INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Generative Programming Approach To Developing DSL Compilers

Charles Consel — Latry Fabien — Laurent Réveillère — Pierre Cointe

N° 5550

Avril 2005

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5550--FR+ENG

A Generative Programming Approach To Developing DSL Compilers

Charles Consel * , Latry Fabien * , Laurent Réveillère * , Pierre Cointe †

Thème COM — Systèmes communicants
Projet Phoenix

Rapport de recherche n° 5550 — Avril 2005 — 21 pages

Abstract:
Domain-Specific Languages (DSLs) represent a proven approach to raising the abstraction level of programming. They offer high-level constructs and notations dedicated to a domain, structuring program design, easing program writing, masking the intricacies of underlying software layers, and guaranteeing critical properties.

On the one hand, DSLs facilitate a straightforward mapping between a conceptual model and a solution expressed in a specific programming language. On the other hand, DSLs complicate the compilation process because of the gap in the abstraction level between the source and target language. The nature of DSLs make their compilation very different from the compilation of common General-Purpose Languages (GPLs). In fact, a DSL compiler generally produces code written in a GPL; low-level compilation is left to the compiler of the target GPL. In essence, a DSL compiler defines some mapping of the high-level information and features of a DSL into the target GPL and underlying layers (*e.g.,* middleware, protocols, objects, ...).

This paper presents a methodology to develop DSL compilers, centered around the use of generative programming tools. Our approach enables to structure the development of a DSL compiler on facets that represent dimensions of compilation. Each facet can then be implemented in a modular way, using aspects, annotations and specialization. Because these tools are high level, they match the needs of a DSL, facilitating the development of the DSL compiler, and making it modular and re-targetable.

We illustrate our approach with a DSL for telephony services. The structure of the DSL compiler is presented, as well as practical uses of generative tools for some compilation facets.

Key-words: Compilation, Domain-Specific Languages, Generative Programming

* Author's current address: INRIA / LaBRI - ENSEIRB, 1 avenue du Dr Schweitzer, Domaine Universitaire, F-33405 Talence Cedex, France.

† Author's current address: École des Mines de Nantes, La Chantrerie - 4, rue Alfred Kastler, B.P. 20722, 44307 NANTES Cedex 3, FRANCE.

Une Approche de Programmation Générative au Développement de Compilateurs de DSL

Résumé :

Les langages dédiés (DSLs) représentent une approche avérée pour élever le niveau d'abstraction de programmation. Ils offrent des constructions haut-niveau et des notations dédiées à un domaine, structurant la conception d'un programme, facilitant son écriture, masquant les complexités des couches logicielles sous-jacentes, et garantissant certaines propriétés critiques du domaine.

À première vue, les DSLs facilitent un véritable mapping entre un modèle conceptuel et une solution exprimée dans un langage de programmation spécifique. Néanmoins, les DSLs compliquent le processus de compilation du fait du fossé du niveau d'abstraction qui sépare le langage source et le langage cible. La nature des DSLs rend leur compilation très différente de la compilation de langages généralistes (GPLs). En effet, un compilateur de langage dédié produit généralement du code écrit dans un langage GPL; la compilation bas-niveau est ensuite laissée au niveau du compilateur du GPL cible. Par essence, un compilateur de DSL définit un mapping des informations et caractéristiques haut-niveau d'un DSL à la fois dans le GPL cible et les couches sous-jacentes (*e.g,* middleware, protocoles, objets, ...).

Ce papier présente une méthodologie pour développer des compilateurs de DSL, centrée autour de l'utilisation d'outils de programmation générative. Notre approche permet de structurer le développement d'un compilateur de DSL en facettes, qui représentent des dimensions de compilation. Chaque facette peut ensuite être implémentée d'une manière modulaire, en utilisant les aspects, les annotations, et la spécialisation. Ces outils étant haut-niveau, ils correspondent au besoin d'un DSL, facilitant le développement d'un compilateur de DSL, et le rendant modulaire et re-targétable.

Nous illustrons notre approche avec un DSL pour les services de téléphonie. La structure du compilateur du DSL est présentée, ainsi que l'utilisation pratique d'outils génératifs pour certaines facettes de compilation.

Mots-clés : Compilation, Langages Dédiés, Programmation Générative

Contents

1 Introduction

"Generative software development aims at modeling and implementing system families in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages [6]".

At the design level, generative software development emphasizes the role of Domain-Specific Languages (DSLs) as a way to bridge the gap between high-level modeling and General-Purpose Languages (GPLs). Nevertheless, from the programming language viewpoint, there is a lack of methodology and a lack of tools to support the development of DSL compilers. The purpose of this paper is to apply some of the mainstream techniques, promoted by the generative programming community, to develop DSL compilers.

