

# Semantics Centric Solutions for Application and Data Portability in Cloud Computing

Ajith Ranabahu and Amit Sheth

Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis)

Wright State University,

Dayton, Ohio 45435

Email: { ajith, amit }@knoesis.org

## Abstract

*Cloud computing has become one of the key considerations both in academia and industry. Cheap, seemingly unlimited computing resources that can be allocated almost instantaneously and pay-as-you-go pricing schemes are some of the reasons for the success of Cloud computing. The Cloud computing landscape, however, is plagued by many issues hindering adoption. One such issue is vendor lock-in, forcing the Cloud users to adhere to one service provider in terms of data and application logic.*

*Semantic Web has been an important research area that has seen significant attention from both academic and industrial researchers. One key property of Semantic Web is the notion of interoperability and portability through high level models. Significant work has been done in the areas of data modeling, matching, and transformations. The issues the Cloud computing community is facing now with respect to portability of data and application logic are exactly the same issue the Semantic Web community has been trying to address for some time.*

*In this paper we present an outline of the use of well established semantic technologies to overcome the vendor lock-in issues in Cloud computing. We present a semantics-centric programming paradigm to create portable Cloud applications and discuss MobiCloud, our early attempt to implement the proposed approach.*

## 1 Introduction

Cloud computing has become one of the most important evolutions computer science has seen recently. The success for Clouds can be attributed to the ability to provide seemingly unlimited computing resources almost instantaneously and also to the pay-per-use pricing schemes. The consumer market, primarily small and medium businesses has embraced the Cloud as the primary source of their com-

puting resource needs. Major enterprises and even the government has invested in *private Clouds* for efficiency and energy considerations.

The Cloud computing landscape however is plagued by many issues hindering adoption. One major issue is vendor lock-in, where the data and applications are very hard to be moved to other systems (thus *locked-in*) forcing the Cloud users to adhere to one service provider. This lock-in is present at two different levels with respect to the National Institute of Science and Technology (NIST) categorization [12].

- Within the same type of Clouds, termed *vertical heterogeneity*. For example, Infrastructure as a Service (IaaS) is one category of Clouds where raw infrastructure resources are exposed as services. Heterogeneity between IaaS providers qualifies as vertical heterogeneity.
- Within different types of Clouds, termed *horizontal heterogeneity*. Horizontal heterogeneity is always present and fundamentally hard to overcome due to the shift in the paradigm. For example, Platform as a Service (PaaS) Clouds follow the philosophy of making the device and platform details transparent to the Cloud user. Such Clouds typically require the use of specialized libraries and constrained runtimes. An application that assumes the availability of control over operating system would require extensive changes when moving into a platform cloud.

Both types of heterogeneities force Cloud users to stick to a single vendor or a particular type of Cloud or impose heavy expense on porting.

In this paper we suggest a top-down methodology for the application development process based on a semantic partitioning of an application. The objective of this methodology is to come up with a specification of an application at a significantly higher level of abstraction, facilitating interoperability and portability. These specifications can later be

run through generic transformers to generate targeted (platform specific) code and data specifications or used directly with capable *virtual machine*. Our development process relies heavily on Domain Specific Languages (DSL) to enable convenient yet powerful descriptions.

In Section 4 of this paper we discuss a semantic based *separation of concerns* for an application. Then we introduce a DSL and demonstrate an application specification formulated using these DSLs in Section 5. We present the results of our implementation and the lessons learned in Section 6.

## 2 Motivation

While there are many cases that highlight the importance of portability, we chose two particularly representative cases from the Cloud computing usecases white paper [1] and one case from academia. The Cloud computing usecases white paper is a result of on-going discussions among professional software engineers and is based on real-world scenarios. The scientific research use-case comes from the authors collaborative work with biologists.

