INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Telephony Software Engineering: A Domain-Specific Language Approach

Laurent Burgy — Charles Consel — Fabien Latry — Julia Lawall — Nicolas Palix — Laurent Réveillère

N° 5548

Avril 2005

Thème COM

Rapport de recherche

N 0249-6399 ISRN INRIA/RR--5548--FR+ENG

Telephony Software Engineering:
A Domain-Specific Language Approach

Laurent Burgy * , Charles Consel * , Fabien Latry * , Julia Lawall † , Nicolas Palix * , Laurent Réveillère *

Thème COM — Systèmes communicants
Projet Phoenix

Rapport de recherche n° 5548 — Avril 2005 — 27 pages

Abstract: Telephony is evolving at a frantic pace, after having converged with computer networks and multimedia, and now offers a host of new functionalities. This evolution is led by the *Session Initiation Protocol* (SIP) for Voice over IP (VoIP) and third generation mobile phones. SIP supports user mobility and user availability notification. It manages various multimedia dialogs between parties, including audio/video conversations, games and instant messaging.

Because of the growing scope of the telephony domain, developing a telephony service requires a programmer to have an intimate knowledge of telephony rules and constraints, as well as the SIP protocol and its numerous companion protocols. Programming for a SIP platform also requires expertise in distributed systems, networking, and a SIP API, which is often large and complex. These requirements make telephony software engineering an overwhelming challenge.

This paper presents a domain-specific language for developing telephony services. This language, named *Session Processing Language* (SPL), offers domain-specific constructs and extensions that abstract over the intricacies of the underlying technologies. By design, SPL guarantees critical properties, far beyond the reach of general-purpose languages. SPL has been formally specified. We have also designed a SIP virtual machine to provide a high-level interface dedicated to service development.

Key-words: Domain-Specific Language, SIP, Software Engineering

* Author's current address: LaBRI / ENSEIRB, 351 cours de la Libération, F-33405 Talence Cedex, France.

† Author's current address: DIKU, University of Copenhagen, 2100 Copenhagen Ø, Denmark

Génie Logiciel de téléphonie : une Approche Langage Dédié

Résumé : De nos jours, la téléphonie évolue à une cadence infernale. Après avoir convergé avec les réseaux informatiques et multimédia, elle offre maintenant un large spectre de nouvelles fonctionnalités. Cette évolution est menée par le protocole *Session Inititation Protocol* (SIP) pour la Voix sur IP (VoIP) et la téléphonie mobile de troisième génération. SIP fournit un support à la mobilité des utilisateurs et à la notification de présence. Il permet également de gérer divers types de dialogues multimédia incluant les conversations audio/vidéo, les jeux et les messageries instantanées.
À cause de l'ampleur grandissante du domaine de la téléphonie, développer des services de téléphonie demande au programmeur une profonde connaissance des règles et contraintes inhérentes à la téléphonie ainsi que de SIP et ses protocoles compagnons. Programmer une plateforme SIP nécessite de plus une expertise en systèmes distribués, réseaux, et de maîtriser une API SIP souvent vaste et complexe. Ces exigences font du développement de services de téléphonie un véritable challenge.
Ce papier présente un langage dédié pour le développement de services de téléphonie. Ce langage, appelé *Session Processing Language*, offre des constructions et des extensions dédiées au domaine qui abstraient les difficultés des technologies sous-jacentes. De par sa conception, SPL garantit des propriétés critiques, bien au delà de celles que peuvent garantir des langages généralistes. SPL a été spécifié de manière formelle. Nous avons également conçu une machine virtuelle SIP qui fournit une interface haut-niveau dédiée au développement de services.

Mots-clés : Langage Dédié, SIP, Génie Logiciel

Contents

1 Introduction 4

2 SIP Background 6
2.1 SIP Basics 6
2.2 SIP Services 8
2.3 Programming SIP Services 8

3 From a Protocol to a Domain-Specific Virtual Machine 12
3.1 Operations 12
3.2 Partitioning of Operations 13
3.3 State 14

4 From a Domain-Specific Virtual Machine to a DSL 15
4.1 Organizing Sessions Into a Hierarchy 15
4.2 Intra-Handler Control Flow 16
4.3 Inter-Handler Control Flow 16

5 Semantics 18

6 Safety Properties 22
6.1 Execution without runtime errors 22
6.2 Respecting the SIP protocol 23

7 Related work 24

8 Conclusion 25

1 Introduction

Evolution in telephony has been occurring at a frantic pace ever since this area has converged with computer networks and multimedia. Now that telephony can interact with systems such as databases and Web services, it can offer a host of new functionalities. Meanwhile, telephony services represent a vast application area, ranging from hotlines to telemarketing centers.

At the forefront of modern telephony is the *Session Initiation Protocol* (SIP) [15, 14]. SIP is a fast-emerging signaling protocol for Voice over IP (VoIP) and third generation mobile phones. It is standardized by the IETF[1] and adopted by the ITU.[2] This protocol enables creating, modifying and terminating a communication between parties. Communications include audio/video communications, games, and instant messaging. SIP supports user mobility by giving a user a symbolic address that can be associated with various communication devices. It also provides for user availability; that is, willingness of a callee to accept a dialog.