What is a DSL? From a programmer viewpoint, a DSL is typically created to model a program family [3]. The commonalities and the variabilities found in the target program family suggest abstractions and notations that are domain specific [5]. In contrast with GPLs *à la* Java or C++, a DSL has a narrow application scope and must be readable for domain experts.

In general, a domain-specific textual program is a concise set of high-level declarations, focusing on what to compute, as opposed to how to compute it. As an illustration, consider the telephony domain, and more specifically service creation. Conceptually, telephony services represent variations in creating, modifying and terminating a communication between parties. While this program family in a GPL covers the full implementation of services, its counterpart in a DSL corresponds to variations of service logic, abstracting over implementation details.

From a language designer viewpoint, a DSL makes domain-specific information an integral part of the programming paradigm. As such the programmer can be viewed as being prompted by the language to provide domain-specific information. This information may take the form of domain-specific types, syntactic constructs and notations. It serves domain-specific concerns, such as library interfacing, optimization, instrumentation, profiling and verification of the generated code. As of now, there are neither specific methodology, nor dedicated support tools, well suited to handle the compilation of both the high-level nature of a DSL and the richness of the built-in domain-specific information.

Challenges in DSL compilation. When mapping a DSL to a GPL, the higher level the DSL is, the more program generation is needed to bridge the gap with the target execution environment. Concretely, one program line written in a DSL commonly compiles into many lines in GPL. For instance, we have developed a DSL for telephony services, named SPL [1], that compiles into Java. An SPL program is on average 4 times more concise than its Java counterpart.

Not surprisingly, GPL-translated programs include rather large program templates. The process of generating these templates can be quite complex, relying on various conditions, and requiring a number of instantiations by computing and inserting constants. Without any dedicated tool support, this process can be quite laborious and error-prone. The resulting generator is often cumbersome and hard to debug. Additionally, the lack of tool support makes it difficult to have a modular treatment of the domain-specific concerns exposed by a program.

This paper. We propose a methodology to develop DSL compilers. The key idea of this methodology is to rely on generative programming tools [7]. These tools enable to model the high-level nature of DSLs, and the richness of the built-in domain-specific information, in terms of program generation. This modeling can be done in the context of various generative programming approaches. Our methodology is composed of two steps: compiling program logic and performing generative programming.

First, the DSL program logic is translated into an abstract GPL representation. This representation is abstract because it includes operations whose interpretation is not necessarily defined yet; it may thus not be executable. Although abstract, this translation generates a representation that encodes domain-specific information and that is structured on code regions of interest for further compilation treatment. In doing so, the representation is amenable to generative tools that represent the second step of our methodology.

Generative programming tools are used to define the compilation of domain-specific facets of a DSL in terms of program generation processes. A category of facets addresses the mapping of a DSL program into a target execution environment. A second category of facets is devoted to the compilation of language abstractions. A third category of facets defines code generation processes that are specific to the subject DSL program. This approach enables to modularize the program generation process of a DSL compiler. Furthermore, each generative programming approach provides a paradigm, associated abstractions and tools dedicated to a specific family of program generation. The compiler developer can thus choose the most appropriate generative programming approach for a given facet.

Our approach is illustrated by three generative programming approaches, namely AOP [12], annotations [2], and program specialization [4, 9]; conceptually others approaches can be considered. AOP is well-suited to introduce cross-cutting behaviors in GPL-translated programs (*e.g.,* prologue and epilogue code of API invocations). Annotations enable non-functional concerns to be introduced in the compilation process (*e.g.,* resource management). Program specialization can address optimization-oriented program generation (*e.g.,* customization of software components).

Refining facets further leads us to distinguish between functional and non-functional facets. On the one hand, the functional facets define program generation processes that make the GPL-translated program executable. On the other hand, non-functional facets enrich the language execution in terms of requirements like performance, reliability and security.

Contributions.

- **A methodology to develop DSL compilers.** We define a methodology to develop a DSL compiler. We introduce a GPL-translated abstract representation of the program logic. This representation is structured so that further compilation can produce a spectrum of implementation variations.
- **A novel use of generative programming tools.** Our methodology relies on a novel use of generative programming. This methodology facilitates the generative programming process required by DSL compilation: high-level generative programming paradigms can be used to modularly process domain-specific information and abstractions.
- **A case study.** We use our methodology to develop a compiler for a DSL dedicated to the creation of telephony services. We present aspects, annotations and specialization opportunities that model various compilation dimensions of this DSL.

Outline. The rest of this paper is organized as follows. Section 2 introduces our case study, a DSL for telephony services. Section 3 presents our methodology to develop a DSL compiler. Section 4 focuses on compiling the program logic. Section 5 and 6 presents the compilation of functional and non-functional language units. Finally, Section 7 discusses our approach, and Section 8 concludes.

