session is initiated. Such information, although conveyed by the signalling platform, is outside the scope of a protocol such as SIP. When Alice receives the OK response and related information, her client sends an ACK message directly to Bob to notify that the session has been established. This last message concludes the so-called *three-way handshake* to establish an SIP session.

Notice that a signalling server, as presented so far, is offering a fixed behavior, in that, upon receiving a request it computes a so-called *signalling action* to be invoked, based on some fixed scheme (*e.g.*, forwarding or rejecting a call). Next, we discuss existing approaches to enriching the behavior of a signalling server by allowing telephony services to be programmed.

## 2.2 Programming Telephony Services

Defining a service logic is an enabling capability not only to customize the handling of a call with respect to a variety of criteria, such as current date and caller domain, but also to integrate the processing of a call into a rich computing environment, consisting of Web resources, databases, . . .

Because it uses a client-server model, a signalling server can readily use industrial-strength techniques to achieve a high degree of server programmability. We now examine two of the most widespread techniques of programmability, namely SIP CGI [7] and SIP Servlets.



**Fig. 3.** Implicit control-flow (*stateless interaction*).

*SIP CGI* is the Common Gateway Interface [2], well-known in the world of Web services, applied to the context of SIP. Upon receiving an incoming request, a *SIP CGI server* dispatches a so-called *SIP CGI script* to process the request and determine a processing. The script can be written in any general-purpose language and is responsible for writing out a signalling action. When (or if) the script terminates, the signalling server interprets this output; if the signalling action is legal, it is promptly carried out by the server.

*SIP Servlet* is essentially a Java callback interface for handling SIP requests and responses. The programmer may register callback methods, for instance `doInvite`, `doSuccessResponse`, `doRedirectResponse`, and `doErrorResponse`, which are invoked when corresponding signalling actions occur in the server. These callback methods are then responsible for instructing the underlying signalling server about which signalling action to perform through a high-level interface.

### 2.3 Call Processing Interaction Model

Compared to SIP CGI, the level of abstraction of SIP Servlets is higher in that a number of the low-level SIP protocol details are abstracted away by high-level interfaces. However, both programmability layers share a common trait: they are limited to the local processing of one server interaction (*i.e.*, one request/response pair). This view contrasts with the global processing of a call that encompasses all the steps towards establishing a session.

Figure 3 depicts a typical sequence of interactions from a signalling server's viewpoint. When receiving a session invitation request from some signalling component, it performs a sequence of computations that, before terminating, instructs the server to forward the request onto some other signalling component on the network. When the response eventually comes back from the network, another, conceptually independent, sequence of computations is performed to treat this new incoming "request", which in the example ends with a final response to the original request. From the signalling server, the two processings are essentially independent, each limited to one server interaction. Any communication beyond this boundary has to be conducted through explicit saving and restoring of the necessary state.

## 2.4 Evaluation

Let us now assess the benefits and shortcomings of the existing approaches to programming telephony services.

**Benefits** An obvious advantage of the SIP CGI and SIP Servlets approaches is that they rely on wide-spread and validated technologies. From a domain-specific viewpoint, another benefit of these approaches is that the processing of one server interaction has no implicit state. This property enables requests to be efficiently processed by re-using request handlers from a process pool, thus avoiding the overhead associated with process creation and destruction.

**Shortcomings** Yet, the CGI and Servlet approaches have a number of deficiencies that limit the scope of programming telephony services.

*Low-level programming.* Whether using SIP CGI or SIP Servlets, programming telephony services requires the understanding of the intricacies of the underlying protocol. Furthermore, the programmer must have an intimate knowledge of the supporting development library.

*Implicit control flow.* Programming a telephony service is a rather daunting task because the processing of the entire call session is scattered across different program fragments: one for the processing of each server interaction. In fact, the control flow of a service depends on the interactions with other signalling components, that is, the responses to signalling actions. Developing programs with such implicit control flow is error-prone and precludes static analysis to guarantee safety and security properties about the entire service.

