

# A DSL Paradigm for Domains of Services: A Study of Communication Services

Charles Consel and Laurent Réveillère

INRIA – LaBRI  
ENSEIRB – 1, avenue du docteur Albert Schweitzer  
Domaine universitaire - BP 99  
F-33402 Talence Cedex, France  
{consel, reveillere}@labri.fr  
Home page: <http://compose.labri.fr>

**Abstract.** The domain of services for mobile communication terminals has long become a fast-moving target. Indeed, this domain has been affected by a continuous stream of technological advances on aspects ranging from physical infrastructures to mobile terminals. As a result, services for this domain are known to be very unpredictable and volatile. This situation is even worse when considering services relying heavily on multimedia activities (*e.g.*, games, audio and/or video messages, etc.). Such an application area is very sensitive to a large variety of aspects such as terminal capabilities (graphics, CPU, etc.), bandwidth, service provider's billing policies, QoS, and user expectations.

To address these issues, we present a paradigm based on domain-specific languages (DSLs) that enables networking and telecommunication experts to quickly develop robust communication services. Importantly, we propose implementation strategies to enable this paradigm to be supported by existing software infrastructures.

Our DSL paradigm is uniformly used to develop a platform for communication services, named Nova. This platform addresses various domains of services including telephony services, e-mail processing, remote-document processing, stream processing, and HTTP resource adaption.

## 1 Introduction

Recent technological advances in physical infrastructures and terminals make it possible to offer a vast variety of communication services. In particular, many wireless technologies, like GPRS, UMTS, 802.11 and Bluetooth, ensure sufficient bandwidth for multimedia applications dealing with audio and video streams [1, 2]. Furthermore, handheld computers are getting smaller and smaller and are beginning to offer performance competing with low-end PCs in terms of graphical capabilities and computing power. Also, handheld computers and telephone terminals are converging, as suggested by the number of applications they both offer (schedules, address books, note books, etc.). These trends should rapidly turn these two devices into one.

In fact, looking at the future of telecommunications, the question is no longer when customers will have a communication terminal with enough bandwidth to run multimedia applications. As reported by several major telecommunication players [3], this situation will occur in the near future with UMTS terminals, or comparable technologies. Rather, the critical question is: *What services should a communication terminal offer?*

Not only is this question not answered today, but, as in any emerging area, the needs and expectations of customers are unpredictable and volatile.

*Services are unpredictable* because communication terminals offer possibilities that average customers do not actually grasp. In fact, the logic that dictates the dynamics of the market is not based on technical considerations. For example, SMS-based services represent a rather unexpected booming sector considering that typing messages on the primitive keyboard of a GSM phone is a notoriously laborious process. Yet, SMS messages have proved to be a lasting fad generating sizable revenues.

*Services are volatile* because the capabilities of communication terminals enable an endless set of potential services; to a large extent, human creativity is the limit in creating new services! As a consequence, offering a fixed set of services is a limiting factor to the dissemination of the technology. In fact, like services for Intelligent Networks, time-to-market will surely be a key competitive advantage when tracking customer needs for new services.

Unpredictability and volatility make the supply of new services vital. In the context of telecommunications, the platform owners should not constrain third-party service providers. Instead, they should encourage them to participate in the service creation process so as to increase and diversify the supply of new services. Currently, platforms are often closed to potential service providers because conventional models rely on controlling the market from the platform to the end-users. Beyond economical reasons, platform openness is prohibited because it compromises the robustness of the platform.

### **Our Approach**

To address the key issues of service creation, we introduce a paradigm based on domain-specific languages (DSLs). A DSL is developed for a domain of services. It provides networking and telecommunication experts with dedicated syntax and semantics to quickly develop robust variations of services within a particular domain. Because of a critical need for standardization, the networking and telecommunication area critically relies on a protocol specification to define a *domain of services*. Our approach aims to introduce variations in a domain of services without requiring changes to the protocol. Furthermore, the DSL is restricted so as to control the programmability of variations. Finally, we describe how the most common software architecture of the networking and telecommunication area, namely the *client-server model*, can be extended to support our DSL paradigm.

