10-2011

A Domain Specific Language for Enterprise Grade Cloud-Mobile Hybrid Applications

Ajith H. Ranabahu
Wright State University - Main Campus

E. Michael Maximilien

Amit P. Sheth
Wright State University - Main Campus, amit@sc.edu

Krishnaprasad Thirunarayan
Wright State University - Main Campus, t.k.prasad@wright.edu

A Domain Specific Language for Enterprise Grade Cloud-Mobile Hybrid Applications

Ajith Ranabahu
Kno.e.sis Center
Wright State University
Dayton OH, USA
ajith@knoesis.org

E. Michael Maximilien
IBM Research
San Jose CA, USA
maxim@us.ibm.com

Amit Sheth
Kno.e.sis Center
Wright State University
Dayton OH, USA
amit@knoesis.org

Krishnaprasad Thirunarayan
Kno.e.sis Center
Wright State University
Dayton OH, USA
tkprasad@knoesis.org

ABSTRACT

Cloud computing has changed the technology landscape by offering flexible and economical computing resources to the masses. However, vendor lock-in makes the migration of applications and data across clouds an expensive proposition. The lock-in is especially serious when considering the new technology trend of combining cloud with mobile devices.

In this paper, we present a domain-specific language (DSL) that is purposely created for generating hybrid applications spanning across mobile devices as well as computing clouds. We propose a model-driven development process that makes use of a DSL to provide sufficient programming abstractions over both cloud and mobile features. We describe the underlying domain modeling strategy as well as the details of our language and the tools supporting our approach.

General Terms

Cloud Computing, Domain Specific Language, Cloud-Mobile hybrid applications, Program generation, Programming abstractions

Keywords

cloud computing, domain specific languages, programming abstractions

1. INTRODUCTION

The innovative mobile startup company Foursquare[1] enables users to broadcast their location via their mobile devices, find where their friends are located, as well as interact with local businesses for incentives to visit their physical locations. Foursquare currently supports five mobile phone platforms and hosts their back-end services on the Amazon EC2 cloud. Foursquare is an excellent example of a modern technology company where the technical focus is on both mobile and cloud platforms. It has become common to have customers interact via mobile devices while the back-end is hosted on a pay-as-you-go cloud. Using a cloud for back-end services is a viable solution for young technology businesses since it allows them to avoid significant upfront costs and enable rapid growth. This is a reflection of the current technology landscape where businesses are using mobile platforms for their front-end applications, while using computing clouds for their back-end services.

Technology companies similar to Foursquare face two important challenges. The first challenge is managing the development of functionally equivalent applications for heterogeneous mobile platforms. The second one is maintaining a portable back-end to mitigate any catastrophic outage, for instance, being able to fail-over onto multiple clouds, avoiding a single point of failure for the back-end services.

The importance of maintaining a portable back-end was highlighted during the recent outage of Amazon EC2 [2]. This lead to a catastrophic unavailability of the services offered by Foursquare, damaging its reputation. The fact that Foursquare was not able to restore services during the EC2 outage indicates the extent of their dependence on a single cloud. Foursquare is just one of many tech-businesses that was effected by the EC2 outage, further underscoring the importance of having portability in cloud applications.

Developing portable cloud applications using current state-of-the-art is a cumbersome process. The primary reason for this is the heterogeneity across different cloud platforms, e.g., different APIs, different data models, different query languages and different supported frameworks. These heterogeneities are sometimes deliberately created to retain users (vendor lock-in) or can arise due to the lack of standards. This ultimately results in many cloud applications being rebuilt from scratch, when the need to port arises.

Given the strong incentive to use clouds in the face of portability issues, we believe what is needed is a unified modeling, development, deployment and management of cloud applications. This paper presents a platform agnostic development component of the solution to this problem. We focus on

[1] https://foursquare.com/

[2] http://blog.rightscale.com/2011/04/25/amazon-ec2-outage-summary-and-lessons-learned/

the constrained domain of cloud-mobile hybrid applications. Cloud-mobile hybrids are applications that have mobile device based front-ends but use cloud based back-ends for data storage and processing, such as the Foursquare application.