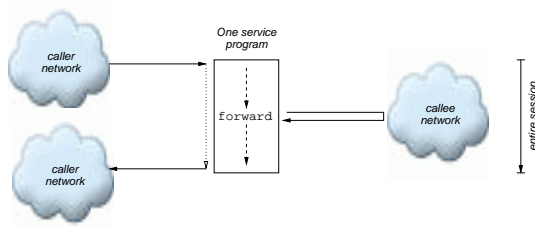
*Explicit data management.* Because a service consists of various program fragments, executed at different stages, each fragment needs to communicate beyond server interaction boundaries. As a result, the programmer needs to explicitly save and restore state (and control). In SIP CGI, this process also requires the state objects to be manually encoded and decoded. SIP Servlets does have a notion of a session object containing various per-session information and providing a convenient database to hold per-session data. However, data still need to be stored and retrieved explicitly.

Not only do the shortcomings examined above require the programmer to be highly trusted by the owner of the telephony platform, but they also demand high degree of expertise. This situation is a key limiting factor to provide platform owners with a rich variety of services and impedes the emergence of telephony service providers.

## 2.5 Domain-Specific Requirements

Making such a commodity as telephony subject to computer programming should not compromise its availability. To this end, based on our domain analysis, we present some essential properties that telephony services should exhibit.

*Security.* Because the services share a telephony platform, non-interference of services should be a guaranteed property. Typically, interference may be caused by an unbounded use of resources (*e.g.*, signalling server CPU).



**Fig. 4.** Explicit control-flow (*session-centered paradigm*).

*Safety.* Besides conventional safety properties, call completion should be guaranteed for a telephony service. This property should be checked before the service is executed to minimize the risks of losing calls due to a faulty service.

Notice that, although critical, none of the above requirements can be achieved by existing server programming techniques. Indeed, as discussed earlier, SIP CGI and SIP Servlets are low-level, have implicit control flow, and require explicit data management.

### 3 Sessions

One of the paramount advantages of domain-specific languages is that the level of abstraction can be made to match that of the problem domain. Concepts inherent to the problem domain can be turned into *abstractions* in a domain-specific language.

For the domain of telephony services, the notion of a *session* is a key concept and occurs naturally (the “S” in SIP). A session is the result of the complete sequence of server interactions and computations partaking in processing a call from a given signalling server’s perspective.

There is a strong analogy to the world of Web services where the concept of a session abstraction has been successfully applied [8, 9].

However, the communication model of telephony services is more complex. From a signalling server, the communication may go in *two* directions: backwards towards the caller, or forward towards the callee. In a Web service, communication only goes back directly to the client.

As mentioned earlier, existing telephony service solutions, for efficiency considerations, break apart the notion of a session by decoupling the launching of a request and the subsequent handling of its response.

#### 3.1 From Signalling Actions to Signalling Abstractions

In the context of telephony services, we aim to introduce a session abstraction that takes the form depicted in Figure 4. A telephony service becomes *one* program, conceptually running as a single process on a signalling server. The dots correspond to general-purpose computations which may be written in any programming paradigm (imperative, functional, object oriented, ...). Signalling actions such as the forwarding request are turned into high-level domain-specific signalling *abstractions* with a clear semantics.

A Signalling abstraction may be viewed as a synchronous remote procedure call initiated by the server, rendering the flow of control explicit to the programmer. The `forward` signalling abstraction takes as argument the necessary information to invoke the appropriate signalling action on the underlying platform. Execution is then conceptually suspended on the signalling server until the response comes back. At this point, the response is delegated back to the program as the result of the `forward` construction. Hereafter, execution proceeds normally with the same local state as whence it executed the `forward` construction. Notice that execution suspension is only conceptual; it may be implemented more efficiently to maximize service throughput as explained in Section 5.

### 3.2 Session Benefits

The benefits from applying the session-centered paradigm to telephony services can be divided into three main categories; let us examine each of them.

*High-level of abstraction.* The session-based signalling abstractions raise the level of abstraction, in that, fewer details are exposed to the programmer, who consequently does not need to be knowledgeable of, nor have expertise with, many of the underlying low-level protocol details. This eliminates altogether a whole spectrum of errors. Additionally, services are much more concise, increasing productivity, readability, and maintenance. Finally, without these details, the signalling abstractions are technology independent and are *retargetable* in the sense that they may be compiled to different underlying technologies and platforms as we shall see in Section 5.