Because of its large scope, SIP comes with a myriad of companion protocols, ranging from media formats to real-time transport protocols. And, although telephony is shifting to the computer realm, it retains a number of idiosyncratic rules and constraints, not known to the average programmer.

SIP is based on a client-server model. A SIP platform is typically implemented as a distributed system that consists of servers providing the various SIP functionalities. A fast-growing number of SIP platforms is becoming available; some target telephone carriers, others small businesses. A SIP platform can either be located at the core or at the edge of the network.

Some SIP platforms provide basic functionalities, programmable in a low-level language like C, while others offer a rich programming interface in languages like C# or Java. Regardless of the abstraction level and the programming language, all SIP platforms include very large and intricate APIs, as demonstrated by JAIN SIP, a standardized Java interface to SIP [17]. JAIN SIP consists of 130 classes and more than 3000 methods.

Realizing the potential of SIP requires a proliferation of telephony services. Yet, the origins and combined technologies used in the telephony domain make software engineering of telephony services an overwhelming challenge. To develop a service, a programmer must have an extensive expertise in the telephony domain, SIP and related protocols, distributed programming, networking, and SIP APIs. Correct use of all of these features is paramount because telephony is often heavily relied on and a buggy service can make telephony unavailable to a specific user, or even to all the users of a platform.

This Paper

This paper presents a domain-specific language (DSL) [6, 20], named *Session Processing Language* (SPL), whose goal is to ease the development of telephony services without sacri-

[1] IETF: Internet Engineering Task Force.
[2] ITU: International Telecommunications Union.

ficing safety. The design of SPL is based on a thorough analysis of the telephony domain. Because it offers high-level abstractions, it frees the service developer from low-level programming details and intricacies of underlying technologies. SPL guarantees critical properties, far beyond the reach of general-purpose languages (GPLs), by introducing domain-specific concepts and semantic restrictions.

The contributions of this paper are as follows.

- **SIP virtual machine.** To remedy the shortcomings of existing APIs, we have defined a SIP virtual machine, named the SIP VM, that provides a high-level interface, dedicated to service development. The SIP VM is centered around the notion of a session, consisting of a set of operations and a state, which structures the development of telephony services. Furthermore, by abstracting over the platform, the SIP VM enables services to be portable.
- **High-level, formalized language.** SPL introduces notations and abstractions that are specific to the domain of telephony services, facilitating the development process and offering expressivity. SPL programs are very concise as compared to their counterparts written in a GPL. For example, on average, SPL programs are 4 times more concise than their JAIN SIP counterparts. The semantics of SPL has been formally specified, enabling a precise definition of its interaction with the SIP VM. This formal definition is a foundation for defining program analyses.
- **Explicit control and data flow.** Existing SIP APIs obfuscate the control and data flow of services. In contrast SPL enables one to express the explicit control and data flow of a service, both locally to a service method and between related methods.
- **Verifiability.** SPL provides domain-specific constructs, restrictions, and extensions that make it possible to verify critical properties at the level of the telephony domain, the SIP protocol, the SIP platform and the service.
- **Real services.** We have used SPL to develop real telephony services for our university department, managing the calls of secretaries, system managers and the faculty. SPL has demonstrated an ease of programming, low learning overhead and robustness of the resulting services.

The rest of this paper is organized as follows. Section 2 presents SIP in more detail. Sections 3 and 4 present our approach, including the SIP VM and the SPL language. Section 5 describes the semantics of SPL and Section 6 describes SPL safety properties. Finally, Section 7 presents related work and Section 8 concludes.

2 SIP Background

We first present the SIP protocol, and then assess existing approaches to implementing SIP services.

2.1 SIP Basics

SIP is based on a client-server model. A SIP message can be sent or received by a client. A sent message is said to be *outgoing*; otherwise, it is said to be *incoming*.

SIP is a text-based protocol similar to other recent protocols such as HTTP[3] and RTSP.[4] A SIP message begins with a line indicating whether the message is a request (including a protocol method name) or a response (including a return code). A sequence of required and optional headers follows. Finally, a SIP message includes a body containing other information relevant to the message.

Logically, SIP is composed of three main entities, depicted in Figure 1: a *registrar server*, a *proxy server*, and a *user agent*. A registrar server allows a SIP user to record his current location. A proxy server dispatches SIP messages, whether incoming or outgoing, requests or responses. Typically, a SIP platform provides a registrar server and a proxy server to manage the messages to or from the users in a given domain. Additional proxy servers may be present in the network. Finally, a user agent exists for each communication device, and performs all SIP-related actions on its behalf.

To support mobility, a user is assigned a SIP URI (Uniform Resource Identifier), which is a symbolic address, analogous to an e-mail address. When a SIP proxy receives a message for a local URI, it asks the local registrar server to translate the URI into contact information for a specific user agent. A user must thus inform the registrar server of the user agent at which he would like to receive messages. As an example, the top of Figure 1 shows that Laura, in the domain `labri.fr`, has SIP URI `sip:laura@labri.fr`. When Laura wants to be reachable via her IP phone, she sends a `REGISTER` request including the phone's address to the `labri.fr` registrar server. If the response is the success code `OK`, subsequent calls to Laura will be routed to her IP phone. For another example, at the bottom of Figure 1, Bob in domain `inria.fr` has set his current location to be his laptop by communicating its IP address when registering.