We use our DSL paradigm to develop the Nova platform that consists of different domains of services, namely, telephony services, e-mail processing, remote-document processing, stream and HTTP resource adapters. A DSL has been developed for each domain of services; the definition of this domain of services is itself derived from a specific protocol.

To illustrate our presentation, we consider services dedicated to access mailboxes remotely. Such services allow an end-user to access messages stored on a remote server. In fact, there are many ways in which these services can be realized. One domain of services is defined by the Internet Message Access Protocol (IMAP) [4, 5]. We show that variations need to be introduced in this domain of services to adapt to a variety of user requirements. We develop a DSL to enable service variation to be introduced without compromising the robustness of the e-mail server.

## Overview

Section 2 identifies the key requirements of a platform for communication services. Section 3 describes how to introduce and control programmability in such a platform. Section 4 presents strategies to introduce programmability in the client-server model. Section 5 shows how it scales up to a complete platform for communication services. Section 6 discusses lessons learned during the development of Nova. Section 7 gives concluding remarks.

## 2 Requirements

In this section, we present the key requirements that a platform should fulfill to successfully address the rapidly evolving domain of communication services.

**Openness.** Most families of services are bound to evolve, often rapidly and unpredictably in emerging domains such as multimedia communications. To address this key issue, a platform for communication services has to be open. This openness should enable a wide variety of services to be easily introduced. In fact, each community of users ought to be offered specific services, applications and ways of communicating that reflect their interests, cultural backgrounds, social codes, etc.

**Robustness.** An open platform for communication services is destined to be shared and fueled by as many service providers as possible. As in the context of software providers, service providers can either be companies, or individuals devoted to some community, as illustrated by the development effort of Linux. Like software, new services can only spread without restrictions if safety and security are guaranteed. For example, in the context of telephony services, a buggy or malicious call-processing service may crash (or compromise) the entire underlying signaling platform.

**Composability.** When developing a software system, a programmer typically delegates some treatments to specific software components. Similarly, when developing a sophisticated service, one would want to combine some sub-families of services. As a consequence, different services should not interfere and should be able to be combined and used intensively without degrading the performance of the platform.

**Scalability.** Our target platform should be scalable with respect to various aspects. It should be open enough to cover most emerging user needs; its robustness should be demonstrated in the context of a sizable community of service providers, and address the critical properties of each domain of services; finally, the platform should offer appropriate support for a wide range of communication services.

### 3 The DSL Paradigm

In this section, we present the main aspects of our DSL paradigm, starting from a protocol and ending with a DSL.

#### 3.1 From a Protocol to a Domain of Services

A protocol goes beyond the definition of the rules and conventions used by a server and a client to interact. A protocol is the outcome of a careful analysis of a domain of services. It introduces the fundamental abstractions of a target domain of services in the form of requests. These abstractions are parameterized with respect to specific client data. These parameters and the server responses suggest key data types for the domain of services. Examining a protocol to determine its domain of services is a valuable approach because communication services in networking and telecommunications are systematically based on a protocol.

For example, the Internet Message Access Protocol (IMAP) defines a domain of services for remote access to mailboxes [4, 5]. This domain of services aims to provide a user with access to messages stored on a remote server.

A protocol for communication services is traditionally implemented as a client-server architecture, typically over a network of machines. A client sends a request to the server to access a specific service. The server processes incoming requests and sends responses back to the corresponding clients. A protocol is detailed enough to allow a client to be implemented independently of a given server. In fact, a client commonly corresponds to different implementations so as to provide various subsets of the domain of services. These variations enable the client to adapt to specific constraints or user preferences. In contrast, although there usually exists different implementations of a server, these implementations tend to cover the entire domain of services.

In the IMAP case, there exists various client implementations that support the IMAP protocol, ranging from simple e-mail reading tools targeted toward

embedded systems (*e.g.*, Althea [6]) to integrated Internet environments (*e.g.*, Microsoft Outlook [7] and Netscape Messenger [8]) for workstations.