This paper makes the following contributions:

1. We discuss our modeling strategy for cloud-mobile hybrids, based on the Model-View-Controller (MVC) design pattern.
2. We present a DSL based on the MVC modeling, highlighting the applicability in the enterprise space.
3. We detail the Web based tools that allow convenient composition using our DSL.
4. We present a partial objective evaluation, comparing code statistics of generated applications.

We call our system *MobiCloud*, signifying its applicability across the cloud and mobile spaces. The DSL is referred to as the *MobiCloud DSL*.

2. THE MOBICLOUD DSL

We now present our language design strategy and the details of the DSL.

2.1 MVC based DSL Design

The MVC design pattern was first discussed by Trygve Reenskaug [6] and later detailed by Steve Burbeck [1], as a user interface design paradigm for the Smalltalk programming language. MVC has since been used as the primary design strategy for a number of application frameworks, Apache Struts [3] and Ruby on Rails [4] being two recent examples.

In exploring the typical applications being developed as mobile device based front-ends for cloud based applications, we made the following observations:

- The front-end and the back-end applications are usually managed as separate projects. These projects depend on well-defined service interfaces to implement either the client or the service functions.
- Front-end applications are tied strongly to the back-end application. Although some level of loose coupling is possible, updates to the back-end applications would eventually have to be propagated to the front-end applications. Such change propagation requires sweeping changes to the front-end application code, often to multiple front-end applications targeted towards different mobile devices.
- A significant effort is needed to debug the applications due to their use of remote procedure calls (RPC).

However, regardless of the separation between the back-end and the front-end through a service layer, the functional components still maintain their relevance to the MVC design. For example, the model data structure is present in both the back-end and the front-end, equivalent in behavior yet different in implementation. Similarly, the back-end service implementations typically act as controllers that filter the operations on the data storage.

[3]http://struts.apache.org/
[4]http://rubyonrails.org/

Thus, we modeled the entire functionality, i.e., the functionality of the front-end mobile application as well as the back-end cloud application, as a single unit, based on MVC. The DSL-based representation of this model can be automatically transformed to the required software components as needed.

Our approach has the following benefits:

- The same model can be used to *generate* functionally equivalent, mobile front-end applications, as well as cloud back-end applications targeting different platforms.
- Many developers are familiar with the MVC design pattern, thus this modeling has a gentler learning curve.
- Modeling complexity is reduced by treating the entire functionality as a single unit. The complexity, however, is now transferred down to the DSL transformation mechanism.
- The high level modeling includes data modeling that can be used to generate data transformations. Although such transformation are out of the scope of this paper, they are important in solving the problem of application migration across clouds.

Following the MobiCloud approach, application code portability is achieved by simply regenerating a functionally equivalent application for the target platform. Achieving data portability is also supplemented by the ability to generate data transformations. Performing an application migration (both code and data); however, would require the services of a middleware layer, such as the IBM Altocumulus research project [4].

2.2 Details of the DSL

MobiCloud DSL closely resembles the MVC design by providing constructs for each of the three key components: *model*, *view*, and *controller*. Each of these constructs act as place holders to collect details of the respective component. For example, the *model* construct captures a unique identifier for a model and the attributes of the model as key-value pairs.

We developed the MobiCloud DSL in stages. The first generation (MobiCloud I) consists of only the basic constructs and predefined action behavior. The second generation (MobiCloud II) consists of extensions that enable a variety of additional capabilities such as the use of predefined models, views, or controllers.

Listing 1 depicts a first generation MobiCloud script for a simple task manager application, intended to illustrate relationships between the components as well as the constructs of the DSL. It includes :

1. A *model* with four attributes, used to store task details (lines 4 to 7). This construct expresses the data structure that is needed to store the required application data.
2. A *controller* with two actions for creating and retrieving tasks (lines 9 to 12). Given that the data structure of interest is *task* (indicated by using the model name in actions in lines 10 and 11), the controller auto-generates the operations for creating and retrieving a task data structure.