Making a call To further present SIP, we describe the main steps involved in making a call, illustrated by Bob calling Laura once they have both registered their current location. To make the call, Bob's user agent emits an `INVITE` request including his current location and the callee's SIP URI, `sip:laura@labri.fr`. Like an e-mail message, the request is routed through a sequence of proxy servers until it reaches the domain `labri.fr`. At this point, the `labri.fr` proxy server queries the registrar server to locate Laura and forwards the request to her IP phone. Upon receipt of the request, Laura's phone starts ringing. If

[3]HTTP is the HyperText Transfer Protocol for transferring World Wide Web documents.
[4]RTSP is the Real Time Streaming Protocol for controlling the delivery of data with real-time properties.

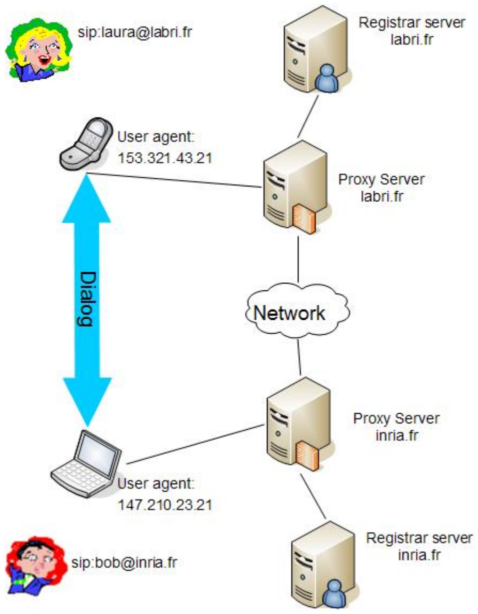

Figure 1: SIP architecture

she picks it up, an `OK` response, including Laura's location, is sent back to the caller. Bob then sends an `ACK` message to Laura to confirm the establishment of the call. At this point a conversation can start. When either Bob or Laura wants to end the conversation, they hang up the phone, causing their user agent to send a `BYE` request to the other party.

Management of SIP communications The SIP protocol introduces four concepts used in the management of communications: transaction, dialog, subscription and registration.

A *transaction* consists of a request and the response it triggers. For example, a `REGISTER` request sent to a user agent and its response form a transaction. A SIP message includes headers identifying the transaction it belongs to.

A *dialog* is a SIP relationship between two user agents that persists for some time. A dialog is initiated by an `INVITE` request and confirmed by an `ACK` request. A SIP message includes a header identifying the dialog it belongs to. A dialog persists until one party sends a `BYE` request.

A *subscription* allows a SIP user to request notification from a remote user agent when a certain event occurs. For example, one may subscribe to a presence event, to be informed when a particular user becomes available. A user interested in an event sends a `SUBSCRIBE` request to the responsible remote user agent, named the notifier. When the event occurs, the notifier sends a `NOTIFY` request to the subscriber. A subscription must be refreshed periodically.

A *registration* comprises the dialogs created by a user via the information provided to a particular registrar server. A registration persists until the user unregisters or his registration expires.

2.2 SIP Services

Most SIP platforms allow users to install services on an additional SIP entity known as a *feature server*. A service processes incoming and outgoing messages, essentially acting as a router.

We illustrate the notion of SIP service with our example. Suppose that a service has been installed for Laura that only allows customer calls to reach her IP phone during office hours. When Laura's proxy server receives the `INVITE` request from Bob, it invokes the associated feature server to obtain her service. The service is launched by the SIP platform with the `INVITE` request as input. If Bob's SIP URI is not part of Laura's customer list, the service rejects it by returning an error response (*e.g.,* `UNAUTHORIZED`). If Bob is a customer, the service gives control back to the SIP platform to *forward* the `INVITE` request to Laura's user agent (*i.e.,* her IP phone). Forwarding this request creates a new transaction that ends when the response is received or confirmed. If Laura is already on the phone, her user agent sends a `BUSY HERE` response back to the service. The service may then redirect the call to her voice mail, creating another transaction.

A SIP service may require that some state be associated with a given transaction, dialog, or registration. For example, Laura's service uses a customer list, which needs to be maintained while she is registered. The use of such state requires that the proxy server retain enough information to be able to associate relevant SIP messages to an existing transaction, dialog, or registration. The treatment of such messages is said to be *stateful*. Statefulness has some cost, and consequently very frequently executed services, such as those associated with an entire domain, are typically designed without any transaction, dialog, or registration state.

2.3 Programming SIP Services

Several SIP platforms offer an API for developing SIP services. These APIs are based on general-purpose programming languages such as C, Java and C#, and support various levels of abstraction. To illustrate the software engineering challenges involved in developing SIP services, we use the well-established and powerful Java-based platform JAIN SIP [8, 17]. The API provided by this platform consists of 130 classes and over 3000 methods.

Figure 2 shows the implementation of a simple service using JAIN SIP. This service maintains a counter of the calls that have been forwarded to a secretary, when the SIP user associated with the service is unable to take the call. The counter is set to `0` when the user registers, augmented when a call is forwarded to the secretary, and logged when the user unregisters. Some details have been omitted to save space. The boxes in Figure 2 connect the JAIN SIP code to the SPL implementation of the same service shown in Section 4, but are also used in the explanations below.