### 3.2 Variations of a Domain of Services

A given protocol may correspond to a variety of client and server implementations to account for customer needs. Although these implementations offer service variations on either a client or the server side, they must all be in compliance with the underlying protocol to make the client-server interaction possible. As such, these implementations can be viewed as a *program family*; that is, programs that share enough characteristics to be studied and developed as a whole [9]. Commonalities mainly correspond to the processing of the requests/responses that have formats and assumptions specified by the protocol. Variabilities on the server side consist of defining different semantics for client requests. On the client side, variabilities correspond to implementing different treatments for server responses.

In the IMAP case, our goal is to identify variations characterizing a scope of customized accesses to a mailbox. We explore these variations systematically by considering the various levels involved in accessing a mailbox, namely, an access-point, a mailbox, a message, and its fields (*i.e.*, message headers and message parts). At each level of this hierarchical schema, we study what programmability could be introduced with respect to the requests of the IMAP protocol. We refer to the programmability of each level as a *view*.

This hierarchical approach to views enables user preferences to be defined comprehensively: from general coarse-grained parameters, such as the terminal features, to the specific layout of a message field. A view at a given level may treat a message field as an atomic entity, *e.g.*, deciding whether to drop it. At another level, a view may trigger specific treatments for parts of a message field.

### 3.3 From Variations of a Domain of Services to a DSL

Enabling service variations to be introduced in a protocol relies on the ability to easily and safely express a variation. To this end, we propose to use DSLs as a programming paradigm. This kind of languages has been successfully developed and used in a wide spectrum of areas [10].

Many approaches, such as family analysis, have been developed to drive the design of a DSL with respect to a specific program family [11, 12, 9, 13]. These approaches aim to discover both commonalities and variations within a program family to fuel the design process of a DSL. A step toward a methodology for DSL development has been presented by Consel and Marlet [14]. Recently, this approach has been revisited to structure the development of DSLs on the notion of program family [15].

In the context of the IMAP case, we have designed a DSL, named Pems, that enables a client to define views on remote mailboxes, specifying how to adapt mailbox access to constraints or preferences like device capabilities and

available bandwidth. For example, one can filter out messages with respect to some criteria to minimize the length of the summary of new messages received.

```

view accesspoint PDA {
  Mobility = YES;
  Screen   = 320 * 240;
  Color    = NO;
  Bandwidth = 10MB/s;
  Mailbox_view = nomadic(1MB);
}

view mailbox nomadic(size s) {
  if (From == "joe@mail.fr")
    bind boss;
  if (Size > s)
    ignore;
  bind tiny;
}

view message boss {
  From as cst("The Boss!");
  Date;
  Subject;
  Body;
  Attachment[] as bwImages(30KB);
}

view field Attachment bwImages(size s) {
  if (Attachment.size > s)
    return "Attachment too big:" +
      Attachment.size;
  if (Attachment.type in "image/*")
    return blackwhite(Attachment.value);
  ignore;
}

```

**Fig. 1.** Pems example

As illustrated by the example shown in Figure 1, the Pems language makes it possible to define views at four different levels: access-point, mailbox, message, and message field. An access-point consists of a set of parameters such as the client terminal features, the characteristics of the link layer, and a mailbox view. A mailbox view aims to select the messages that belong to the view. It consists of a list of condition-action pairs. When a condition matches, the corresponding action is performed. An action can either drop the current message or assign it a category of messages for its processing. The message view defines a set of fields, relevant to the client, for a given category of messages. Also, a view may be assigned to some fields to trigger specific treatments. Finally, the field view aims to convert field values into some representation appropriate to the access-point.

The IMAP example illustrates how our DSL paradigm makes a protocol open to variations to cope with user needs. Thanks to the abstractions and notations provided by Pems, one can easily write customized services for accessing a mailbox. Moreover, various verifications can be performed on a service before its deployment to preserve the robustness of the underlying platform. In the IMAP case, the Pems compiler checks various program properties such as non-interference, resource usage, and confidentiality.