2 Case Study

Our approach to developing DSL compilers is illustrated by a DSL to create telephony services. We choose the Session Initiation Protocol (SIP) [13, 14] as the underlying signaling protocol. This protocol is used for Voice over IP (VoIP) and third generation mobile phones. It is standardized by the IETF[1] and adopted by the ITU.[2]

SIP is a rapidly emerging protocol that places telephony into mainstream computer science. It is combined with a number of existing protocols to handle various aspects of communications (*e.g.,* transport and real-time streaming). It relies on the client-server model. SIP platforms provide rich programming interfaces in languages like C# and Java. The general-purpose nature of these programming interfaces make them very large and intricate to use. This situation is demonstrated by JAIN SIP, a standardized Java interface to SIP [8, 15]. JAIN SIP consists of 130 classes and more than 3000 methods.

The need for a DSL in this domain stems from three main reasons. First, modern telephony is a software intensive area because of the host of new functionalities it offers. Second, programming telephony services now requires extensive expertise in SIP and its companion protocols, distributed programming, networking, and SIP programming interfaces. Third, a study of existing telephony services shows that programming a service logic in a given

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

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

    registration {
      int count;

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

      void unregister() {
        log (count);
      }

      dialog {
        response incoming INVITE() {
          response resp = forward;
          if (resp != /SUCCESS) {
              return forward 'sip:secretary@company.com';
          } else {
            count++;
            return resp;
          }
}}}}}
```

Figure 1: The counter service in SPL

platform with a GPL is quite a laborious and error-prone process. It requires recurring code patterns as the prologue and epilogue of invocations of the SIP programming interface.

We have developed a DSL named Session Processing Language (SPL) that enables telephony services to be define concisely, using high-level abstractions and notations [1]. SPL enables the programmer to concentrate on the service logic, abstracting over low-level intricacies such as the protocol details and the underlying platform programming interface.

An SPL program defines a telephony service to which users can subscribe. The *session* is a key notion in SPL; it structures the development of a telephony service. A session consists of a set of handlers and a state. A handler defines a treatment for a protocol request or a platform event. A handler may be omitted, if no service logic is associated with it. A state allows some data to be maintained across a set of handlers. SPL offers different kinds of sessions that form a hierarchy. A simple counter service written in SPL, exhibiting this hierarchy, is displayed in Figure 1.

The innermost session is the *dialog*: it manages a communication between parties. A dialog is created by the `INVITE` request, confirmed by the `ACK` request, and terminated by the `BYE` request. A dialog session defines handlers for the SIP requests and platform events pertaining to the communication management. In the counter service, an `INVITE` handler is defined to process incoming calls. This handler systematically *forwards* (*i.e.,* routes) a call to the user who subscribed to this service. If this forward does not succeed (*e.g.,* the user is busy), the call is forwarded to a secretary. Otherwise, the call is accepted, a counter is incremented, and the response is returned to the caller. The session variable `count` used in this handler is defined in the session surrounding a dialog, namely the *registration* session. Such a session is created for each user of the counter service that registers on the SIP platform by sending the `REGISTER` request. This session defines a state that consists of a unique variable (`count`), initialized in the `REGISTER` handler. A user is unregistered either when his registration lease expires or when the `REGISTER` request contains a zero-lease. Both situations are viewed by SPL as a unique event named `unregister`, for which a handler can be defined in the SPL program. In our example, the `unregister` handler invokes a function that logs the counter, when the session is terminated. At the top of the session hierarchy is the *service* session. Such a session is created when the service is deployed on the platform and deleted when it is undeployed.

SPL abstracts over a number of protocol and platform issues among which is the statefullness of a transaction. Let us explain this issue because it illustrates the gap between SPL and a SIP platform. This issue will later be used in examples. A transaction consists of a request and the response it triggers. Existing SIP platforms provide separate interface entries for processing requests and responses. A telephony program may process a request and its response independently; such a transaction is said to be *stateless*. Alternatively, a telephony program may need to match a response with the original request to continue some treatment initiated when the request got processed. Such a transaction is said to be *statefull* because it requires the platform to retain enough information to link a response to the original request. SPL abstracts over these implementation details. As shown in the `INVITE` handler, when the `INVITE` request is forwarded, the response is treated within the same handler. Statefullness of the transaction is determined by analyzing the service. In fact, SPL offers statefullness throughout the session hierarchy. In our example, the `count` variable is used in both the registration and dialog sessions. When a SIP message is processed by an SPL program, its GPL-translated version executes some code to trigger the corresponding handler and to extract the appropriate state.

3 A Methodology to Develop DSL Compilers

Our starting point to develop a DSL compiler is to define a direct translation of the *program logic* into a GPL representation. This translation is direct in that it aims to generate abstract GPL code, not necessarily executable, where DSL mechanisms are not yet expanded but rather left uninterpreted. Although abstract, this translation encodes domain-specific information into the GPL representation, in a context where they can later be interpreted.

In our telephony case study, for example, while the routing of a SIP message in SPL is not concerned with statefullness, this property is explicitly encoded in the GPL-translated version.