*Explicit control flow.* The explicit control flow improves readability for the programmer, eliminating a range of control-related errors. The control flow is explicit and thus permits automated global control- and data-flow analysis. Indeed, many analyses depend on whether requests can be related to responses. Explicit control flow is also a necessary condition for guaranteeing termination, as discussed in Section 4. Protocol transition invariants, like ensuring that certain header fields have not been tampered with across interactions, is now possible. But also more global properties can now be verified, like determining that a call acceptance response originates from a client (*i.e.*, the response is not forged by a server). Finally, explicit control flow makes it possible to automatically determine global behavioral properties of services (see Section 4.4).

*Automatic data management.* Since the semantics of the `forward` construction automatically preserves the state, the programmer no longer needs to 1) determine which part of the state to explicitly preserve across interaction and 2) actually make it persist by manually encode, save, decode, and restore it. All this tedious and error-prone bookkeeping disappears completely from the programming task, improving on program conciseness.

## 4 Call/C

Call/C is a domain-specific language for developing telephony services; it is targeted at inexperienced or untrusted programmers to facilitate end-user and third-party development. It is based on the session-centered paradigm explained in the previous section with a simple C-like imperative core language for general-purpose computation. Opening up the server platform

to third-party development makes it crucial to abide by the strong safety and security requirements presented in Section 2.5. To do so, it naturally has some deviations from C in the form of safety restrictions and domain-specific extensions permitting program analysis.

Domain-specific language technology enables a language to offer domain-specific syntax, properties, optimizations, along with many other benefits [3, 4]. We now go through the most interesting domain-specific extensions and safety restrictions.

#### 4.1 Domain-Specific Abstractions

In the following, we present the Call/C abstractions dedicated to dispatching, signalling actions, as well as the domain-specific types, including call request and response structures.

**Service Dispatching** Contrary to the world of Web services, when a client issues a request (*i.e.*, places a call), no service program is designated to handle this request. For a single signalling server to support many services, a mechanism for mapping requests to services is needed.

To address this need, SIP Servlets provides a notion of a *deployment descriptor* with *servlet mappings* which are rules written in an XML file that specifies which service programs are invoked for which requests. This is done by expression matching of fields of incoming requests. Comparatively, SIP CGI provides no mechanism for dispatching.

*Dispatching Rules.* In Call/C, a declarative construct, named `dispatch`, provides a mechanism for service dispatching. The dispatching construct takes a boolean predicate and an identifier naming a *call processing function* to be executed when the boolean predicate is *true* for a request.

Atomic boolean predicates take an identifier designating a field (*e.g.*, `from`, `to`, or `subject`) of an incoming request and a regular expression. The predicate evaluates to *true* if and only if the field of the incoming request is in the language induced by the regular expression. Predicates generalize straightforwardly to any boolean combination of atomic predicates. Consider the following example:

```
dispatch toINRIA when (to ~ ".*@inria.fr");
```

This dispatching rule instructs the signalling platform to execute the call processing function, `toINRIA`, when it receives a call for someone at INRIA (*i.e.*, with an appropriate address in the `to` field).

Since many telephony services are for individual users, Call/C provides a convenient special regular expression, `user`, whose value is specified at service-deployment time, along with the following predefined dispatching rules:

```
dispatch incoming when (to ~ user);
dispatch outgoing when (from ~ user);
```

*Call Processing Functions.* When writing personalized services, the call processing functions `incoming` and `outgoing` will thus be executed when the call is *to*, respectively, *from* a particular user. Many personalized Call/C programs therefore have the following structure:

```

/* toplevel declarations (types, variables, functions) */

response incoming(call in) {
    /* imperative actions processing an incoming call */
}

response outgoing(call out) {
    /* imperative actions processing an outgoing call */
}

```

As can be seen, call processing functions are typed to take a *call structure*, `call`, as argument and to deliver a *response structure*, `response`, both of which will be explained later.

**Signalling Actions** Signalling abstractions include *call forwarding* and *call responding*, both of which are central in realizing the session-centered paradigm. The construction `forward` takes a call structure and a destination address, and returns a response structure; it *forwards* the call onto the address given and is synchronously blocked until the response is received. The `respond` construct takes a response structure and terminates computation, delivering the response as the final result of the call processing.