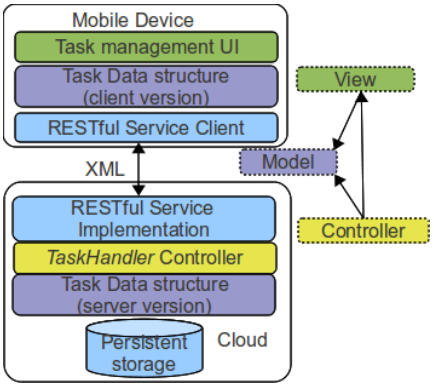


Figure 1: Generated components for the Task Manager application. The model contributes to two equivalent data structures while the controller primarily contributes to a back-end service wrapper. The view mainly contributes to the mobile front-end portion, creating the UI

3. Two *views* with basic user interfaces to add and retrieve tasks (lines 14 to 19). The views also assume, based on the referred action and the model, the appropriate UI rendering. For example, the create view includes text boxes (or equivalent UI component in the target platform) for string attributes.

Listing 1: The DSL script for the *task manager* application in MobiCloud. Extra line breaks have been inserted for formatting.

```
1 recipe(:todolist) do
2   metadata({:id => 'task-manager'})
3    # models
4   model(:task, {:name=>:string,
5   :description => :string,
6   :time => :date,
7   :location => :string})
8  #controllers
9  controller(:taskhandler) do
10    action :create, :task
11    action :retrieve, :task
12  end
13  # views
14  view :add_task, {:models =>[:task],
15  :controller => :taskhandler,
16  :action => :create}
17  view :show_tasks, {:models =>[:task],
18  :controller => :taskhandler,
19  :action => :retrieve}
20 end
```

The current MobiCloud tools (Section 5) are capable of producing Android and Blackberry applications as front-ends and Google App Engine (GAE) and Amazon EC2 applications as back-ends. The generated applications are readily deployable, either using the MobiCloud tools or using the respective cloud or mobile development kits (SDK). For brevity, we conclude our description of MobiCloud I here and direct the readers to Manjunatha et. al. [3] for details. In the next section, we discuss the new extension capability of the second generation MobiCloud DSL.

3. EXTENSIONS

The second generation MobiCloud (MobiCloud II) added an extension capability to allow predefined models, views, or controllers to be available to the language, simply by requiring the extension in a script.

Listing 2 shows a partial code fragment from the second generation MobiCloud, exemplifying the use of an extension. In general, an extension adds specific capabilities to the generator. For the illustrated URL extension, the generators receive the ability to generate controller code to fetch contents of a URL, and optionally, assign values from the output to a specified model. Fetching the contents of a URL is a capability present in all cloud platforms but each platform carries its own quirk; for example, GAE requires the use of the Google URL fetch library to access external URLs while EC2 has no such restriction.

Listing 2: Using the URL fetch extension in MobiCloud. Extra line breaks have been inserted for formatting.

```
1 # Generic HTTP fetcher
2 # exemplified using Yahoos network time fetcher
3 recipe :http_fetch do
4     # Enabling generic http extension
5   extensions ['http']
6     # metadata
7   metadata({:id => "httpfetcher"})
8    # models
9   model :time_value,{:ts => :int}
10  #controllers
11  controller :time_manager do
12      action :fetch_time,:time_value,
13      {:type=>'http',
14       :url => 'http://developer.yahooapis.com/xx',
15       :params => {:appid => 'xxx'},
16       :return_mapping =>
17           {:ts=> '/Result/Timestamp'}}
18       end
19  # views
20  view :view_time, {:model =>:time_value,
21              :controller => :time_manager,
22              :action => :fetch_time}
23 end
```

3.1 Integrating Extensions

Extensions are integrated to the generators via predefined hooks. The most common method of integration is to globally augment the semantic object model of the parsed DSL by inserting predefined models or views during the post model creation hook. For generation tasks that require specific code to be inserted, such as for controllers, targeted extension hooks are used. These are platform specific, i.e.,they are required to follow the specific code guidelines for the respective platform. For example, the URL Fetch extension illustrated in the example modifies the controller templates in both GAE and EC2 to insert customized versions of a URL data extraction function. Extensions can also insert extra libraries to the respective projects. This extension mechanism is illustrated in Figure 2.