The GPL-translated program logic can then be the input to various interpretations guiding two compilation dimensions: *functional* and *non-functional language units*. On the one hand, the functional units of the language are intended to complement the GPL-translated program logic to make it executable. On the other hand, the non-functional units of the language address implementation refinements or enrichments. The compilation of functional units varies with respect to both the target GPL and the target execution environment. In addition to these dimensions, the compilation of non-functional units may vary with respect to requirements like performance, reliability and security.

Whether or not functional, the compilation of DSL units can be decomposed into treatments that are inherent to either the target execution environment, the language or the program. These three categories of treatment are named *facets*.

The goal of the *execution environment facet* is to bridge the gap between the DSL execution model, possibly implicit, and the target execution environment. The DSL execution model is supposed to be high level and portable, abstracting over the intricacies of the target execution environment. The execution environment facet is intended to generate the necessary code to interface the GPL-translated program logic with the underlying layers. Considering the SPL case, the execution environment facet bridges the gap between the explicit event-handling architecture of JAIN SIP and the implicit SPL model based on request handlers. For example, default request handlers need to be introduced to create and delete session states, whenever an SPL program does not define handlers for the corresponding requests.

The *language facet* is concerned with the interpretation and expansion of language mechanisms, whether or not explicit in the GPL-translated program logic. Such a facet generates recurring code patterns intrinsic to the language. For example, routing operations of all SPL services need to be analyzed to determine their statefullness. This information is made explicit in the GPL-translated program logic by adding some information in each invocation of the routing operations.

Lastly, the *program facet* invokes compiler treatments that are specific to the subject program. For instance, in the case of SPL, the request handler necessitates specific state extraction operations, depending on what session variables occur. Such a treatment is specific to a given SPL service and thus part of the program facet.

4 Compiling The Program Logic

The goal of compiling the logic of a program is to produce a representation that abstracts over implementation details while being amenable to generative programming tools. To be applied successfully, generative programming tools generally introduce some constraints on the program structure and may require program generation parameters. Program structure is needed to make the GPL-translated program logic match the granularity of the program entities manipulated by the generative programming tools. The program generation parameters are typically needed to customize the program generation process or the execution generated program. These parameters may correspond to compilation data that are domain-specific information and are encoded in the translated program or/and in generative programming declarations (*e.g.,* an aspect declaration).

Our approach enables the program generation process to be modularized: each module is responsible for a “slice” of program generation needed to compile a given DSL. This modularization is particularly well-fitted to explore the implementation scope offered by the high-level nature of a DSL.

An example of a GPL-translated program logic is displayed in Figure 2. It is the SPL program (showed in Figure 1) compiled into Java for a JAIN SIP interface [15]. This interface is typical of a client-server model in that it requires a telephony service to implement a `SipListener` interface, providing `processRequest` and `processResponse` methods. Additionally, a method is declared to handle various platform timeouts. Also, private methods have been introduced to handle the registration and un-registration of the service owner, as well as the forwarding of call invitations (responses are treated in `processResponse`). As can be noticed in `handler_INVITE`, for example, the service logic has been straightforwardly translated into Java: expressions manipulating variables are reproduced verbatim; message routing is translated into invocations of `sendRequest` for the SPL `forward` and `sendResponse` for the returning responses.

Let us now examine how to compile the program logic for generative programming approaches.

4.1 Aspect-Oriented Programming

The AOP approach consists of a joint point language, to identify locations in the program execution, and an advice language, to define additional behavior at these locations [10].

Our goal is to produce a GPL-translated program logic that matches the expressivity of the joint point language. That is, to structure the translated program so that it matches the granularity of the joint point identification and the semantics of the associated advice. Conceptually, to fit the AOP approach, DSL compilation must introduce specific code structuring and communicate compilation data.

The GPL-translated program logic needs some structuring when some code is to be inserted at some program point. Identifying the target program point requires language entities such as a method definition or invocation, variable occurrences, and declarations. As a result, a sequence of commands may need to be placed into a method in a translated program to enable code to be inserted before, after, or around its execution. The top of Figure 2 shows private methods that have been introduced by the DSL compiler to easily insert code as the prologue and epilogue of invocations of SPL request handlers.

Compiling the program logic also involves collecting or computing information about the DSL program that needs to be made explicit in the translated program. This strategy separates the production of the information and its exploitation, leaving its interpretation to other compilation passes. In the context of AOP, passing information can either be made by adding extra arguments to operations or by introducing specific instance variables . The latter case is further discussed in Section 5.2.

In principle, functional DSL units correspond to aspects that can performed statically since, intuitively, these aspects refine the static compilation process. In contrast, non-functional units may compile into both static and dynamic aspects. Static aspects may be used to expand the implementation of specific operations, whereas dynamic aspects may define conditional monitoring actions.