## 4 A Software Architecture to Support Programmability

Now that service variations can be introduced without changing the protocol, we need to study a software architecture that can support the programming of these variations. Because most protocols for communication services are implemented using a *client-server model*, we propose to examine strategies aimed to introduce

programmability in this model. These adaptations of an existing software architecture should demonstrate that our DSL paradigm is a pragmatic approach to introducing service variations. In fact, each of these strategies has been used in Nova as illustrated later in this section and in Section 5.

Although adaptations could either be done on the client side or the server side, we concentrate on the latter to relieve the client terminal from adaptation processing and the network from unnecessary data.

#### 4.1 Script-Enabled Server

Scripting languages are commonly used to introduce variations on the server side. Scripts (*e.g.*, CGI scripts [16]) can be parameterized with respect to some client data. However, this strategy is limited because scripting languages are often unrestricted, and thus, only the server administrator can introduce service variations. Such a limitation clearly contradicts our openness requirement.

One strategy to map our DSL paradigm into a script-enabled server is to compile a DSL program into a script, as proposed by the Bigwig project in the domain of Web services [17]. This strategy has two main advantages: it makes programmability more widely accessible without sacrificing robustness, and it allows the existing support for programmability to be re-used. Of course, not all servers are combined with a scripting language; and, even if they are, the purpose of this language may not coincide with the service variations that need to be addressed.

The script-enabled server approach has been used in Nova to introduce programmability in the domain of telephony services as described in Section 5.1.

#### 4.2 Proxy Server

An alternative approach to defining service variations consists of introducing a proxy server that runs client programs and invokes the unchanged server. Client programs are written in a scripting language or a general-purpose language. Robustness of the platform is guaranteed by the physical separation of the server and the proxy: they run on different machines. This approach still has a number of drawbacks. First, it consumes bandwidth for communications between the proxy and the server. Second, it requires very tight control of resource consumption (*e.g.*, CPU and memory) to prevent one script from interfering with others. Third, a buggy script can compromise the server by overflowing it with requests.

Our DSL paradigm could also be beneficial in the context of proxy servers. The idea is to have the proxy run DSL programs, as opposed to running programs written in an arbitrary language. Because the DSL programs are safe, the proxy can even run on the same machine as the one hosting the server, and thus eliminate network latency.

As described in Section 5.3, the HTTP resource adapters of Nova rely on the proxy server approach to introducing programmability.

### 4.3 Programmable Server

To further integrate programmability in the server, we proposed to directly make the server programmable [18]. To do so, our strategy consists of enabling the semantics of a request to be, to some extent, definable. The processing of a request can be seen as parameterized with respect to a service variation that takes the form of a DSL program. The DSL restrictions guarantee that service variations do not compromise server robustness. Also, in contrast with a proxy-based approach, this tight integration of service variations in the server minimizes performance overhead.

We have modified the implementation of an existing IMAP server to make it programmable. A client can introduce its own service variations in the form of a Pems program. This DSL program is deployed and run in the server, after being checked.

The implementation of Pems is traditional: it consists of a compiler and a run-time system. The compiler is a program generator that takes a Pems program and performs a number of verifications to fulfill the robustness requirements on both the server and client sides. The Pems program is then translated into C code. Finally, this C code is compiled and loaded into the server when needed.

An original IMAP server [19] has been made programmable so that code can be dynamically loaded to extend its functionalities. Yet, binding a service implementation to a particular context is a remaining key issue. Indeed, it is orthogonal to the software architecture used to support programmability. This issue is detailed elsewhere [18].

## 5 The Nova Platform

To assess our approach, we have used the DSL paradigm to develop a programmable platform for communication services, named Nova. It consists of a programmable server and a DSL for each target application domain. Five application domains are currently covered by Nova: e-mail processing, remote document processing, telephony services, streams, and HTTP resource adapters. Let us briefly present these different application areas.