The following example service forwards incoming calls to the reception within office hours and to a voicemail service outside working hours, assuming the function `isWorkHour` is appropriately defined:

```

response incoming(call in) {
    response res;
    if (isWorkHour())
        res = forward(in, 'sip:reception@company.com');
    else
        res = forward(in, 'sip:company@voicemail.com');
    respond res;
}

```

**Domain-Specific Types** Call/C has domain-specific types for SIP addresses, call requests, and responses.

*SIP Addresses.* Call/C has a special type, `sip`, for lexically legal SIP address values as defined in the SIP protocol definition [1]. Call/C can therefore statically verify that all SIP constant addresses are well-formed.

*Predefined Call Structure.* The call request structure, `call`, is defined through a familiar, type-safe, C-struct type constructor as follows:

```

typedef struct call {
    const sip from;
    sip to;
    string subject;
    const string call_id;
}

```

The fields `from` and `to` designate the SIP address of the call originator and recipient, respectively. The subject of a call can be any string. The field `call_id` contains a random value guaranteed to be unique to each call and is often used as a database key. Call/C does not expose all message header fields to the programmer. For instance, the value `cseq` defines an interaction sequence number and is often used to derive control flow information to explicitly

tie together responses with forwarded requests. In the session model, this is no longer needed as the control flow is automatically managed by Call/C's high-level signalling abstractions.

The `call` type is predefined; no constructor function is exposed to the programmer. As a result, call values can only come as arguments to call processing functions, ensuring that only authentic calls can be processed. The fields defined as constant, through the `const` type modifier, are available as read-only to respect underlying signalling invariants.

*Predefined Response Union.* A response is defined as a discriminated union; this type constructor is similar to the one used to declare union values manipulated by Remote Procedure Calls (RPC) [10].

```
typedef union response switch (kind) {
  case ok:          void;                // 200 OK
  case redirect:   sip contact;         // 302 MOVED TEMPORARILY
  case busy:       void;                // 486 BUSY HERE
  case reject:     string reason;       // 603 DECLINE
  case error:      int code;            // xxx ERROR
}
```

Call/C distinguishes five different response kinds; `ok`, `redirect`, `reject`, `busy`, and `error`. The corresponding SIP response header octal codes are listed as comments. Responses of kind `redirect` contain a *contact address* of type `sip`; `reject` responses carry a *reason phrase* of type `string`; and, the `error` kind includes an integer *error code*.

For manipulating union values in a type-safe manner, the `switch` control structure is generalized to take a variable of type union with `case` branches naming union discriminator kinds. Union alternatives may thus only be manipulated in branches corresponding to their discriminator.

Since Call/C is not targeted towards performing multimedia communication, there is no constructor function for `ok` responses which, when returned, would establish such a communication. In contrast, it is possible to decline calls through the predefined response constructor function, `reject`.

## 4.2 An Example

The following service forwards an incoming call to the company boss, then if the response is `busy` or `reject`, the subject of the call is modified, and the call is forwarded to a secretary.

```
sip boss = 'sip:CEO@company.com';
sip secr = 'sip:secretary@company.com';

response incoming(call in) {
  response res = forward(in, boss);
  switch (res.kind) {
    case busy:
    case reject:
      in.subject = "[Fwd: " + in.subject + "];
      res = forward(in, secr);
      break;
  }
  respond res;
}
```

### 4.3 Further Domain-Specific Requirements

To further address the strong safety and security issues imposed by the domain, we introduced a number of constraints in designing Call/C. These constraints included eliminating pointer expressions, limiting array indexing, providing run-time support to handle exceptional cases, and statically ensuring service termination. Let us briefly elaborate on this last constraint.

To statically guarantee termination, when a service is compiled with the appropriate option, no unbounded iteration constructs, `do`, `while`, `for`, or recursion is permitted. However, we have incorporated a C#-style `foreach` control structure for bounded iteration on array types. Furthermore, all remote procedure calls interacting with non-Call/C code are required to specify a timeout and a timeout-handler (exceptional) statement.

### 4.4 Domain-Specific Verifications

As mentioned in Section 3.2, the session-centered paradigm enables a whole range of automated analyses about the global behavior of a running service. So far, we have only considered a resource consumption analysis which statically calculates the maximum number of signalling actions issued by a Call/C telephony service.

## 5 Implementation and Retargetability

Being a high-level domain-specific language for telephony services, Call/C is not biased towards any particular signalling platform or telephony service programming model. This neutrality renders Call/C *retargetable* in that it may generate code for different programming layers.