```
public class Counter implements SipListener {
  [...]
  private void handler_REGISTER (Request rq, String method) {
    count = 0;
    sendRequest (false, rq_request);  /**  SPL, line 10  **/
  }
  private void handler_unregister (Request rq, String method) {
    local.log (count);  /**  SPL, line 14  **/
    sendRequest (false, rq_request);  /**  SPL: default behavior  **/
   }
   private void handler_INVITE (Request rq, String method) {
     sendRequest (true, rq_request);  /**  SPL, line 19  **/
   }

   public void processRequest (RequestEvent requestEvent) {
     String method = rq_request.getMethod();
     if (method.equals (Request.REGISTER)) {  /**  SPL, line 5  **/
       if (!registrar.hasExpiresZero (rq_request)) {
         if (!registrar.hasRegistration (rq_request)) {  /**  SPL, line 8  **/
           handler_REGISTER (rq_request, method);
         } else {  /**  SPL: default behavior  **/
           sendRequest (false, rq_request);
         }
       } else if (registrar.hasRegistration (rq_request)) {  /**  SPL, line 13  **/
         handler_unregister (rq_request, method);
       } else {  /**  SPL: default behavior  **/
         sendRequest (false, rq_request);
       }
     } else if (method.equals (Request.INVITE)) {  /** SPL, line 18  **/
       handler_INVITE (rq_request, method);
     } else {  /**  SPL: default behavior  **/
       sendRequest (false, rq_request);
     }
   }

   public void processResponse (ResponseEvent responseEvent) {
     String method = rs_request.getMethod();
     rs_responseCode = rs_response.getStatusCode();
     if (method.equals (Request.INVITE)) {  /**  SPL, line 18  **/
       if (rs_responseCode >= 300) {  /**  SPL, line 20  **/
         AddressFactory addressFactory = getAddressFactory();
         SipURI sipURI = addressFactory.createSipURI ("secretary", "company.com");
         rs_request.setRequestURI (sipURI);
         sendRequest (false, rs_request);  /** SPL, line 21  **/
       } else {
         count++;
         sendResponse (true, rs_response);  /**  SPL, line 24  **/
       }
     } else {  /**  SPL: default behavior  **/
       sendResponse (false, rs_response);
     }
   }

   public void processTimeout (TimeoutEvent timeoutEvent) {[...]}
 }
```

Figure 2: Java-translated program logic

A key advantage of our approach is to permit a DSL compiler to be designed and structured in terms of modules, that is, aspects. Each aspect defines a specific DSL behavior whose cross-cutting nature is genuinely managed by AOP.

4.2 Annotations

Annotations can be included in the translated program to make some information explicit as the DSL program gets mapped into a lower-level representation. Annotations are traditionally used as extra information describing non-functional language issues.

Like aspects, annotations enable to modularize the compilation of a DSL in that it introduces information that are later interpreted with respect to a given annotation processor.

Different sets of annotations have different goals. There may be annotations to make resource management explicit, or annotations intended to trigger check pointing of some state. When compiling the program logic, the aim is to generate annotations that are as self-contained as possible to minimize the work of the annotation processor occurring at a later phase. To do so, the location of an annotation is a key parameter. Also, arguments to the annotation may be needed as additional input to the annotation process.

4.3 Program Specialization

Program specialization is another use of a generative programming tool for DSL compilation. It addresses the optimization of DSL implementation building blocks. The idea is that part of a DSL can often be seen as a language gluing software components together. Because these components can be glued in a variety of contexts they must be highly generic. While this approach has obvious software engineering advantages, in practice, it may entail a significant performance penalty. To alleviate this problem, program specialization enables generic software components to be customized with respect to the context in which they are used. Because software components are invoked by compiler-generated code, the customization contexts can be determined by definition of the DSL compiler. Furthermore, since components are fixed, or slowly evolving, their *specializability* can be determined precisely. As a result, this program transformation can be made fully predictable, which is not the case for arbitrary written program, as like most program optimizations.

5 Compiling Functional Units

Compilation of functional units fits particularly well with AOP: it incrementally refines the semantics of language mechanisms, whether or not explicit in the source program. We do various forms of code expansion triggered by method names and instance variables.

We use a wide-spread and well-tested AOP tool, AspectJ [11], to define the compilation of various functional units needed for SPL. Limitations discussed in the following examples are not intrinsic to the AOP approach but are rather specific to AspectJ tool. These limitations are further discussed in Section 7.

5.1 Execution Environment Facet

The goal of functional execution environment facets is to bridge the gap between the DSL execution model and the target execution environment. In our case study, these facets are intended to generate the necessary code to interface the program logic with the underlying JAIN SIP platform.