4. ENTERPRISE INTEGRATION

An important addition, possible via the extensions mechanism, is enterprise integration. One such featured extension is the *Salesforce* extension that allows one to integrate Salesforce.com data, such as a contact list. Salesforce [5] is a popular enterprise application platform to create business applications and provides many support services such as authentication, scaling etc. Salesforce integration requires a significant learning and debugging effort, especially the OAuth [6] based authentication mandated by Salesforce for

[5]http://www.salesforce.com/
[6]http://oauth.net/

Listing 3: A Salesforce contact extraction application written using the MobiCloud DSL. The salesforce extension adds extra actions as well as predefined models. Extra line breaks have been inserted for formatting

```
# Salesforce contact list manager
recipe :sforce_contacts do
  # salesforce extension
  extensions ['salesforce']
  # metadata
  # Adding the salesforce extension
  metadata({:id => "salesforce_contacts",
            # mandatory values from the remote application
            :salesforce_clientid => 'xxxxxxx',
            :salesforce_clientsecret => '2788412111461228187',
            :salesforce_server_root => 'na3'      })
  # models
  model :salesforce_contact # salesforce contact object, attributes predefined by the extension
  #controllers
  controller :contact_manager do
    # fetch & display contacts from salesforce
    action :fetch,:salesforce_contact,{:type => 'salesforce'}
   end
  # views
  view :view_contacts, {:model =>:salesforce_contact, :controller => :contact_manager,
                                                                    :action => :fetch}
end
```

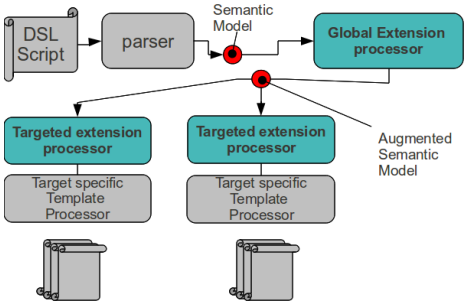


Figure 2: Extension processing happens by augmenting the parsed model globally or locally, and/or modifying specific code segments in the templates

selected content. Using the MobiCloud Salesforce extension saves significant time and effort in developing integrated applications by shielding developers from the intricacies of the OAuth mechanism. The extension adds the necessary libraries and user interfaces for authentication with Salesforce, augmenting the base controllers and views, as well as predefined data structures needed to extract Salesforce data.

Listing 3 illustrates the DSL code for an application that displays a Salesforce contact list on a mobile device. The following extra components are generated when this DSL is used with the code generators.

- All required data structures to represent a Salesforce contact. In this case, data structures for an organization and a contact are generated. The attributes for these data structures are defined by following the Salesforce service descriptions.
- The views for the corresponding Salesforce models.
- A view that acts as the front-end for the OAuth based authentication. When users try to perform actions that need authentication, they are automatically redirected to this authentication view.
- A controller component (a servlet in the case of a Java Web application) that handles the authentication. This controller caches the credential data following the OAuth protocol. It also acts as the callback endpoint for the OAuth handler.
- The necessary data storage provisions for the models as well as the credential caches.

Some extensions under development will add support for custom data type inclusions, UI customization and integration of popular services such as Google Maps [7], etc.

5. ONLINE TOOLS

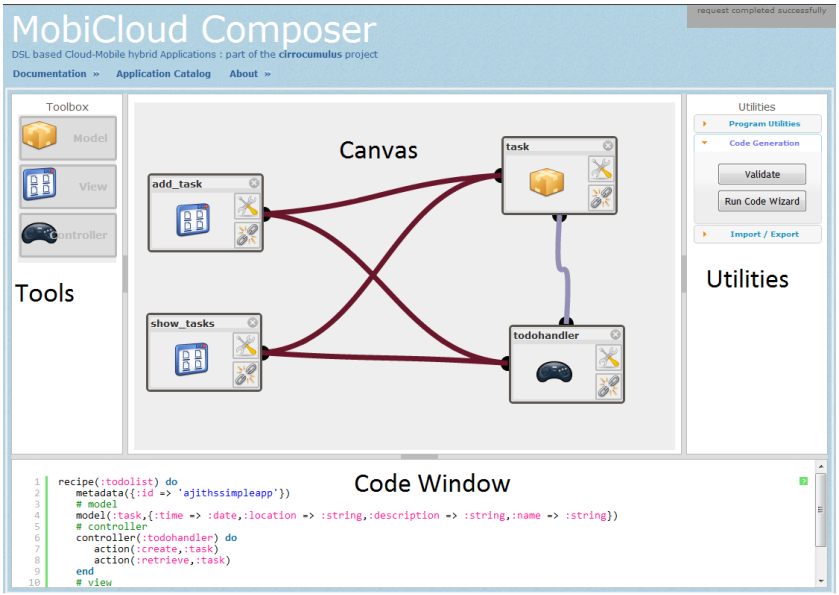
Although the DSL itself is simple and concise, an MVC design becomes much more palatable when done graphically. Such graphical design also alleviates a significant portion of the language learning curve. Thus, we created two Web-based applications to compose and store MobiCloud applications.