### 2.1 Payroll Processing in the Cloud

This scenario includes the experience in moving a payroll processing application to an Infrastructure (IaaS) Cloud. This application requires an Enterprise Java Bean (EJB) enabled application server as well as a relational (i.e. SQL supported) data store. An infrastructure Cloud is clearly suitable for this type of installation since there is flexibility to control features from the hardware specifications and up. However, the tight requirement of being bound to an enterprise Java environment and also a relational data store makes this application almost impossible to be ported to a platform based Cloud such as Google Appengine or Microsoft Azure. This is disadvantageous in a business perspective, for example, platform Clouds offer very competitive pricing for resources but it would not be possible to port the code without significant upfront investments. In a technical perspective, prominent Infrastructure Cloud providers have had catastrophic failures and hence it is essential to place fail-safes in different Clouds to improve availability. This is also not possible in this case without significant porting effort. These considerations are highlighted in the white paper as serious concerns in porting the application to different Clouds.

### 2.2 Cloud based Logistics Management Application

In this scenario, an application was built from scratch to automate a manual logistics control process for a medium sized business. The center piece of this application is the Cloud based global data store, exposed via services. Current data store is managed in the Google Bigtable [3],

a highly scalable, schema less, document-oriented (i.e. non relational) data store coupled with Google Appengine Cloud. The primary portability concern in this scenario is the dependence on the schema-less data store of which non of the other Clouds directly support. The service implementations depend on queries written in Google Query Language (GQL), a declarative language comparable to SQL, built with a different paradigm. This dependency dictates that when the data store is ported, the service implementations would also require transformations to the relevant query language.

### 2.3 Cloud based Statistical Processing Services for Life sciences

Some of the experiments that take place in the biology domain, particularly the experiments that involve equipment such as the Nuclear Magnetic Resonance (NMR) Spectrometer, generate extremely large numerical datasets. These datasets need to be passed through a number of statistical processes to get useful information. Many of the statistical algorithms can be parallelized. Some of the most frequent statistical processes were implemented based on Apache Hadoop by the Kno.e.sis Cloud team, enabling some of the computations to be run on a small internal cluster as well as on public infrastructure Clouds [9]. However, an opportunity to use a commercially available platform Cloud came up later which could not be utilized primarily due the significant effort needed to port the existing implementation to suit the platform Cloud.

These real world scenarios highlight two different aspects of the Cloud landscape today.

- Legacy applications may require significant effort in porting to Cloud environment due to the tight coupling with particular technologies and/or data organization.
- Even when applications are written from scratch, they are targeted and thus locked for a particular Cloud. A porting effort for a different Cloud becomes a one time exercise.

The above scenarios motivated us to take a closer look at the current application development practices and investigate methodologies for developing applications in a Cloud agnostic manner. Both industry and academia suffer from the current practice of developing exclusively for a target Cloud, economically, technically, or both. Some experts even argue that vendors will never standardize their services for business reasons [5], highlighting the importance of catering for heterogeneity.

In the subsequent sections of the paper, we will use a todo list manager as our running example. This application is non-trivial yet simple enough to illustrate the new development process. Web based todo list managers (Such

as the popular *Remember the milk* online task manager<sup>1</sup> may need to scale horizontally due to the large volume. The front-end, however, need to be readily accessible, ideally through a mobile device.

### 3 Background

There are two areas of pertinent background work.

- Software Engineering research that has explored the theoretical foundations of applications and language abstractions.
- Semantic Web services research that introduced the high level separation of concerns for Web services. This research is helpful in understanding the use of semantics.

Software applications have been studied for many years and there has been many paradigm shifts already. Notable milestones in these paradigm shifts include the transition from sequential to object-oriented programming (OOP) as well as the rise in aspect oriented programming (AOP) [6]. AOP is the state of the art in addressing the isolation of cross cutting concerns such as security and provenance. AOP however is not being adopted widely due to the difficulties of readability and many other issues that conflict with the established programming practices. Steimann [18] discusses some of these issues in detail. It is clear, however, that AOP is not considered to be the most appropriate method to separate the concerns governing an application, although such a separation is immensely useful.

Another direction that has been taken by software engineering research community is to apply domain specific languages (DSL) to application development. The premise is that general purpose programming languages do not always provide the convenience a programmer would need when addressing a particular domain. For example, in developing a mathematics oriented program, it is convenient to have abstract notions for functions, matrices and other mathematical operators such as exponents. In fact, introducing such an abstraction enables domain experts (non-programmers) to directly create programs with little or no knowledge in programming. This approach has been used in many domains at different levels of granularity as exemplified by Matlab [8] SQL and many others.