### 5.1 Call Processing

Telephony services are executed over a signaling platform based on the *Session Initiation Protocol* (SIP). We have designed a dialect of C to program call processing services, named Call/C. In contrast with a prior language, called CPL [20], our DSL is a full-fledged programming language based on familiar syntax and semantics. Yet, it conforms with the features and requirements of a call processing language as listed in the RFC 2824 [21]. In fact, our DSL goes even further because it introduces domain-specific types and constructs that allow verifications beyond the reach of both CPL and general-purpose languages. The example shown in Figure 2 illustrates the use of the Call/C language to program

a call forwarding service. This service is introduced by defining a behavior for the *incoming* request<sup>1</sup> of the SIP protocol. When a call is received, the incoming entry point is invoked with information about the caller. In this call forwarding service, the incoming call is redirected to `sip:dana@labri.fr`. If this redirection is itself redirected, then the location of the new call target is analyzed. If the location is not a voice-mail address, then the redirection is performed. Otherwise, the call is redirected to the callee's voice-mail `sip:john@voicemail.labri.fr`.

```

response incoming(call in) {
  response res = forward(in, sip:dana@labri.fr);
  switch(res.kind) {
    case redirect:
      if (! match(res.contact, ".*@voicemail.*") {
        return forward(in, res.contact);
      }
    default:
      return forward(in, sip:john@voicemail.labri.fr);
  }
}

```

**Fig. 2.** Call/C example

The script-enabled server approach is commonly used for programming telephony services in a SIP-based signaling platform. Examples include SIP CGI, SIP servlets and Microsoft's proprietary SIP programming API.

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

Our current implementation of the Call/C language in Nova targets two very different programming layers, namely SIP CGI, with C as a scripting language, and SIP Servlets.

## 5.2 Remote Document Processing

Accessing a document stored on a remote server may involve various processing before getting the document in a format appropriate for a specific *sink* (*i.e.*, a local machine or a device). The target format could depend on a variety of parameters, such as the capabilities of the access-point and the available bandwidth. To address these issues, we have developed a simple protocol for remote-document processing, named RDP, and a language aimed to define both conversion rules and sinks for documents, called RDPL.

The RDP server facilitates the process of converting documents into forms in which they are required (for example, a 1024x786 *jpeg* image with 32-bit color to a 160x200 image with 256 colors suitable for a PDA). Besides, RDP enables

<sup>1</sup> Strictly speaking, a call is initiated with the request *Invite* of the SIP protocol.

the server to redirect documents to specific sinks (for example, a PDF file may be redirected to a printer, or a fax machine).

An RDPL program specifies two main parts, as illustrated by the example presented in Figure 3. The first part is a list of sinks and a definition of their capabilities (*e.g.*, overleaf printing in the case of a printer). The second part is a filter graph that defines the intermediate steps and the filters used for the format conversion of a document. Filters can be combined to perform a wide range of transformations.

```

SINK printer1 {
  Help      "Printer on the 1st floor";
  Interface printer1_spooler;
  Capa      Overleaf  bool;
  Help      Overleaf  "Print retro/verso?";
  Capa      Priority   integer;
  Help      Priority   "0: Normal 1: High";
  Format     ps;
  Spool     /var/spool/documentqueue;
}

Filters {
  txt ps  enscript;
  txt mp3 txt2mp3;
  jpg gif convert;
  jpg png convert;
  gif png convert;
  gif jpg convert;
  wav mp3 lame;
  mp3 wav mp3play_wrapper;
  wav ogg oggencode;
  ogg wav oggplay_wrapper;
}

Formats {
  ps, txt, mp3, jpg, gif, png, ogg, pdf, wav;
}

```

**Fig. 3.** RDPL example

### 5.3 HTTP Resource Adaptation

More and more devices are being used to access HTTP resources (*e.g.*, PDAs, cell phones, and laptops). Such devices have different capabilities in terms of memory, computation power, graphic rendering, link layer, etc. In addition to classical HTML pages, HTTP resources may also include sound, image, and video. We have designed and developed a language, called Hades, for specifying the adaptation of HTTP resources to the features of a target access-point.