5.1 MobiCloud Composer

The composer is a graphical tool that can be used to generate MobiCloud code using graphical components. Figure 3(a) illustrates the Web based user interface of the composer. Graphical icons representing model, view, and controller constructs can be dragged on to a canvas and connected to create the required configuration. The code is created on the fly and displayed below in the code window.

The advantage of this graphical UI is two fold. First, it alleviates the need to learn the syntactic details of the language. A composition can be made entirely in the graphical form, without writing any code. Use of graphical expressions makes it easier to visualize the application and hence results in faster composition. The second is that it facilitates other convenient additions such as direct deployment and packaging. The composer provides packaging for selected mobile platforms and also the direct deployment capability to clouds. For example, when the Android application is

[7] http://code.google.com/apis/maps/index.html



(a) MobiCloud composer user interface. The canvas at the center contains draggable widgets that are *plumbed* together to graphically depict their relationships. This image illustrates the graphical composition of the task manager application listed in Listing 1.

(b) The generated application, running on an Android device. The image illustrates the auto generated add_task view.

Figure 3: The Composer UI and the generated application running on an Android device

compiled via the composer, it creates an Android package (APK file) that can be installed in a compatible device.

The composer tools are available online for public use [8]. The current graphical composer is only MobiCloud I capable. A text based composer is available for MobiCloud II compositions.

5.2 MobiCloud Catalog

MobiCloud catalog is a storage and catalog solution for MobiCloud scripts. Users can store MobiCloud scripts, either manually or by exporting directly from the composer. The composer also has integrated features to search and import code from a catalog. The users can simply search for publicly available scripts and import them to the composer, without leaving the composer. A MobiCloud catalog instance is available for public use [9].

6. EVALUATION

We present the important code metrics of generated applications to highlight the manual effort that is needed to build these applications. Table 1 highlights the lines of code in the DSL vs. the generated applications. These statistics were obtained using the metrics plugin [10] and the CLOC tool [11].

The code statistics clearly show that there is a significant reduction in effort by using the DSL. Even for a simple application with one model and one controller, the Android front-end alone requires around 500 lines of Java and XML code. Similarly, using an extension, such as the salesforce extension, adds significant increase in the Java code and complexity to the back-end.

[8]http://mobicloud.knoesis.org/
[9]http://mobicloud-catalog.knoesis.org/
[10]http://metrics.sourceforge.net/
[11]http://cloc.sourceforge.net/

7. DISCUSSION

The first generation MobiCloud provided evidence of the benefits of a DSL in the development process. MobiCloud advocates a model driven development process that us likely to be preferred over traditional model driven approaches for the following characteristics:

1. The DSL is simple and no heavy upfront design is required. This approach meshes well with Agile development techniques which value quick iterations over heavy up front designs.
2. The tool performance is sufficient to do rapid prototyping. The typical code generation task takes only a few seconds.
3. The generated code may be used as boilerplate, i.e., developers can use MobiCloud to simply get rid of the repetitive programming and focus on the more creative aspects, such as customizing the UI.

HTML 5 has come up as a strong alternative to cross platform applications. However, HTML 5 still lacks strong platform integration features and is not able to replace the user experience of a native mobile application. Thus it is safe to assume that a tool like MobiCloud, with native front-end generation capabilities, will be relevant, in spite of advanced Web standards such as HTML 5. MobiCloud can be simply extended to generate an HTML 5 front-end, if such a capability is desired.

