# Language Technology for Internet-Telephony Service Creation

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

* Department of Telecommunications
LaBRI - INRIA, France
Email: {burgy,consel,latry,palix,reveillere}@labri.fr

† DIKU
University of Copenhagen, Denmark
Email: julia@diku.dk

***Abstract*— Telephony is evolving at a frantic pace, critically relying on the development of services to offer a host of new functionalities. However, programming Internet telephony services requires an intimate knowledge of a variety of protocols and technologies, which can be a challenge for many programmers. Furthermore, because telephony is a resource heavily relied on, programmability of telephony platforms should not compromise their robustness.**

**This paper presents an approach to creating telephony services that builds on programming language technology (*i.e.,* language design and implementation, language semantics, and program analysis). We have developed a language, named *Session Processing Language* (SPL), that offers domain-specific constructs, abstracting over the intricacies of the underlying technologies. By design, SPL guarantees critical properties that cannot be verified in general-purpose languages. SPL relies on a Service Logic Execution Environment for SIP (SIP-SLEE) that introduces a design framework for service development based around the notion of session.**

**SPL and SIP-SLEE have been implemented and they are now being used to develop and deploy real services, demonstrating the practical benefits of our approach.**

## I. 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 organizing the telephone communications within a small business to telemarketing centers.

A variety of approaches have been proposed for developing services targeting SIP-based systems. Many of these approaches are based on general-purpose programming languages such as C, Java and C#. Platforms that rely on these languages offer the programmer a powerful but complicated access to their advanced functionalities. Although expressive, these approaches can not enforce the safe execution of a service. To overcome these limitations, some platforms introduce a restricted scripting language (*e.g.* CPL [1] or LESS [2]) for programming a service. However, services written using these scripting languages are usually limited to coarse-grained processing or dedicated treatments. To combine expressivity and simplicity, some platforms like the Microsoft Live Communication Server introduce hooks to shift the processing of a SIP message from a restricted scripting language (MSPL [3]) to a powerful API (C#).

In practice, most telephony platforms offer solutions for enabling service programming. However, such solutions often result into very large and complex APIs for providing both in-depth and in-breadth access to the platform. APIs also offer little support for structuring services or for managing service data. Finally, this openness comes at the expense of the robustness of the underlying platform, which can disrupt such a basic resource as telephony.

Our goal is to enable a programmer to concentrate on defining *what* the service logic should do as opposed to dealing with every single detail of *how* to implement it.

### *This paper*

This paper presents a Domain-Specific Language (DSL) [4], [5], named *Session Processing Language* (SPL), whose goal is to ease the development of telephony services without sacrificing safety. SPL relies on a Service Logic Execution Environment for SIP (SIP-SLEE) that provides a design framework for service development based around the notion of session. The design of SPL results from a thorough analysis of the requirements for a language for programming Internet telephony services. 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 that cannot be verified in general-purpose languages (GPLs), by introducing domain-specific concepts and semantic restrictions.

The rest of this paper is organized as follows. Section II defines requirements for a language dedicated to programming telephony services. Sections III and IV present our approach, including the SIP-SLEE and the SPL language. Section V describes SPL safety properties. Finally, Section VII concludes and provides some directions for future work.

## II. Requirements

Because telephony is heavily relied on, a platform which allows services to be executed raises a variety of issues that must be resolved [6]. Let us list some of these issues.

- What is the service programmer community (*e.g.*, administrator, end-user)?

- What information about the protocol messages are provided to the service?
- What level of control does the service have over the server's execution?
- What are the restrictions on the resources available to the service?

To address these issues, requirements for service programming have been already partially defined [6]. Based on these results, we have recently conducted a study of different existing SIP platforms and the paradigm they introduce to enable the development of telephony services [7]. This analysis is based on our experience on designing and implementing domain-specific programming languages (DSLs).

DSLs offer a solution to reconcile the expressivity of GPLs and the simplicity of scripting languages, while improving safety. A DSL is a programming (or specification) language dedicated to a particular domain or problem. The authors and others have successfully developed and used DSLs in various domains such as telephone switching systems [8], [9], protocols [10], operating systems [11], device drivers [12], and routers in networks [13]. As a result of these successes, DSLs have recently received a lot of attention from both the research and industrial communities [14].

Leveraging on DSL experience, we have revisited existing requirements for a language dedicated to telephony service creation.

*a) Multi-purpose language:* An important question for a telephony platform is whether or not untrusted users should be allowed to write and deploy services. Evidently, if untrusted users are allowed to develop services, they should be provided with a programming language that prevent a faulty or malicious service from crashing the platform. However, a proliferation of different languages may be disconcerting for the service developer.