A JAIN SIP service instantiates the `SipListener` interface, requiring the methods `processRequest` for processing requests, `processResponse` for processing responses, and `processTimeout` for processing timeouts triggered by the platform. Typically, each method is structured as a dispatch on a request name or response code. In our example, the `sec_calls` class implements this interface. The `processRequest` and `processResponse` methods specify a treatment of `REGISTER` and `INVITE` requests and responses, while the `processTimeout` method specifies only a treatment of `REGISTER` timeouts, handling the case where a user allows his registration to expire.

The principal problem in implementing our service is to maintain the counter, which must be accessible to the treatment of all messages associated with a given registration. Java provides only two scopes: object fields that are global to the service and local variables that are local to the processing of a single request, response, or timeout. Accordingly, the programmer must explicitly implement the association between a registration and a counter. When the service receives a `REGISTER` request for a user that is not yet registered (block C1), it creates an object of the `State` class (block C0_1) that represents the counter for this registration. This `State` object is stored in a global environment `env` under an identifier `id` that is unique to the registration. Subsequently, when an `INVITE` request arrives for a registered user, `processRequest` forwards the request to the user. When the response arrives, `processResponse` executes the code comprising blocks C3 and C4. Block C3 handles the case where an error code is received, indicating that the user is unable to take the call. In this case, the counter associated with the user is retrieved from `env` and incremented before forwarding the call to the secretary. Finally, when the user unregisters (block C2_1) or his registration times out (block C2_2) the counter is logged and removed from `env`.

To improve performance, transactions should be stateless whenever possible. In the example shown in Figure 2, the JAIN SIP platform treats a transaction as stateless whenever the last argument to a forward operation is `null`. Statefulness is required when, as in the example above, some data defined in the request processing is used in the response processing. Statefulness also enables retransmission at the platform level, ensuring that every request obtains a response. If a service is partially stateless, and thus does not guarantee this property, then requests to which it does not respond may be retransmitted by the requester. Such retransmission reinvokes the service, and is thus only acceptable when the request processing is idempotent and does not involve much computation. In the example, block C3 increments the counter and forwards the request to the secretary. In this case, there is no further need for the counter in processing this call, but the transaction must be stateful because the processing of an `INVITE` request increments the counter, and is thus not idempotent.

The counter example clearly shows the difficulty of developing even simple services with existing approaches. Because these approaches are based on general-purpose languages, there is no support to guide the programmer through the intricacies of the protocol and the API. The separation of a service into distinct entry points for request and response processing

```
class State {
 private int cnt;
 public State() {}                                       C0_1
 void setCounter (int x) { cnt = x; }
 int getCounter() { return cnt; }
}

public class sec_calls implements SipListener  {
    [...]
 public void processRequest (RequestEvent requestEvent) {
  Request rq = requestEvent.getRequest();
  SipProvider sipProvider = (SipProvider)requestEvent.getSource();
  ServerTransaction st = requestEvent.getServerTransaction();
  String method = rq.getMethod();

  if (st == null) {
    if (method.equals (Request.INVITE) ||
        method.equals (Request.REGISTER) ) {
      st = sipProvider.getNewServerTransaction (rq);
    }
  }
  [...]
  if (method.equals (Request.REGISTER)) {
    if (!registrar.hasExpiresZero (rq)) {
      if (!registrar.hasRegistration (rq)) {
        State state = new State();
        int id = env.getId (st);                         C1
        env.setEnv (id, state);
        state.setCounter (0);
      }
    } else if (registrar.hasRegistration(rq)){
        int id = env.getId (st);
        State state = (State)env.getEnv (id);            C2_1
        local.log (state.getCounter());
        env.delEnv (id);
    }
    forward.request (targetURIList, sipProvider, rq, st);
  }
  if (method.equals (Request.INVITE)) {
    if (registrar.hasRegistration (rq)) {
      Vector targetURIList = registrar.getContactsURI (rq);
      forward.request (targetURIList, sipProvider, rq, st);
    }
    else {
      Response r = msg.createResponse (Response.NOT_FOUND, rq);
      if (st != null) st.sendResponse (r);
      else sipProvider.sendResponse (r);
    }
  }
    [...]
 }

 public void processResponse (ResponseEvent responseEvent) {
  Response response = responseEvent.getResponse();
  SipProvider sipProvider = (SipProvider)responseEvent.getSource();
  ClientTransaction ct = responseEvent.getClientTransaction();

  if (ct != NULL) {
    Request rq = ct.getRequest();
    if (rq.getMethod().equals (Request.REGISTER) &&
      response.getStatusCode() >= 300) {
        int id = env.getId (ct);
        env.delEnv (id);                                  C0_2
        forward.response (sipProvider, response, ct);
    }
    if (rq.getMethod().equals (Request.INVITE)) {
      Request rq = ct.getRequest();
      ServerTransaction st = rq.getServerTransaction();
      Vector targetURIList = null;
      if (response.getStatusCode() != 200) {
        int id = env.getId (st);
        State state = (State)env.getEnv (id);
        state.setCounter (state.getCounter() + 1);
        AddressFactory addr = getAddressFactory();
        SipURI sipURI = addr.createSipURI ("secretary", "nist.gov");   C3
        targetURIList = new Vector();
        targetURIList.addElement (sipURI);
        forward.request (targetURIList, sipProvider, rq, st);
      } else {
          forward.response (sipProvider, response, ct);  C4
      }
    }
  [...]
  } else {
      forward.response (sipProvider, response, null);
  }
 }

 public void processTimeout (TimeoutEvent timeOutEvent) {
  Timeout timeout = timeOutEvent.getTimeout();
  if (timeout.equals (Timeout.REGISTER)){
    int id = timeout.getId();
    State state = (State)env.getEnv (id);                C2_2
    local.log (state.getCounter());
    env.delEnv (id);
  }
 }
}
```