```
public aspect Environment {

  public Request Counter.rq_request;
  public SipProvider Counter.rq_sipProvider;

  pointcut processRequest(): execution(public void Counter.processRequest (RequestEvent));
  before(RequestEvent rq_Event, Counter obj): processRequest() && args(rq_Event) && target(obj) {
    obj.rq_request = rq_Event.getRequest();
    obj.rq_sipProvider = (SipProvider) rq_Event.getSource();
  }
  [...]
}
```

Figure 3: An aspect for an execution environment facet

The aspect program presented in Figure 3 introduces code that implements the SPL model in terms of the JAIN SIP event-handling architecture. To do so, the inserted code extracts the current request message from the `event` object generated by JAIN SIP. Dispatch over the type of request can then be done to invoke the appropriate Java-translated SPL handler. In addition, code to extract the `sipProvider` object is generated to enable further processing of the SIP message (*e.g.*, message headers and transaction creation).

Thanks to the functional execution environment facet presented above, the Java-translated program logic does not have to deal with the intricacies of the underlying JAIN SIP platform. Such a strategy helps to isolate target-dependent program generation in compiler modules, corresponding to aspects.

5.2 Language Facet

Functional language facets are concerned with the interpretation and expansion of language mechanisms. In our case study for example, state management and statefullness of routing operations require to generate recurring code patterns. The corresponding aspects are discussed below.

State management. As illustrated in Figure 2, the Java-translated program logic does not include code for attaching a state to a session, and managing this state across the session life-cycle. For example, the state associated to a registration session needs to be created when processing a `REGISTER` request and deleted either when the request failed or the session ends. We have developed a language facet in AOP dedicated to state management. An excerpt of this facet is depicted in Figure 4.

```
public aspect Language {

  private boolean handler_REGISTER   = true;
  private boolean handler_REREGISTER = false;
  private boolean handler_unregister = true;

  pointcut register(): execution(private void Counter.handler_REGISTER (Request, String));
  before(Request rq, String method, Counter obj): register() && args(rq, method) && target(obj) {
    State state = new State();
    int ident   = obj.env.getId (obj.rq_request);
    obj.env.setEnv (ident, state);
    obj.state = state;
  }

  pointcut processRequest(): execution(public void Counter.processRequest (RequestEvent));
  before(Counter obj): processRequest() && target(obj) {
    if (!handler_REGISTER) {
      String method = obj.rq_request.getMethod();
      if (method.equals (Request.REGISTER)) {
        if (!obj.registrar.hasExpiresZero (obj.rq_request)) {
          if (!obj.registrar.hasRegistration (obj.rq_request)) {
            State state = new State();
            int ident   = obj.lib.env.getId (obj.rq_request);
            obj.lib.env.setEnv (ident, state);
            obj.state = state;
  }}}}}
  [...]
}
```

Figure 4: An aspect for a language facet (1)

The first advice specifies the code to execute for creating a registration state when processing a `REGISTER` request. However, since the handler for the `REGISTER` request is not mandatory in SPL, the corresponding method may not be present in the Java-translated program. If so, some code must be inserted to catch a `REGISTER` request and create the registration state. However, the pattern matching capability of the pointcut language does not permit to test the non-existence of a method invocation. To remedy this problem, this information is encoded as flags introduced into the aspect program. These flags enable the appropriate advice to be selected, as illustrated by the use of the `handler_REGISTER` flag.

Statefullness. In the Java-translated program logic, the first argument of the `sendRequest` method determines the statefullness of the routing operation. This information has been made explicit in the Java-translated program logic thanks to an analysis of the SPL program. The aspect program shown in Figure 5 describes the recurrent code fragment that needs to be executed when processing such an operation. In this example, calls to `sendRequest` are compiled into either stateless or statefull routing operations depending on the value of its first argument.

The use of functional language facets enables the translated program to be closer to the program logic. For example, state management requires a precise understanding of the SIP protocol to determine when sessions need to be created and deleted. Such intricacies are