Therefore, a key challenge is to introduce a programming language that is suitable to write both end-user services and network services, while taking into account the level of trust of the service developer.

*b) Abstraction level:* High-level abstractions enable the programmer to ignore the intricacies of both the platform and the underlying protocols. In doing so, an entire class of errors can be eliminated. The resulting services are easier to read, develop and maintain. By offering high-level abstractions, a domain-specific language is more accessible to users with limited programming experience and little domain expertise.

*c) Verifiability:* The safety of services depend on domain-specific properties that can be identified at four different levels: (1) telephony domain, (2) SIP signaling protocol, (3) SIP signaling platform, and (4) telephony services. Checking these properties in a service is a major step towards ensuring its safe execution.

*d) Analyzability:* Introducing programmability in a domain such as telephony raises a number of challenges. One of these challenges relates to the cost of services that are deployed on a platform. The cost of a service, expressed in terms of resource usage, serves a number of purposes including the admission control of a service, the platform configuration, and the definition of billing policies.

Providing a formalized semantics of a programming language dedicated to telephony services enables reasoning about services. Domain-specific properties related to resource consumption should be automatically verified by program analyses.

*e) Expressivity and usability:* Ensuring the safe execution of a service written in a given programming language must not be done at the expense of the expressivity and usability of that language. Service developers should be able to use mainstream language features such as user-defined types and control constructs. Essential features, such as variables, should not be excluded to ease the analyzability of the language.

## III. Service Logic Execution Environment

This section presents a service logic execution environment for SIP, named SIP-SLEE, providing the programmer with a high-level interface dedicated to telephony service development. A key contribution of the SIP-SLEE is the introduction of a domain-specific notion, named *session*, that represents a design framework for service development.[1] A session consists of operations and a state. We examine each of these components.

### A. Operations

To raise the level of abstraction, the SIP-SLEE 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-SLEE if its meaning is unambiguous (*e.g.,* `ACK`). Some requests are, however, context sensitive, and require interpretation. For example, the SIP request `INVITE` either initiates a dialog or, if used in the context of an existing dialog, modifies dialog characteristics. The SIP-SLEE thus refines a SIP `INVITE` request as either the control method `INVITE`, in the former case, or the control method `REINVITE` in the latter. As a result, 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.

### B. Partitioning of Operations

Existing SIP APIs do not support, or even suggest, an approach to developing services. These APIs represent a direct mapping

[1]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.

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 life-cycle: creation, confirmation, modification, termination.

The SIP concepts subscription and registration also have a life-cycle 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 life-cycle of the various design abstractions. They are thus classified as initial (creation), medial (confirmation and modification) and final (termination).

### C. 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-SLEE enables attaching a state to a dialog, a subscription, a registration or a service, and managing this state across the associated life-cycle. In doing so, 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 life-cycle 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.

## IV. The Session Processing Language

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-SLEE. 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/`.

### A. Organizing Sessions Into a Hierarchy

The syntax of an SPL service reflects the SIP-SLEE session structure. Each kind of session[2] 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 1, that implements a counter service. 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.

[2]Recall that a session in SPL can eiher be a dialog, a subscription, a registration or a service.

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

    registration {
      int cnt;

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

      void unregister() {
        log (cnt);
      }

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

Fig. 1. 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 1, 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. The dialog block only declares an `INVITE` handler. When an incoming call is rejected by the user, this handler increments the `cnt` variable, which is defined in the ancestor session (*i.e.,* the registration block).

As illustrated by the counter service shown in Figure 1, SPL does not require explicit state manipulation. Specifically, the `cnt` variable is defined and used as in any programming language. Such a variable results in the creation of a state object; variable manipulations are automatically mapped by SPL into access, save and restore operations.

### B. 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. Then, some computation must be performed 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-SLEE the current code pointer and state. When the corresponding response is received, the SIP-SLEE restores the code pointer and state of the service, and execution of the handler continues.

Notice that `forward` can be invoked with or without a URI. When no URI, is provided, the current request is forwarded to the the original destination (*i.e.,* Request-URI as defined in the SIP RFC 3261 [15]).

The `INVITE` handler in Figure 1 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.

### C. 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, *i.e.*, 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.