To illustrate this feature, we sketch the generation of target code for two very different layers, namely SIP CGI with C as gateway language and the SIP Servlets Java API. We are currently investigating targeting Microsoft's proprietary SIP programming API, which is also based on SIP. It should also be possible to target JAIN SIP<sup>1</sup>.

The most interesting aspects to illustrate is the code generation of the highly domain-specific signalling abstractions and service dispatching. In the following we examine how each of these two aspects are targeted towards the programming layers; SIP CGI and SIP Servlets. For each programming layer, we also give an example of a target-specific optimization.

### 5.1 Signalling Abstractions

For reasons of efficiency, both SIP CGI and SIP Servlets are stateless mechanisms. The session-centered paradigm, however, is a stateful mechanism wherein state survives signalling actions. Having a stateless target for our compiler opens up opportunities for improving performance.

The stateless approach splits a forwarding action in two independent parts: one fragment to issue the request and another fragment to handle the response. To implement the blocking semantics of the forward abstraction on top of this model, we need to relate a request with its response, and make the state persist across the signalling action.

---

<sup>1</sup> Java Integrated Network SIP (see <http://java.sun.com/products/jain/>).

More precisely, this amounts to: 1) encode the state and control along with a “program counter” to know where to later resume execution; 2) save the state and control on a persistent store; 3) instruct the signalling server to perform the forward action; and 4) terminate.

When the response eventually comes back we need to: 1) relate it with the previous request; 2) retrieve the state and control from the persistent store; 3) decode it; 4) restore the state and control which involves resuming execution with the statement following the forward statement (as designated by the program counter).

Note that these are steps that a service programmer needs to perform manually when maintaining the state across signalling actions. In Call/C, however, the compiler can automate this process completely by generating code to save all variables in scope at all forward program points.

**SIP CGI** Since SIP CGI has no serialization support, our compiler has to explicitly encode and decode all Call/C types. All variables are saved into a file whose name is parameterized with the environment variable `call_id`. Since this uniquely identifies a session, subsequent interactions can retrieve their state by reading this file parameterized by their `call_id`. In SIP CGI signalling actions are communicated to the signalling server through formatted instructions on standard out.

*Optimization.* SIP CGI provides a means for disabling script invocation for subsequent interactions<sup>2</sup>. This feature may be exploited by the compiler and activated for tail-recursive forward statements that just immediately return the response of the forward.

**SIP Servlets** SIP Servlets have convenient provisions for storing serializable, per-session, data<sup>3</sup>. However, composite user types still need to implement the `Serializable` interface and objects still need to be put in and retrieved explicitly. Signalling actions are performed by invoking the `Proxy::proxy()` method and termination is done by returning from the callback handler.

*Optimization.* SIP Servlets provide a mechanism<sup>4</sup> for automatically forwarding redirect responses received to the address designated as the new *contact* address. The compiler may directly determine from the control flow if this feature may be applied to minimize overhead.

## 5.2 Service Dispatching

Below we have briefly outlined how Call/C generates SIP CGI and SIP Servlet code for the service dispatching constructions.

**SIP CGI** Since SIP CGI lacks a notion of service dispatching, we have inlined the dispatching using `if`-tests in the `main` of the generated C program to determine which, if any, call processing functions needs to be executed.

<sup>2</sup> This feature is enabled by adding the following header line `CGI-Reexecute-On: never`

<sup>3</sup> Session data is manipulated by the methods `putAttribute()` and `getAttribute()` in `SipSession`.

<sup>4</sup> The method `Proxy::setRecurse()`

For the evaluation of the predicates, we statically compile the regular expressions to finite-state automata to efficiently and operationally determine whether the header fields match the regular expressions. If no call processing functions are dispatched, the script will perform the default action (as if no CGI program was present).

**SIP Servlets** For generating Sip Servlets code, the dispatching predicates are compiled to the XML *servlet mappings* files.

## 6 Related Work

The programming layers SIP CGI and SIP Servlets have been discussed extensively above. Coarse-grained telephony services can also be defined using various Graphical User Interfaces (GUIs). In principle, GUIs may directly produce SIP CGI or SIP Servlets. Another approach consists of producing an intermediate representation such as the Call Processing Language (CPL) [11, 12]. CPL is not a programming language; it is a restricted configuration language aimed to describe simple service variations.