```
public aspect Language {
  [...]
  pointcut rq_sendRequest(): call(public void Counter.sendRequest (boolean,Request)) &&
    (withincode(public void Counter.processRequest (..)) || withincode(* Counter.handler_* (..)));
  void around(boolean b, Request r, Counter obj): rq_sendRequest() && args(b,r) && target(obj) {
    try {
      if (b) {
        ClientTransaction ct = obj.rq_sipProvider.getNewClientTransaction (r);
        ct.sendRequest();
        return;
      } else {
        obj.rq_sipProvider.sendRequest (r);
        return;
      }
    } catch (Exception ex) {}
  }
  [...]
}
```

Figure 5: An aspect for a language facet (2)

factorized in the language facet, enabling the compiler writer not to address all the DSL implementation issues at once.

5.3 Program Facet

Functional program facets are concerned with the generation of code that is specific to a program. For example, creation and manipulation of the state attached to a session depends on the session variables and the handlers that use them.

Such a situation is illustrated by the SPL program shown in Figure 1, where the `count` variable is used in the `REGISTER` handler (line 9). This variable occurrence leads to the declaration of an aspect, displayed in Figure 5, that inserts the class definition `State` and the methods to access the instance variable. Furthermore, this aspect specifies that each occurrence of the `count` variable in the Java-translated program logic must be replaced by an access to the state.

Functional program facets allow to define DSL compilation at the granularity of a program, using implicit or explicit information from the source program.

6 Compiling Non-Functional Units

Non-functional DSL units are compiled by exploiting information that refines or extends the resulting program implementation. Just like functional units, compilation of non-functional units cover all of the DSL facets, that is, execution environment, language and program.

6.1 Execution Environment Facet

Program specialization [4, 9] has been successfully used to customize an execution environment according to a specific usage context. In our case study, the JAIN SIP platform is

```
public aspect Program {

  public int Counter.count;
  public class State implements Lib.State {
    private int count;
    void setCount (int x) { count = x; };
    int getCount () { return count; };
  }

  pointcut set_count(): set (int Counter.count);
  void around(int count, Counter obj): set_count() && args(count) && target(obj) {
    obj.state.setCount(count);
  }

  pointcut get_count(): get (int Counter.count);
  int around (Counter obj): get_count() && target(obj){
    return obj.state.getCount();
  }
}
```

Figure 6: An aspect for a program facet

responsible for invoking specific methods in the Java-translated program logic for processing requests and responses. When no method is defined in the Java program, the platform does not need to pass the SIP message to the SPL service. Instead, it can directly perform the default platform behavior. Such a filtering of messages is similar to packet filtering in networking where only packets of interest are channeled to the application layer. Program specialization has already been applied in this context [17] and has demonstrated its benefits. In our case study, a similar strategy would aim to specialize the message filtering of the JAIN platform with respect to request names of interest to a program logic.

This approach enables to automatically and systematically specialize highly generic software components, such as JAIN SIP components, according to a customization context derived from the program logic, such as an SPL service. Beyond performance, specialization opens opportunity to reduce the foot-print of a software layer.

6.2 Language Facet

Chander *et al.* [2] have recently proposed an approach to resource-bounds checking. Their approach limits the resource usage accordingly to a policy, specifying the resources that a program can use, along with the corresponding usage bounds. The key idea is to ensure that for each operation that consumes resources, an adequate amount is still available. They propose to annotate a program with a *consume* e command specifying that e units of resource are used, and with an *acquire* e command specifying that e units of resources must be reserved. One of the advantages of this approach is to make it possible to use a theorem-prover to prove that adequate checks are being performed to guarantee correct resource usage for a given program. The verifier is composed by two components: a safety condition generator that extracts logical predicates (safety conditions) whose validity implies resource-usage safety and a prover that prove the predicates.

```
service limit_forward {
  processing {
    uri<4> operators = <...>;
    uri<3> callers = <...>;

    registration {
      dialog {
        response incoming INVITE() {
          foreach (caller in callers) {
            if (FROM == caller)
              return forward;
          }
          return forward operators;
}}}}}
```