The example shown in Figure 4 illustrates the use of Hades. It specifies that video content must be removed and that images are to be replaced by a link to an image converted into gray-scale *jpeg* format.

```

accesspoint PDA {
  Mobility = YES;
  Screen   = 320 * 240;
  Color    = NO;
  Bandwidth = 10MB/s;

  image {
    changeFormat("JPEG");
    grayscale();
  }
}

html {
  image {
    externalize ("[IMAGE]");
  }

  video {
    remove();
  }
}

```

Fig. 4. Hades example

The ICAP protocol [22] was designed to facilitate better distribution and caching for the Web. It distributes Internet-based content from the *origin* servers, via proxy caches (ICAP clients), to dedicated ICAP servers. These ICAP servers focus on specific value-added services such as access control, authentication, language translation, content filtering, and virus scanning. Moreover, ICAP enables adaptation of content in such a way that it becomes suitable for other less powerful devices such as PDAs and mobile phones.

Our implementation of the Hades language relies on the ICAP protocol. We have developed a compiler that takes an Hades program and generates code to be loaded in an ICAP server. The Squid Web proxy is used as an ICAP client to enable the HTTP resource adaptation specified by the Hades program.

#### 5.4 Stream Processing

The last application area covered by Nova is stream adaptation. We have developed a language to specify multimedia stream processing, named Spidle [23]. This language is used to program a server that adapts a stream to particular features of a target access-point.

```

filter RPE_Encoding {
  interface {
    stream inout bit[40][16] e;
    stream out bit[13][3] xMc;
    stream out bit[1][6] xmaxc;
    stream out bit[1][2] Mc;
  }
  instantiate {
    merger Padding pad;
    filter Weighting w;
    filter RPE_Grid_Selection gs;
    filter APCM_Quantization q;
    filter APCM_Inverse_Quantization iq;
    filter RPE_Grid_Positioning gp;
  }
}

[...]

[...]
map {
  e -> pad.si;
  pad.so -> w.e;
  w.x -> gs.x;
  gs.xM -> q.xM;
  gs.Mc -> gp.Mc;
  gs.Mc -> Mc;
  q.man -> iq.mant;
  q.exp -> iq.exp;
  q.xMc -> iq.xMc;
  q.xMc -> xMc;
  q.xmaxc -> xmaxc;
  iq.xMp -> gp.xMp;
  gp.ep -> e;
}
}

```

Fig. 5. Spidle example

The example shown in Figure 5 consists of a Spidle program that defines a network of *stream tasks*. *Flow declarations* specify how stream items flow within stream tasks (*i.e.*, graph nodes) and across stream tasks (*i.e.*, graph edges), as well as the types of these stream items.

A stream task can either be a *connector* or a *filter*. Connectors represent common patterns of value propagation. Filters correspond to transducers; they can either be *primitive* or *compound*. A primitive filter refers to an operation implemented in some other programming language. This facility enables existing filter libraries to be re-used. A compound filter is defined as a composition of stream filters and connectors. This composition is achieved by *mapping* the output stream of a task to the input stream of another task.

## 6 Assessment

In this section, we review the lessons learned from our experience in developing Nova. We provide some insights obtained from the study of the different domains of services supported by Nova. Some performance and robustness aspects are discussed and related to existing works. Finally, we address some of the issues raised by the introduction of domain-specific languages.

### 6.1 Introducing Programmability

The most favorable situation to introduce programmability is when a server is already programmable via a scripting language. A DSL can then simply be viewed as a high-level abstraction of an existing programming layer. This layer is used as the target of the DSL compiler, as shown in the case of Call/C. However, the compilation approach is only possible, if the target layer fulfills the programming requirements needed to express the desired service variations. Otherwise, a proxy-based approach can be used. This approach is interesting because, following the case of the scripting language, it does not entail changes in the origin server. This approach has been successfully used to write adaptations of HTTP resources in Hades. However, a proxy-based strategy is limited in that some functionalities require to directly manipulate the server state. Furthermore, introducing a proxy incurs either a CPU overhead, if the proxy runs on the server machine, or a network overhead, if the proxy runs on a remote machine. In the former case, an additional process is needed to adapt requests and responses, and computations may be wasted, if data are produced by the origin server but pruned by the proxy. In the latter case, communications between the server and the proxy generate network traffic that may increase latency. These potential problems motivate a third approach to introducing programmability: modifying the server to make it programmable. Beyond collapsing a server and a proxy, this approach enables programmability to reach potentially all of the functionalities of the server. Furthermore, experience shows that it does not incur significant overhead in terms of execution time, and does not introduce any network overhead, as discussed by the authors [18].