Much work has been done in the area of DSLs [3, 4]. The notion of a domain-specific high-level session abstraction was successfully applied to the area of HTML-based Web services in the language Mawl [8, 13, 14]. It was later shown in the context of the <bigwig> [9] language how this model permitted sophisticated static analyses yielding strong safety guarantees about the behavior of Web services [15, 16].

## 7 Assessment and Conclusion

A key step in introducing a new domain-specific language is its assessment. This assessment typically takes into account criteria that are inherent to the domain under study, as well as criteria that are related to the DSL approach [3]. This assessment section follows the same structure.

Using a GPL, the programmer needs to explicitly address the following issues in developing a telephony service: *state management*, *session management*, *signalling actions*; and the actual *control logic* such as, general purpose computations, remote procedure calls, and database interactions.

Call/C completely eliminates state and session management, and integrates signalling actions in standard programming. The developer can then concentrate on the control logic which ultimately defines the behavior of a service.

In addition to some fairly standard DSL benefits, such as the factorization of expertise in the compiler, readability, maintainability, design stability, and greater reuse potential, there are some more profound benefits.

Relying on high-level, domain-specific, session-centric abstractions has numerous benefits in addition to generally raising the level of abstraction for the programmer. Services are more concise, the compiler becomes retargetable, and the explicit control flow permits static analyses to ensure strong safety and security properties which are crucial to the domain of telephony services.

The safety and security is what render third-party development possible. A robust language like Call/C is a key step towards having *one* language for the development of services

across the development spectrum from end-users to server administrators to third-party developers.

In fact, programming language research has a lot to offer to the domain of telephony services. In particular, the door is opened for a whole range of behavioral analyses (*e.g.*, resource cost analyses as a basis for billing, admission control, and the server farm size prediction). None of this was possible with existing programming methodologies for advanced telephony service development.

## Acknowledgments

The authors would like to strongly acknowledge Fabien Latry for his help in writing a preliminary version of the Call/C compiler.

## References

1. Rosenberg, J. et al.: Sip: Session initiation protocol. RFC 3261 (2002)
2. National Center for Supercomputing Applications (NCSA): (The common interface gateway (CGI)) Available from <http://hoohoo.ncsa.uiuc.edu/cgi/>.
3. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In Palamidessi, C., Glaser, H., Meinke, K., eds.: Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming. Volume 1490 of Lecture Notes in Computer Science., Pisa, Italy (1998) 170–194
4. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices **35** (2000) 26–36
5. ITU: Packet-based multimedia communication systems. ITU-T H.323. International Telecommunications Union (2000) Geneva, Switzerland.
6. Arango, M. et al.: Media gateway control protocol (MGCP). RFC 2705 (1999)
7. Lennox, J., Schulzrinne, H., Rosenberg, J.: Common gateway interface for SIP. RFC 3050 (2001)
8. Ladd, D.A., Ramming, J.C.: Programming the Web: An application-oriented language for hypermedia services. World Wide Web Journal **1** (1996) O'Reilly & Associates. Proc. 4th International World Wide Web Conference, WWW4.
9. Brabrand, C., Møller, A., Schwartzbach, M.I.: The <bigwig> project. ACM Transactions on Internet Technology **2** (2002)
10. Sun Microsystem: NFS: Network file system protocol specification. RFC 1094, Sun Microsystem (1989)
11. Lennox, J., Schulzrinne, H.: Call processing language framework and requirements. RFC 2824 (2000)
12. Lennox, J., Schulzrinne, H.: Cpl: A language for user control of internet telephony services. Internet Engineering Task Force, IPTEL WG (2000)
13. Atkins, D., Ball, T., Benedikt, M., Bruns, G., Cox, K., Mataga, P., Rehor, K.: Experience with a domain specific language for form-based services. In: Proc. Conference on Domain-Specific Languages, DSL '97, USENIX (1997)
14. Atkins, D., Ball, T., Bruns, G., Cox, K.: Mawl: a domain-specific language for form-based services. IEEE Transactions on Software Engineering **25** (1999) 334–346
15. Sandholm, A., Schwartzbach, M.I.: A type system for dynamic Web documents. In: Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00. (2000)
16. Brabrand, C., Møller, A., Schwartzbach, M.I.: Static validation of dynamically generated HTML. In: Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01. (2001) 221–231