The use of DSLs as a main stream programming tool has been recently advocated by the Software Factories approach by Greenfield et al. [7]. Their approach uses an array of DSLs to create applications, letting the programmer choose a suitable abstraction for the concern at hand. Thus an application becomes a collection of specifications written in multiple DSLs. In fact many of the current Microsoft (and some opensource) development tool suites follow a similar theme. Greenfield argues that the inability of the OOP

<sup>1</sup><http://www.rememberthemilk.com/>

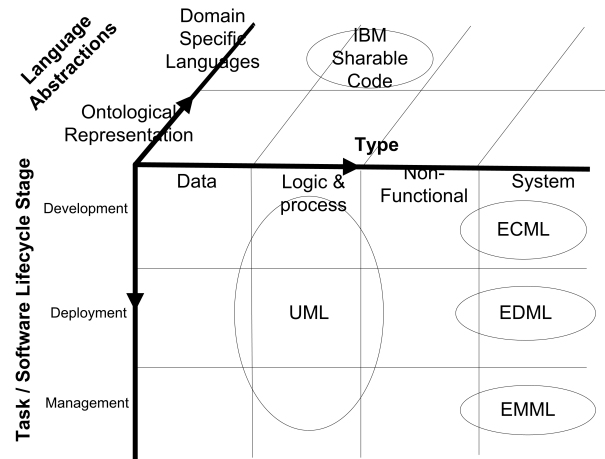


Figure 1: Partitioning of the Modeling space for Clouds

based methods adhering to time and budgetary constraints is primarily due to the lack of sufficient abstractions. DSLs excel at providing these abstractions.

Apart from these major undertakings, there has been a large number of small scale developments based on DSLs both in academia and the industry, ranging from service mashups to parallel programming.

A semantics based *separation of concerns* for a Web service was first presented by Sheth et al. [15, 17], partitioning the semantics of a service to four types. They are data semantics, functional semantics, non-functional semantics, and execution semantics. The difference in using semantics is that the separations are at a high level of abstraction and enables one to focus energy on specific concerns. For example, Quality of Service (QoS) is a non-functional concern that can be addressed and modeled separately from the data considerations. *Intertwining* these considerations is achieved by annotations that link different models and descriptors addressing different concerns. This is the philosophy followed by SAWSDL<sup>2</sup> and SA-REST<sup>3</sup>.

We present a partitioning in the modeling space that gives a better perspective of the relationship of the high level models. This partitioning is based on our earlier categorization presented in [16]. Figure 1 illustrates this partitioning. One of the dimensions is the four types of semantics identified in Section 4. The other two dimensions include the software development life-cycle and the level of language abstraction. The key realization from this partitioning is the fact that there is no single model that fits the needs of all the modeling requirements.

<sup>2</sup><http://www.w3.org/TR/sawSDL/>

<sup>3</sup><http://www.w3.org/Submission/SA-REST/>

## 4 Semantics of an Application

The development strategy we propose is model-driven, i.e. top-down. While bottom-up approaches are known to work well for tightly integrated platforms, most of the loosely coupled development processes are top-down. The best approach is debatable and often depends on the domain. In this case model driven development has clear benefits in terms of supporting portability.

We introduce four types of semantics for an application, inspired by the four types of semantics for a *service*, mentioned in Section 3. This partitioning is necessary to model the different concerns at a sufficiently higher level. The term *application* is used to mean a software program designed to help the user to perform singular or multiple related specific tasks. The partitioning we provide is different than the original four types of semantics identified for a service.

The identified types of semantics are as follows:

- Data semantics
- Logic and process semantics
- Non-functional semantics
- System semantics

### 4.1 Data Semantics

Data semantics address the data aspects of an application. This includes that definitions of data structures, relationships across multiple data structures as well as restrictions on the access of some of the data items.

### 4.2 Logic and process semantics

These are the semantics pertaining to the core functionality (commonly referred to as the *business logic*) of the application. Unlike in a service, the functional and execution semantics are tightly tied together for an application. For example, behavior of a service during an exception may be handled externally although exceptions are an integral part of the core functionality of an application and seldom designed separately.