As can be observed, the DSL paradigm to programming servers is flexible and can adapt to existing infrastructures with techniques ranging from compiler retargetting to parameterization of server functionalities.

## 6.2 Performance

Traditional compilation techniques are applicable to DSLs. In fact, it has been shown that DSL features can enable drastic optimizations, beyond the reach of general-purpose languages [24].

In the IMAP case, we have conducted some experiments to assess the performance and bandwidth usage of the programmable server approach. The results of these experiments show that no significant performance overhead is introduced in the programmable IMAP server, compared to its original version [18]. In the Call/C example, we showed that DSL invariants enabled many target-specific optimizations when generating code for a given programming layer [25].

## 6.3 Robustness

Our approach assumes that service developers may not be trusted by the owner of the server. Furthermore, when the domains of services involve ordinary users, as in the case of Nova, the developer may not be an experienced programmer. As a consequence, the DSL should guarantee specific properties so as to both preserve the integrity of the server and prevent a faulty service to corrupt or destroy user data. Notice that, most of these requirements would not be achievable in the context of general-purpose languages because of their unrestricted expressiveness [14].

In the application area of stream processing, Consel *et al.* [23] showed that the degree of robustness of a Spidle program goes beyond what can be achieved with an equivalent program written in a general-purpose language. For example, stream declarations are checked to guarantee that the composition of stream procedures are compatible with respect to both their types and the flowing direction of stream items.

## 6.4 Cost of DSL Introduction

Let us now present the key issues raised by using DSLs as a paradigm to address domains of communication services.

*Cost of DSL invention.* In our approach, a DSL is developed for each target domain of services. This systematic language invention introduces a cost in terms of domain analysis, language design and implementation. Traditionally, domain analysis and language design require significant efforts. In contrast, our approach relies on a key existing component: a protocol in the target domain. This protocol paves the way for the domain analysis by exposing the fundamental abstractions. It also suggests variations in the domain of services, feeding the language design process.

*Learning overhead.* Some effort is usually required for learning a new language. However, unlike a general-purpose language, a DSL uses domain-specific notations and constructs rather than inventing new ones. This situation increases the ability for domain experts to quickly adopt and use the language [14].

*Programming interface.* The five DSLs currently included in Nova have a textual representation and a C-like syntax. Yet, writing programs in these DSLs can use other representations. For example, one could use an existing framework such as XML to reduce the learning curve for users familiar with these notations. Also, textual forms could be abstracted by visual forms. That is, a DSL may have a graphical representation and be supported by an appropriate graphic-user interface. For example, we have developed a graphical front-end for the development of Spidle programs.

## 7 Conclusions and Future Work

Communication services are well-known to be very unpredictable and volatile. To cope with features, we propose a paradigm based on domain-specific languages. This paradigm enables networking and telecommunication experts to quickly develop variations for a domain of services defined by a protocol. We have used this paradigm to uniformly develop a programmable platform for communication services, named Nova. Our paradigm relies on the client-server architecture to support a protocol, as is usual for communication services. We proposed various strategies to introduce programmability in this software architecture. The dedicated nature of the DSL enables critical properties to be checked on DSL programs so as to ensure the robustness of the underlying server.

Nova is currently targeted at five application areas, namely, telephony, e-mail, remote document processing, stream processing, and HTTP resource adaptation. This preliminary work suggests that our approach scales up in terms of application areas that it can cover. Moreover, the DSL approach has shown to be very effective for making properties, critical to a domain, verifiable by design of the language.