7.1 The Case of the Smallest Common Subset

During initial public exposures of MobiCloud, many developers have raised the issue of just supporting the smallest

Script	Description	DSL Lines of Code	Models	Views	Controllers	Target	Generated Lines of Code (Java, JSP and XML)	Ratio of DSL to Generated
Shop Manager	An application to keep track of jobs and customers for a mechanics shop	17	2	4	2	Android Blackberry EC2 GAE	1244 592 628 1021	1:73 1:35 1:37 1:60
URL Fetcher	Fetches and displays timestamp values from the Yahoo time Web service	9	1	1	1	Android Blackberry EC2 GAE	486 100 289 466	1:54 1:11 1:32 1:52
Salesforce Contacts	Fetches and displays the contact list from a Salesforce account	9	1	1	1	Android Blackberry EC2 GAE	794 - - 1377	1:88 - - 1:153

Table 1: Selected code metrics of generated applications

common subset of features and thereby limiting the usefulness of the DSL. This aspect however, does not limit the capabilities of the DSL severely in our case (mobile and cloud) because features not directly present can always be simulated or approximated in an indirect way during the customized compilation process. That is, all mobile platforms support features that are deemed essential and the code generators can include graceful degradation of application features when the target platform does not have the required hardware or software capabilities. Such a strategy also applies to the cloud platforms that support features with similar semantics, yet are different syntactically and structurally.

8. RELATED WORK

Using DSLs to generate target specific applications is not new. There have been many efforts that have focused on different domains, ranging from hardware drivers [8] to Web applications [5]. Greenfield et al. [2] has provided a methodology to incorporate DSLs into the software engineering process, implemented in specific sections of the Microsoft Visual studio development environment. Simonyi et al. [7] has discussed an abstraction based generative programming approach for software engineering. Although the application of a DSL is not explicitly discussed, the principle followed by Simonyi et al., agrees with the MobiCloud approach.

In the area of mobile application development, there are many commercial vendors that provide unified development kits for multiple mobile platforms. Rhomobile [12] is one such vendor that provides a wide array of capabilities using a Ruby based platform. Other commercial service providers such as PhoneGap [13] and Cabana [14] have used HTML5, CSS and javascript as the foundation of their development platform. Almost all these frameworks depend on the mastery of an existing programming framework and do not provide a high-level view of the application similar to MobiCloud. Google Web Toolkit (GWT) [15] has a conceptual similarity to MobiCloud, since it treats a Web application as a single unit during development. However GWT is not intended for server side portability and only supports HTML driven front-end portability across multiple browsers.

[12]http://rhomobile.com/
[13]http://www.phonegap.com/
[14]https://www.cabanaapp.com/landing/
[15]http://code.google.com/webtoolkit/

9. CONCLUSION

Modeling and developing portable cloud-mobile hybrid applications, is becoming an important requirement for businesses. Using a well-crafted DSL, we can provide an acceptable solution, by generating platform-specific applications with equivalent functionality. The MobiCloud solution has amply demonstrated that such a system is viable, and with the additional capabilities provided by the extensions, it shows promise to provide a comprehensive solution.

10. REFERENCES

[1] S. Burbeck. *Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC).* Softsmarts, Inc., 1987.

[2] J. Greenfield and K. Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 16–27. ACM, 2003.

[3] A. Manjunatha, A. Ranabahu, A. Sheth, and K. Thirunarayan. Power of Clouds In Your Pocket: An Efficient Approach for Cloud Mobile Hybrid Application Development. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 496–503. IEEE, 2010.

[4] E. Maximilien, A. Ranabahu, R. Engehausen, and L. Anderson. IBM Altocumulus: A Cross-cloud Middleware and Platform. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 805–806. ACM, 2009.

[5] M. Nussbaumer, P. Freudenstein, and M. Gaedke. Towards DSL-based Web Engineering. In *Proceedings of the 15th international conference on World Wide Web*, pages 893–894. ACM, 2006.

[6] T. Reenskaug. The Original MVC Reports. *Xerox Palo Alto Research Laboratory, PARC*, 1978.

[7] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. *ACM SIGPLAN Notices*, 41(10):451–464, 2006.

[8] S. Thibault, R. Marlet, and C. Consel. A Domain Specific Language for Video Device Drivers: from Design to Implementation. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 2–2. USENIX Association, 1997.