The branch mechanism is a built-in language construct that enables inter-handler control flow information to be determined accurately. If session variables and conditionals were used to encode branches, inter-handler control flow information would loose considerable accuracy. This situation would disable a number of existing correctness verifications.

```
1  service hotline_info {
2   processing {
3    type hotliner_t uri name; time t_call; int ticks;;
4    ...
5
6    registration {
7     ...
8     response incoming REGISTER(request rq) {...}
9     void unregister() {...}
10
11    dialog {
12     hotliner_t callee;
13     time t;
14
15     response incoming INVITE() {
16      if (TO == 'sip:hotline@domain.com') {
17       foreach (h in hotline_reg) {
18        if (get_status (h) == AVAILABLE) {
19         response r = forward h.name;
20         if (r == /SUCCESS) {
21          callee = h;
22          hotline_reg.remove (h);
23          return r branch hotline;
24         }
25        }
26       }
27       return forward 'sip:voicemail@domain.com';
28      }
29      else {
30       response r = forward;
31        if(r == /SUCCESS) {
32         hotline_reg.remove(TO);
33         return r branch private;
34        }
35        return r;
36       }
37      }
38      void incoming ACK() {
39       branch hotline {
40        t = get_time();
41        set_status (h, PHONE);
42        return;
43       }
44       branch private {...}
45       branch default {return;}
46      }
47      response BYE() {
48       branch hotline {callee.t_call += get_time() - t;
49                       return forward; }
50       branch default {return forward;}
51      }
52      ...
53 }}}}
```

Fig. 2. A hotline example

Branches are illustrated in Figure 2 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.

### D. Service development using SPL

We have implemented and deployed the first version of the SIP-SLEE presented in Section III. This implementation has been developed in Java and is based on a JAIN-SIP stack [16]. We have also developed an interpreter for the SPL

language. These components have been integrated to the SIP-based telephony system of our university.

A variety of services have been written in SPL for the department of Telecommunications of our university. In these experiments, SPL has demonstrated its usability and ease of programming. For example, a service has been defined for the department secretary. This service offers call waiting implemented with a queue of calls where each call is played a waiting message, periodically updated with the expected remaining time before being connected. This service consists of only about 100 lines of SPL; the sources are available at the SPL web site, `http://phoenix.labri.fr/software/spl/`.

## V. Safety Properties

The telephony domain imposes stringent safety and robustness requirements. A service should not itself incur runtime errors and should respect the underlying protocol. We consider some kinds of errors that can occur when programming SIP services with existing SIP APIs, and show how SPL has been designed either to prevent these errors outright or to enable static verifications that detect these errors in SPL services.

We have formally specified the semantics of SPL, enabling a precise definition of its interaction with the SIP-SLEE. This formal definition serves as a foundation for defining program analyses.

*Erroneous call processing:* A SIP service must ultimately perform some signaling action, such that no call is lost. Furthermore, the treatment of each message must be compliant with the SIP RFC 3261 [15]. For example, when a handler successfully forwards a message, it cannot then return an error. For another example, a handler is dedicated to a unique kind of request (*e.g.,* `INVITE`) and thus cannot forward a message of another kind (*e.g.,* `ACK`).

In SPL, signaling actions are associated with explicit keywords, such as `forward` for forwarding a request and `/SUCCESS` for matching a success response, allowing the SPL verifier to straightforwardly check that every execution path through a handler performs at least one signaling action, and that these signaling actions are coherent with each other. Furthermore, the SPL language prevents changing the method name when forwarding a request; the only argument to `forward` is the destination, leaving the request structure, and thus the method name, implicit.

A SIP message contains a number of headers, of which some are optional or read-only. Furthermore, as SIP messages are implemented as text, the underlying implementation of all headers is as strings, although a header may have a more intuitive meaning as *e.g.* an integer or a URI. Errors may include accessing a header that is not present in the current message, trying to modify the value of a read-only header, interpreting a header value as the wrong type of value, or writing the wrong type of value to a header. The SPL language prevents access to a header that is not present, via a construct (not shown in the examples of this paper, for space reasons) that combines both a check for the presence of the header and access to the header value. SPL also provides a specific construct (again not shown) for updating header values. In this case, the SPL verifier checks that uses of this construct only mention writable headers. Both constructs allow for headers to be treated as string-typed or to be accessed using more intuitive types; in the latter case the SPL verifier checks the validity of the type coercion.

*Erroneous state and control management:* A SIP service typically does some initial processing of a request and then forwards the request to one or more parties. Typical SIP APIs separate this service logic into separate entry points for request and response processing. This strategy breaks up the treatment of a given method, making it difficult to follow the service logic, and implies that state, that is needed across the forwarding of a request, must be saved and restored manually, a tedious and error-prone operation. In SPL, request and response processing are contiguous, within a single unit, with a forward operation being no more disruptive to the program structure than an ordinary procedure call. Local variables are implicitly saved and restored across a `forward`. Furthermore, SPL allows variables to be associated with an entire service, a registration, or a dialog, transparently managing the access to these variables across method invocations, thus ensuring that this data is manipulated in a consistent way.