Figure 2: JAIN SIP example service

greatly obfuscates the control and data flow across transactions. The lack of scoping at the level of dialogs and registrations implies the need for complex state management that further obscures the service logic. Finally, the choice between statefulness and statelessness requires a careful analysis of the code and depends on several considerations. Errors in this choice due to non-idempotence of the service are difficult to detect if the service is tested in an environment where dropped messages are rare.

3 From a Protocol to a Domain-Specific Virtual Machine

All existing SIP APIs pursue the same goal: offering both in-depth and in-breadth access to the SIP protocol. In practice, this goal has translated into very large and complex APIs. Because these APIs are based on general-purpose languages, the service programmer must write repetitive glue code as the prologue and epilogue of API invocations. APIs also offer little support for structuring services or for managing service data. Finally, SIP APIs are platform-specific and thus do not permit portability.

To alleviate these problems, we define a domain-specific virtual machine for SIP, named the SIP VM, providing the programmer with a high-level portable interface dedicated to telephony service development. This VM fits into the SIP architecture between the SIP platform and the SIP services, as shown in Figure 3. A key contribution of the SIP VM is the introduction of a domain-specific notion, named *session*, that represents a design framework for service development.[5] A session consists of operations and a state. We examine each of these components.

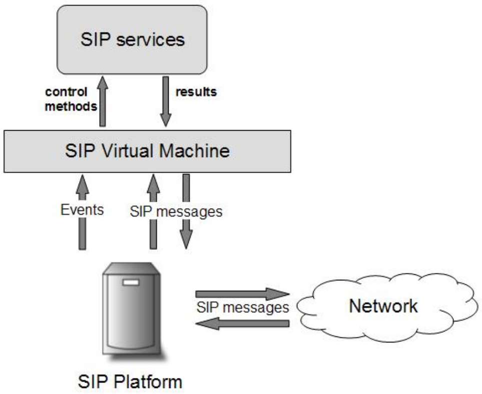

Figure 3: Our SIP architecture

3.1 Operations

To raise the level of abstraction, the SIP VM introduces *control methods* for which a service defines handlers. Control methods represent a uniform SIP interface that is dedicated to services. Control methods include verbatim SIP requests, refined SIP requests and platform events.

A SIP request is propagated verbatim by the SIP VM if its meaning is unambiguous (*e.g.,* `ACK`). Some requests are, however, context sensitive, and thus require interpretation.

[5]Note that *Session* in the name *Session Initiation Protocol* only refers to multimedia communication dialogs. Our notion of session generalizes it to the subscription, registration, and service abstractions.

For example, the SIP request `INVITE` either initiates a dialog or, if used in the context of an existing dialog, modifies dialog characteristics. The SIP VM thus refines a SIP `INVITE` request as either the control method `INVITE`, in the former case, or the control method `REINVITE` in the latter. Thus, the problem of distinguishing between these cases is factorized out of the service code. Verbatim and refined control methods are noted in uppercase. The corresponding service handlers must perform a signaling action.

The third kind of control method notifies a service of events internal to the platform that are relevant to the service logic. For example, the control method `unregister` is associated with the expiration of a SIP user registration. Such control method names are noted in lowercase to indicate that they do not correspond to a SIP message.

3.2 Partitioning of Operations

Existing SIP APIs do not support, or even suggest, an approach to developing services. These APIs represent a direct mapping of the SIP protocol and the programming paradigm used to develop the platform. Yet, the protocol includes concepts that are specific to the telephony domain and could be used as a framework for designing services. Consider, for example, the concept of a dialog that corresponds to a call. A dialog is a natural design thread to develop a service logic; it covers a complete lifecycle: creation, confirmation, modification, termination. What we previously named a SIP concept, the SIP VM elevates to a true design abstraction for services.

The SIP concepts subscription and registration also have a lifecycle and can be viewed as design abstractions. We furthermore introduce a design abstraction for services. At some point, a service is deployed and associated with a user or group of users. This binding should persist until the service is undeployed.

The control methods fit into specific points of the lifecycle of the various design abstractions. They are thus classified as initial (creation), medial (confirmation and modification) and final (termination), as shown in Table 1.