We have started studying the composability of our approach. The study has been conducted in the context of the programmable IMAP server. This server has been combined with the RDP server to transform message fields (typically attached documents) to a format that fits the capabilities of a device. These preliminary studies showed that composing programmable servers does not raise any difficulties. In fact, this is not surprising since the client-server architecture has long proved that it is composable. And, in essence, our approach just adds a setup phase to deploy a DSL program in the server. Once the deployment is done, the server behaves as usual, processing requests and sending responses.

### Acknowledgment

This work has been partly supported by the *Conseil Régional d'Aquitaine* under Contract 20030204003A.

## References

1. Ghribi, B., Logrippo, L.: Understanding GPRS: the GSM packet radio service. *Computer Networks* (Amsterdam, Netherlands: 1999) **34** (2000) 763–779
2. Mock, M., Nett, E., Schemmer, S.: Efficient reliable real-time group communication for wireless local area networks. In Hlavicka, J., Maehle, E., Pataricza, A., eds.: *Dependable Computing - EDCC-3*. Volume 1667 of *Lecture Notes in Computer Science.*, Springer-Verlag (1999) 380
3. O'Mahony, D.: Umts: The fusion of fixed and mobile networking. *IEEE Internet Computing* **2** (1998) 49–56
4. IETF: Internet Message Access Protocol (IMAP) - version 4rev1 (1996) Request for Comments 2060.
5. Mullet, D., Mullet, K.: *Managing IMAP*. O'REILLY (2000)
6. Althea: An IMAP e-mail client for X Windows. <http://althea.sourceforge.net> (2002)
7. Microsoft: Microsoft Outlook. <http://www.microsoft.com/outlook> (2003)
8. Netscape: Netscape Messenger. <http://wp.netscape.com> (2003)
9. Parnas, D.: On the design and development of program families. *IEEE Transactions on Software Engineering* **2** (1976) 1–9
10. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* **35** (2000) 26–36
11. McCain, R.: Reusable software component construction: A product-oriented paradigm. In: *Proceedings of the 5th AiAA/ACM/NASA/IEEE Computers in Aerospace Conference*, Long Beach, California (1985)
12. Neighbors, J.: *Software Construction Using Components*. PhD thesis, University of California, Irvine (1980)
13. Weiss, D.: Family-oriented abstraction specification and translation: the FAST process. In: *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS)*, Gaithersburg, Maryland, IEEE Press, Piscataway, NJ (1996) 14–22
14. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In Palamidessi, C., Glaser, H., Meinke, K., eds.: *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*. Volume 1490 of *Lecture Notes in Computer Science.*, Pisa, Italy (1998) 170–194
15. Consel, C.: From a program family to a domain-specific language (2004) In this volume.
16. IETF: The WWW common gateway interface version 1.1. <http://cgi-spec.golux.com/nca> (1999) Work in progress.
17. Brabrand, C., Møller, A., Schwartzbach, M.I.: The <bigwig> project. *ACM Transactions on Internet Technology* **2** (2002)
18. Consel, C., Réveillère, L.: A programmable client-server model: Robust extensibility via dsls. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montréal, Canada, IEEE Computer Society Press (2003) 70–79
19. University of Washington: Imap server. <ftp://ftp.cac.washington.edu/imap/> (2004)
20. Rosenberg, J., Lennox, J., Schulzrinne, H.: Programming internet telephony services. *IEEE Network Magazine* **13** (1999) 42–49
21. IETF: Call processing language framework and requirements (2000) Request for Comments 2824.

22. IETF: Internet content adaptation protocol (icap) (2003) Request for Comments 3507.
23. Consel, C., Hamdi, H., Réveillère, L., Singaravelu, L., Yu, H., Pu, C.: Spidle: A DSL approach to specifying streaming application. In: Second International Conference on Generative Programming and Component Engineering, Erfurt, Germany (2003)
24. Eide, E., Frei, K., Ford, B., Lepreau, J., Lindstrom, G.: Flick: A flexible, optimizing IDL compiler. In: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, NV, USA (1997) 44–56
25. Brabrand, C., Consel, C.: Call/c: A domain-specific language for robust internet telephony services. Research Report RR-1275-03, LaBRI, Bordeaux, France (2003)