### 4.3 Non-functional semantics

These are semantics not-directly relevant to the business logic but requires consideration, perhaps at a different level. Examples include access control and logging. While these are not part of the core functions, an application nevertheless requires them to be defined. Some of these considerations may require certain libraries or internal code changes. A typical example is logging where the application internally implements the points of logging but the users get to control the granularity of the entries such as INFO (informational content) vs ERROR (only errors).

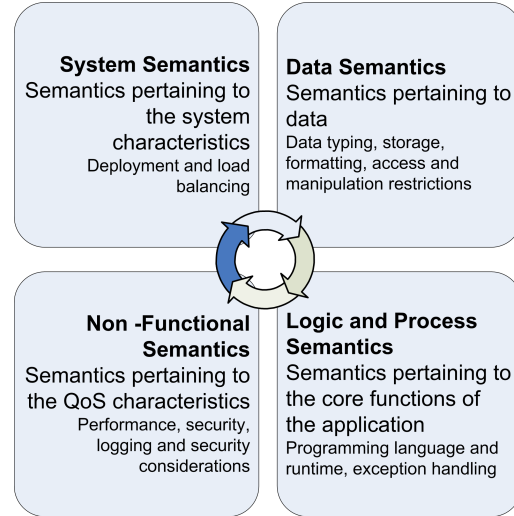


Figure 2: Four types of semantics for an Application

## 4.4 System semantics

System semantics govern the system related concerns of the application. Relevant considerations include deployment descriptions and dependency management. These considerations are neither relevant for the business logic nor the non-functional considerations but become important when the application starts running on a system. Semantic technologies are already being used in this space commercially. Elastra Inc. uses three ontologies (semantic models) to describe the configuration, deployment and management details for an application deployed to an Infrastructure Cloud.

Table 1 shows the different aspects covered by these semantics and also some example DSLs.

## 5 Applying DSL for Application Development

We now present one of our experimental DSLs targeted towards generating Cloud-mobile hybrid applications. A Cloud-mobile hybrid is an application that implements the back-end (data storage, partial business logic) in a Cloud environment but has a mobile device based front-end (user interface). Such an application has double the portability issues since both the front-end and the back-end should support portability in different scopes.

This presented DSL, named *MobiCloud* is modeled after the Model-View-Controller (MVC) pattern. Further details of the language is available from the *MobiCloud* technical report [10] as well as the online toolkit <sup>4</sup>.

Figure 3 illustrates the *todo list* application. This particular code generates a back-end data store for tasks, data access code and a RESTful service capable of creating and retriev-

<sup>4</sup><http://knoesis.org/mobicloud>

Type	Application features	Example DSLs
Data	data definitions	SQL(DDL)
Logic & process	core functional description, exceptional behavior, user interfaces	ISC, MobiCloud, Sinatra <sup>a</sup>
Non-functional System	Security profiles, provenance data build configuration, deployment configuration	SAML <sup>b</sup> GNU Make <sup>c</sup> , Apache Ant <sup>d</sup>

<sup>a</sup><http://www.sinatrarb.com/>

<sup>b</sup><http://saml.xml.org/saml-specifications>

<sup>c</sup><http://www.gnu.org/software/make/>

<sup>d</sup><http://ant.apache.org/>

**Table 1:** Considerations for semantics

```

recipe(:todolist) do
  metadata({:id => 'todo-app'})
  model(:todoitem, {:name=>:string,
                  :description => :string, :time => :date,
                  :location => :string})
  model(:user, {:name=>:string, :id => :int})
  controller(:todohandler) do
    action :create, :todoitem
    action :retrieve, :todoitem
    ...
  end
  view :todo_add, {:models =>[:todoitem], :controller => :todohandler, :action => :create}
  view :todo_show, {:models =>[:todoitem], :controller => :todohandler, :action => :retrieve}
end

```

Annotations in the image:

- Non-functional details and metadata**: points to the `metadata` line.
- Data definitions**: points to the `model` lines.
- Logic and Process definitions covering the back-end actions and the user interface**: points to the `controller` and `view` lines.
- Models**: points to the `model` lines.
- Controllers**: points to the `controller` line.
- Views**: points to the `view` lines.