Concepts	**Control methods**		
	Initial	**Medial**	**Final**
Service	`deploy`		`undeploy`
Registration	`REGISTER`	`REREGISTER`	`unregister`

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

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

Table 1: Control method classification

3.3 State

So far our design abstractions only refer to code (handlers), but services also need to be able to manipulate their own data. The SIP VM enables attaching a state to a dialog, a subscription, a registration or a service, and managing this state across the associated lifecycle. Thus, state management is factorized out of the service code.

Such a grouping of operations and state is analogous to an object in an Object-Oriented language. We refer to such a grouping as a *session*. In light of this notion, we now say that a session encompasses a design abstraction and a lifecycle refers to a session. An initial control method creates a session, a medial control method executes within a session, and a final control method ends a session.

4 From a Domain-Specific Virtual Machine to a DSL

We propose a new language, SPL (Session Processing Language), for developing telephony services. This language includes domain-specific constructs and semantics. It is designed around the abstractions furnished by the SIP VM. This section describes the salient features of SPL. The complete syntax of the language is available at the SPL web site, `http://phoenix.labri.fr/software/spl/`.

4.1 Organizing Sessions Into a Hierarchy

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. This syntax is illustrated by the SPL program `sec_calls`, shown in Figure 4, that implements the counter service presented in Section 2.

```
service sec_calls {
  processing {
    local void log (int);                    C0

    registration {
      int cnt;

      response outgoing REGISTER() {
        cnt = 0;                             C1
        return forward();
      }

      void unregister() {
        log (cnt);                           C2
      }

      dialog {
        response incoming INVITE() {
          response r = forward;
          if (r != /SUCCESS) {
            cnt++;
            return forward 'sip:secretary@nist.gov';   C3
          } else
            return r;                        C4
        }
      }
    }
  }
}
```

Figure 4: The counter service in SPL

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 illustrated in Figure 4, an outermost processing block declares service variables and functions, such as the external function `log`, followed if needed by handler definitions for `deploy` and `undeploy` (absent in our example). A registration block is defined inside the processing block. In our example, the registration block defines the `cnt` variable, a `REGISTER` handler, which initializes the counter, and an `unregister` handler, which logs the counter. Finally, a dialog block is defined inside the registration block. In our example, the dialog block only declares an `INVITE` handler, which increments the counter when an incoming call is rejected by the user.

4.2 Intra-Handler Control Flow

Handlers for verbatim or refined control methods typically perform some computation and then forward the SIP request. This forwarding yields control to the SIP platform, which sends the request. In existing SIP APIs, when the response is received, the service code must explicitly correlate the response with the request and perform some computation to restore the control flow suspended at the forward point before processing the response.

One of the goals of SPL is to factorize these tedious and error-prone computations out of the service. In SPL, a handler is written as a single unit that processes a transaction from the request to the response. When a handler needs to forward a message, it uses the `forward` expression, that gives the SIP VM the current code pointer and state. When the corresponding response is received, the SIP VM restores the code pointer and state of the service, and execution of the handler continues.

The `INVITE` handler in Figure 4 illustrates SPL transaction processing. In this example, an incoming call is forwarded to the user and his response is assigned to the variable `r`. The response is then checked. If the call was not accepted, the original request is redirected to the secretary and the new response is returned to the caller. If the call was accepted, the success response is returned directly.

The interested reader can compare the counter service written in SPL to the version based on JAIN SIP, using the block labels. The SPL version is more concise and higher-level than its JAIN SIP counterpart, in that it does not need to perform all the low-level SIP bookkeeping operations (service state, transaction statefulness, dispatch, *etc.*).

4.3 Inter-Handler Control Flow

Not only is a session a design thread, but it also represents a thread of control. The SIP protocol describes a coarse-grained session control flow, *e.g.*, in a dialog control may flow from `INVITE`, to `ACK`, 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, which introduces a logical subthread across the remaining method invocations of the session.

A branch is chosen for a session when the session is created. The branch is stored in the session state and is used to select the relevant code for each subsequent handler invocation in the session. For a service or registration session, the initial branch is `default`. On returning, a handler can specify a new branch, which overwrites the current one. When a service or registration handler is invoked, the code corresponding to the current branch is chosen, or the code corresponding to the default branch, if nothing else applies. Dialog and subscription sessions are treated similarly, except that the initial branch is inherited from the current branch of the parent registration session and branches accumulate rather than overwrite. On invoking a method, the code corresponding to the branch appearing earliest in the accumulated sequence of branches is selected.

```
1   service hotline_info {
2    processing {
3     type hotliner_t {uri name; time t_call; int ticks;};
4     ...
5     registration {
6      ...
7      response incoming REGISTER(request rq) {...}
8      void unregister() {...}
9
10      dialog {
11       hotliner_t callee;
12       time t;
13
14       response incoming INVITE() {
15        if (TO == 'sip:hotline@domain.com') {
16         foreach (h in hotline_reg) {
17          if (get_status (h) == AVAILABLE) {
18           response r = forward h.name;
19           if (r == /SUCCESS) {
20            callee = h;
21            hotline_reg.remove (h);
22            return r branch hotline;
23           }
24          }
25         }
26         return forward 'sip:voicemail@domain.com';
27        }
28        else {
29         response r = forward;
30         if (r == /SUCCESS) {
31          hotline_reg.remove (TO);
32          return r branch private;
33         }
34         return r;
35        }
36       }
37       response incoming ACK() {
38        branch hotline {
39         t = get_time();
40         set_status (h, PHONE);
41         return forward;
42        }
43        branch private {...}
44        branch default { return forward; }
45       }
46       response BYE() {
47        branch hotline {callee.t_call += get_time()-t;
48                         return forward; }
49        branch default { return forward; }
50       }
51       ...
52  }}}}
```

Figure 5: A hotline example

Branches are illustrated in Figure 5 by fragments of a hotline service, written in SPL. When an `INVITE` request is sent to the hotline SIP URI, the `hotline` branch is selected (line 22). When the callee is a named person, the `private` branch is chosen (line 32). When the `ACK` request is received, the processing depends on whether the call is private or for the hotline. The correct branch is automatically selected.