*Erroneous resource management:* SIP APIs based on general-purpose languages do nothing to protect against safety errors that can occur in these languages. For example, APIs do not protect against infinite loops, and APIs based on C do not protect against out-of-bounds accesses to data structures or accesses to freed data. SPL allows only bounded loops, as illustrated by the use of `foreach` in Figure 2 (line 16), and includes appropriate checks on data structure accesses, as found in Java. Furthermore, SPL has no mechanism for dynamically allocating data, ensuring that service execution fits within a known memory bound.

## VI. Assessment

Many call processing languages have been proposed to enable service programming. In this section, we present a comparative study between SPL and four different scripting languages, namely CPL [1], SCML [17], LESS [2], and CCXML [18]. This study shows that SPL is the only language that fully addresses the requirements presented in Section II.

The expressiveness of LESS and CPL have been intentionally reduced to make them accessible to end-users without programming background. SCML and CCXML require more expertise and thus target technical users.

With SPL, we are pursuing different goals in that we are raising the level of abstraction by introducing domain-specific notions, such as sessions, and domain-specific constructs, such as branches. By hiding some of the intricacies of the SIP protocol into appropriate language abstractions, SPL enables the programmer to focus on the essence of the service logic to be defined, rather than its implementation details. Therefore, this high-level language makes programming telephony services accessible to more programmers.

Compared to CPL, SCML, LESS and CCXML, LESS is the only language providing verifications that go beyond a syntactic pass of the source program. LESS addresses feature interaction problems and provide a detection mechanism that relies on program analysis. As described in Section V, SPL enables much more stringent safety and robustness properties to be checked. These properties are ensured by a variety of automatic program analyses that exploit the high-level domain-specific constructs of the language.

Most of existing scripting languages for programming telephony services are restricted in that they do not provide mainstream programming constructs such as loops and variables. Removing such language features prevent the service developers from implementing services that require elaborate processing and state management. As illustrated by the fragments of a hotline service in Figure 2, SPL provides most of the language features that exist in GPLs such as Java or C, while preserving the safety of services.

## VII. Conclusion and Future Work

In this paper, we have presented a service logic execution environment for SIP, providing the programmer with a high-level interface dedicated to telephony service development. This SIP-SLEE is centered around the notion of a session that structures the development of a service. Additionally, we have introduced a DSL named SPL that offers high-level notations and abstractions for service development. This language hides the subtleties of SIP platforms, 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.

We are now exploring a number of research avenues. We are curently investigating a program analysis for detecting feature interaction of SPL services. We are also developing large services, in areas such as company hotlines and telemarketing. Finally, we are studying other ways to program in SPL beyond text. In particular, we are working on a visual version of SPL whose goal is to enable non-programmers to easily create their services.

## Acknowledgement

This work has been partly supported by the *Conseil Régional d'Aquitaine* under contract 20030204003A and by the European Commission under the IST Integrated Project AMIGO.

## References

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

[2] X. Wu and H. Schulzrinne, "Programmable end system services using sip," in *Proceedings of The IEEE International Conference on Communications 2002*. IEEE, 2003.

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

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

[5] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, June 2000.

[6] L. J. Lennox, "Services for internet telephony," Ph.D. dissertation, Columbia University, Dec. 2003.

[7] L. Burgy, L. Caillot, C. Consel, F. Latry, and L. Réveillère, "A comparative study of SIP programming interfaces," in *Proceedings of the ninth International Conference on Intelligence in service delivery Networks (ICIN 2004)*, Bordeaux, France, Oct. 2004.

[8] N. Gupta, L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss, "Auditdraw: Generating audits the fast way," in *Proceedings of the Third IEEE Symposium on Requirements Engineering*, Jan. 1997, pp. 188–197.

[9] D. Ladd and C. Ramming, "Two application languages in software production," in *USENIX Symposium on Very High Level Languages*, New Mexico, Oct. 1994.

[10] S. Chandra and J. Larus, "Experience with a language for writing coherence protocols," in *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct. 1997.

[11] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel, "Microlanguages for operating system specialization," in *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*. Paris, France: Computer Science Technical Report, University of Illinois at Urbana-Champaign, Jan. 1997.

[12] 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*, vol. 25, no. 3, pp. 363–377, May 1999.

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

[14] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Aug. 2004.

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

[16] J. Deruelle, M. Ranganathan, and D. Montgomery, "Programmable active services for JAIN SIP," National Institute of Standards and Technology, Tech. Rep., June 2004.

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

[18] "CXML: W3C, "voice browser call control: CCXML version 1.0."." [Online]. Available: http://www.w3.org/TR/ccxml/