```
public void processRequest(RequestEvent requestEvent) {
 [...]
  if (method.equals (Request.INVITE)) {
   //@ acquire 4
   Header f_h = rq_request.getHeader(FromHeader.NAME);
   String FROM = f_h.toString();
   int i = 0;
   int size = callers.size();
   while (i < size) {
    String caller = (String)callers.get(i);
    if (FROM.compareTo (caller) == 0) {
      //@ consume 1
      rq_sipProvider.sendRequest(rq_request);
      return;
    }
    i++;
   }
   //@ inv(i <= 3, 3 - i)
   //@ consume 4
   rq_sipProvider.sendRequest(operators, rq_request);
   return;
  }
}
[...]
```

Figure 7: Annotations for a language facet

Figure 7 illustrates this approach. The left of this figure shows an SPL program where a list of *callers* is defined: three persons authorized to contact the service owner. If not authorized, a call is redirected to a list of four *operators* until one of them picks up the phone. In the telephony domain, the forwarding action represents a key resource because it triggers a chain of operations that may be costly in the telephony platform.

The Java-translated program logic is displayed in the right of Figure 7. Routing operations have been annotated, as well as the loop command with an invariant ($inv(i \leq 3, 3-i)$). Through annotation analysis, we can determine that this example requires, at worst, the reservation of four resource units (*acquire 4*), corresponding to the case where no operator picks up the phone. As a result, we can assure that enough resources have been reserved before the program execution. Note that our *consume* annotations are made explicit in the translated program but they could just as well be incorporated in library by extending the existing JAIN API.

This example illustrates the use of an existing tool to perform some non-functional processing of DSL programs. That is, resource-bounds checking is introduced by re-using an existing annotation language and underlying tools. In doing so, the DSL compiler writer is guided by the annotation language to determine the required non-functional information to be provided in the translated program. This strategy prevents him from re-doing an analysis of this non-functional domain. Furthermore, annotations allow a modularization of the compiler in that their processing is left to a later phase, performed by dedicated existing tools.

6.3 Program Facet

Unlike, functional *program facets* that define compiler treatments specific to a subject program, non-functional *program facets* correspond to information collected on a given program. If we consider the example shown in Figure 7, an extension of the annotation approach could collect the *acquire* annotations for each SPL handler, to compute the maximum number of routing operations performed by the SPL program. This number could then be used by a security policy of the execution environment. In doing so, some restrictions could be enforced on the number of routing operations performed by an SPL program, to preserve the platform performance. This enforcement can occur prior to the program execution with respect to currently available resources. This process could be seen as admission control. The approach could be extended to all of the resources consumed by a program. As a result, a telephony service would be accompanied by a list of the resources required for its execution.

Note that this program facet is different from the language facet, presented in Section 6.2. Indeed, the language facet did not address resource consumption globally to the service; it only ensured that a consumed resource had been previously allocated.

An important issue in the telephony domain is service billing. Examining the kind of compilation treatments to perform to introduce billing operations, one can observe that it amounts to defining non-functional aspects, analogous to monitoring activities (*e.g.,* logging). As an example, some timer could be enabled in the handler that starts a call session, and disabled in the handler terminating a call.

7 Discussion

Our methodology for DSL compiler development is to translate the logic of a program into a GPL representation that is amenable to generative programming approach. One of these approaches relies on AOP to introduce specific behaviors at some locations in the GPL-translated program.

Our methodology for DSL compiler development heavily relies on generative programming approaches and corresponding tools. In doing so, their features, or even limitations, need to be factored in when developing the compiler. Concretely, limitations regarding AOP were discussed earlier. In fact, they did not concern the approach but rather the AspectJ tool used for our experiments. For example, variable introduction is only possible at the level of a class, not inside a method. Moreover, aspects are context-insensitive in that they do not permit to directly manipulate the variables that are in the scope of a cross-cutting point. This feature would permit to define finer grained advice and improve the quality of the generated code.

The generality of meta-programming [16] makes it an alternative approach to AOP, as well as most other generative programming approaches. In this context, a DSL compiler resembles an interpreter annotated so as to execute the static language actions and to produce code for the dynamic language actions. Meta-programming tools give a fine-grained control over program generation, which can occur at any program point. A key issue that needs to be explored is how to modularize DSL compilation with meta-programming, to mimic what we do with AOP, annotations and program specialization.

8 Conclusion and Future Work

In this paper, we present a new methodology to develop DSL compilers. Our methodology is composed of two steps: compiling program logic and performing generative programming. Compiling a program logic produces a GPL-translated representation that abstracts over implementation details while being amenable to generative programming tools. These tools allow to model the high-level nature of DSLs, and the richness of the built-in domain-specific information, in terms of program generation.

Our approach enables to modularize the program generation process of a DSL compiler. Each generative programming approach provides a paradigm, associated abstractions and tools, dedicated to a specific family of program generation. The compiler developer can thus choose the most appropriate generative programming approach for a given compilation dimension.

We have used our methodology to develop a compiler for a DSL dedicated to the creation of telephony services. We present aspects, annotations, and specialization opportunities that model various compilation dimensions of this DSL. This case study demonstrates that a DSL can make use of generative programming approaches and techniques in very effective ways. In essence, a DSL exposes information about programs that can be mapped by our approach into the realm of generative programming.

Compared to traditional approaches for compiler development, our methodology enables to have a modular treatment of the domain-specific concerns exposed by a program. The resulting compilation process of a DSL is made simpler and less error-prone.

References

[1] L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, and L. Réveillère. Telephony Software Engineering: A Domain-Specific Approach. Research Report RR-5548, INRIA, Bordeaux, France, Apr. 2005.

[2] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proceedings of the 14th European Symposium on Programming (ESOP)*, volume 3444, pages 311–325. Springer-Verlag, 2005.

[3] C. Consel. *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, chapter From A Program Family To A Domain-Specific Language, pages 19–29. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004.

[4] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, Jan. 1993. ACM Press.

[5] 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.

[6] K. Czarnecki. Overview of generative software development. In *Proceeding of the Unconventional Programming Paradigm.* To appear as Springer LNCS (Fradet, P. editor), September 2005.

[7] M. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* 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] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353. Springer-Verlag, 2001.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag, 1997.

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

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

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

[16] W. Taha. *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, chapter A Gentle Introduction to Multi-stage Programming, pages 30 – 50. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004.

[17] S. Thibault, C. Consel, J. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, Sept. 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