5 Semantics

In this section, we describe the semantics of SPL, focusing on the semantics of sessions, control method invocations, and forward expressions. The complete semantics is available at the SPL web site.

Sessions SPL is designed around the concept of a *session*, encapsulating a set of variables and handlers. Each session is associated with a unique *label* and with an *address* that 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 subsessions:

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

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, with the following type, extends the global state with an entry for a new session:

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

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 subsessions 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 subsessions 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 of subscriptions may refer to the registration variables. Final methods are thus classified as *persistent*, meaning that the session no longer accepts subsessions but is not destroyed until all subsessions terminate, or *non-persistent*, meaning that the session and all subsessions 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 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:
  $\phi, \sigma \models_{\text{mi}} \langle \mathit{method\_name}(\mathit{address}), \mathit{direction}, \mathit{uri} \rangle$

- Method continuation:
  $\langle \sigma, address \rangle, \langle envs, uri, local\_env \rangle \models_{\mathrm{mc}} s, resp$
- Handler body:
  $\langle \sigma, address \rangle, envs, uri, s \models_{\mathrm{h}} decls, stmt$
- Return to the SIP VM: $value, \sigma$

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:

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

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

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

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 $\mathsf{set\_persistence}$ 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 [9].

The semantics of `forward` uses the following configurations:

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

We consider the case where `forward` has no arguments, as illustrated by line 29 of Figure 5:

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

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

The first rule initiates the forwarding operation. This rule uses $\mathsf{update\_envs}$ 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.

6 Safety Properties

The telephony domain imposes stringent safety and robustness requirements. These intervene at two levels: First a telephony service should execute without runtime errors under all circumstances, to ensure continuity of service for other calls. Second, a telephony service should respect the invariants of the underlying protocol. In the context of SPL, these properties are ensured in part by the language structure and in part by analyses that exploit the high-level domain-specific operators of the language.

6.1 Execution without runtime errors

Execution without runtime errors requires that each operation used in the implementation of a service be able to complete on its provided inputs. For the operations specific to telephony services, possible errors include type errors, use of undefined variables, missing branches, incorrect saving and restoring of the state on a forward operation, and incorrect use of the SIP platform API.

Types The SPL type checker detects the same kinds of type mismatches that occur in general-purpose languages. It also checks SIP-specific properties of message manipulation. A handler can manipulate the headers of the initial request as well as those of any responses received in the course of handler execution. As SIP messages are represented as text, message headers are implemented as strings. SPL, however, imposes a typed view of these headers, and the type checker inserts the necessary coercions and ensures that the types are respected. Furthermore, the SIP protocol imposes that some headers are read-only, while others are write-only. The type checker ensures that only the allowed operations are performed. Finally, a handler must return either a response or void, according to the expectations of the SIP VM. The type checker verifies that the return type of each handler conforms to these expectations.

Variables and branches For variables, we require that a variable be defined before it is used. For branches, we require that a handler provide a block of code that corresponds to the current branch or sequence of branches. Both of these properties require reasoning about inter-handler control flow, which must take into account properties of the SIP protocol and the semantics of the branch mechanism. We focus on the problem of validating variable uses. The treatment of branches is similar.

We use as an example the hotline service of Figure 5, in which we are interested in verifying the validity of the use of the session variable `callee` in the `BYE` handler (line 47). First, we observe that `callee` is defined in the `INVITE` handler (line 20), which by the SIP protocol is guaranteed to be executed before `BYE`. Nevertheless, this observation is not sufficient to ensure that the use of `callee` is well-defined, because the definition of `callee` in `INVITE` appears under a conditional. The initialization of `callee`, however, dominates a return (line 22) in which the branch is set to `hotline`, and this is the only code in the dialog that sets the branch to `hotline`. Because the use of `callee` in `BYE` is in the `hotline` branch, we may conclude that the use of this variable is well-defined.

SPL provides a variable initialization analysis that takes the protocol and branches into account. Without the branch mechanism, performing the analysis would require reasoning about programmer-defined flags or other more complex conditions. Because reasoning about arbitrary code is difficult, in practice it would be necessary either to drop the check, providing a lower degree of safety, or to reject a larger set of programs.

Session state Forwarding a request requires leaving the execution of the service and returning to the SIP VM. When a response arrives, control may need to return to the service. For this, it is necessary to save the code pointer and the local variables that may be referred to by the rest of the handler. Collecting the local variables by hand is tedious and error-prone, and has to be redone whenever the code changes. SPL saves the code pointer and needed variables automatically. The saving of the variables can furthermore be automatically optimized to include only the variables that may be referenced after the forward operation.

SIP platform API SIP platform APIs, such as that of JAIN SIP, are very complex. In the SPL framework, interaction with the SIP API is encapsulated in the SIP VM, and thus is not a concern of the service programmer. When programming a SIP service using a general-purpose language, on the other hand, use of the API is mixed with the service logic. The service becomes difficult to understand, and there is no verification that the various API functions are called in a consistent way.

6.2 Respecting the SIP protocol

Respecting the SIP protocol requires that the service treatment of each request be compatible with the original request processing. SPL provides only two kinds of signaling actions: forwarding a request and returning a response. In a forward operation, the forwarded request is implicitly the incoming request, implying that a service cannot forward a fresh request or a request saved from a previous method invocation. While the service can specify a new destination for a request or update some request headers, it cannot change the method name, ensuring that any forwarded request plays the same role in the protocol as the original request. In a return operation, the response must be compatible with any forward operations performed by the method. Specifically, if any forward operation succeeds, the response must be a success code, and if no forward operation succeeds, then the response must be an error code. This property is checked by an analysis that considers each control-flow path through the handler and uses any tests on responses to infer whether forward operations succeed or fail.

The SIP protocol allows some methods to forward any number of requests but allows others to successfully forward at most one. In the latter case, a control-flow analysis similar to the one inferring the allowed response ensures that every forward operation except the first one is guarded by error checks on the results of all earlier forward operations.

7 Related work

A variety of SIP approaches have been proposed for programming SIP services. The SIP Express Router platform [10] 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 [13] 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 expressivity and simplicity. JAIN SIP [8, 17] is the standard Java interface to a SIP signaling stack. As described in Section 2, it provides a powerful solution for developing SIP services. However, programmers still have to deal with protocol intricacies. None of these platforms and frameworks achieve both robustness and expressivity, to the level found in SPL.

Domain-specific languages (DSLs) have been studied for many years and have shown benefits in terms of expressiveness, conciseness, safety and performance [20]. They have been successfully designed, implemented and used, both academically and industrially, in a variety of areas including component configuration [7], financial products [2], networking [18], and operating systems [12, 19].

Session-based DSLs have been studied mainly in the context of HTML-based Web services. Mawl [3, 11] defines a service as a set of sessions, each of which is a possible service entry-point. The <bigwig> [5] language was later introduced to show how the Mawl approach could be extended with sophisticated static analyses yielding strong safety guarantees about the behavior of Web services [4, 16]. Although successful, these approaches are limited to HTTP which is a stateless protocol that is much simpler than SIP.

8 Conclusion

In this paper, we have argued that the scope and underlying technologies of the telephony domain make software engineering of services an overwhelming challenge. To take up this challenge, we have proposed a DSL 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 DSL named SPL that offers high-level notations and abstractions for service development. This language hides the subtleties of a SIP platform, making programs more concise than their GPL counterparts, without sacrificing expressivity. SPL has furthermore been formalized, enabling reasoning about telephony services.

Compared to GPLs, the high-level nature of SPL makes control and data flow of services explicit, both locally to a service method and globally to a session. This high-level nature enables a number of properties to be verified at the level of the telephony domain, the SIP protocol, the SIP platform and the service. These properties cannot be guaranteed in general with a GPL.

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.

We are exploring a number of avenues for further work. Future targets for the SPL compiler include Live Communications Server (Microsoft) and SER. Live Communications Server supports a rich API similar to that of JAIN SIP and relies on C#. SER is a low-level platform intended to be used at the network core, where performance is critical. Optimizations in terms of both space (state) and time are thus critical in this setting. This work would be followed by a performance study to assess these optimizations.

We are also investigating the development of large services, in areas such as company hotlines and telemarketing. Finally, we are exploring other ways to program in SPL beyond text. In particular, we are looking at a visual programming approach that would allow non-programmers to create services in an intuitive manner.

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*, 2005. To appear. Extended version available as the technical report BRICS RS-04-28.

[2] B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language describing financial products. In *IEEE Workshop on Formal Methods Application in Software Engineering*, pages 6–13, Apr. 1995.

[3] D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: a domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May/June 1999.

[4] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pages 221–231, June 2001.

[5] C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2), 2002.

[6] 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.

[7] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[8] J. Deruelle, M. Ranganathan, and D. Montgomery. Programmable active services for JAIN SIP. Technical report, National Institute of Standards and Technology, June 2004.

[9] 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.

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

[11] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia services. *World Wide Web Journal*, 1(1), January 1996. O'Reilly & Associates. Proc. 4th International World Wide Web Conference, WWW4.

[12] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30, San Diego, California, Oct. 2000.

[13] Microsoft. *Live Communications Server 2005 Enterprise Edition Deployment Guide*, Nov. 2004.

[14] A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. RFC 3265, IETF, June 2002.

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

[16] A. Sandholm and M. I. Schwartzbach. A type system for dynamic Web documents. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00*, January 2000.

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

[18] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, IN, Oct. 1998.

[19] S. Thibault, R. Marlet, and C. Consel. Domain-Specific Languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May 1999.

[20] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399