**Figure 3:** A task manager application written in MobiCloud DSL

ing the tasks message. The front-end consists of the UI and service access code capable of creating and retrieving the tasks. While the details of this DSL are omitted for brevity, we use this DSL as a substrate to demonstrate the addition of semantics to this process. Note that the code in Figure 3 has no semantic modeling attached to it.

## 5.1 Adding Semantics

### Data semantics

The first point of addition of semantics to this DSL is data. Although data definitions are present in the models section and clearly separated from the process, these data definitions are not reusable. Well established semantic data definitions can be referenced in the DSL rather than the one-off in-line definitions. The term semantic data definition has been used to refer to concepts defined in an ontology where detail-rich relationships can be attached.

Listing 1 illustrates how the model data can be externalized without compromising the simplicity of the DSL. The

snippet highlights the addition of a data item that adheres to the description of the Friend-of-a-Friend (FOAF) person definition.

**Listing 1:** Referring to an established concept for data

```

...
model :user, { :ref => "foaf:Person" }
...

```

### Logic and process semantics

The least reused component of an application is the business logic. While all other aspects may be composed from ready made components, this is not the norm for business logic. There are, however, well studied solutions that may be applied to recurring problems. These solutions are known as *design patterns* and applied throughout the software industry. Semantics can be used to specify these patterns at a higher level and then linked to the DSL. The generators can then identify the pattern and generate the necessary code or augment the logic accordingly.

Listing 2 shows using an annotation to specify that the application is shared multi-tenant, i.e each user date is separated but the system uses a shared schema [4].

**Listing 2:** Specifying functional characteristic with high level design patterns

```

...
metadata { ... ,
            :pattern => "shared_multi_tenant" }
...

```

### Non-functional semantics

Figure 3 does not specify any non-functional characteristics or QoS. QoS characteristics have been heavily studied in the Semantic Web service research where the value of semantically defined QoS features have been highlighted. For example, there are multiple research attempts to enable QoS matching for services based on semantic models

[14]. These semantic models have been extensively discussed although most of them are services centric. Nevertheless, the experience in designing service based QoS ontologies is very useful in building an application oriented non-functional ontology. The exact nature or the granularity of such an ontology is out of the scope of this paper. Instead we illustrate the capability to add non-functional characteristics defined in ontologies.

There are two methods to incorporate non-functional characteristics to a DSL based program.

- Use annotations in the script to point to externally defined constraints.
- Use a *wrapper* language to associate a configuration script, possibly written in a different DSL.

Listing 3 demonstrates the use of annotations to attach security constraints in our running example. We use an already available security profile ontology to annotate the DSL and highlight that the specified controller actions need to be secured.

**Listing 3:** Using a reference to a security profile

```
...
controller : todohandler do
  action : create , : todoitem ,
          { : security_profile => 'ssl' }
...
end
...
```

## System semantics

System semantics are not as tightly coupled to an application as data or process semantics. Some system aspects are handled separately from development process. In this case we suggest adding system details or constraints as meta data to the DSL. The meta data section in the DSL is explicitly provided for this purpose. For example, the DSL can refer to a ECML and EDML script to define the configuration and the deployment description.

Listing 4 shows a reference to a known software stack description.

**Listing 4:** Using an external deployment description

```
...
metadata { ... ,
  : deployment_profile
  => "... ApacheStack-full.ttl" }
...
```

We now present the DSL complete with all the semantic annotations. Figure 4 shows all the semantic annotations with relevant details highlighted for each type.

## 6 Results and Experience

We are confident about the soundness and practicality of this approach based on the results we obtained using the MobiCloud DSL. The MobiCloud DSL supports only a limited set of semantic additions as of now.

### Importance of Semantics

A DSL can be thought of as a lower grade semantic model. A DSL is a grounding of a portion of a high level semantic model. To specify higher level concepts in a platform agnostic fashion, semantic models are essential.

In the case of MobiCloud we experienced that while the DSL is capable of expressing some of the core specifications, it was difficult to clearly and elegantly specify some aspects of the application. For example, rather than specifying security in a case by case basis, it is always convenient and specify a security profile separately and then attach it to the program. How the particular profile applies to a given platform depends on the platform features and may be defined independently of any other features on an application.

### Portability

In this approach developers would only be concerned about the DSL and would not be exposed to any of the platform specific complexities. The use of semantic models help to link to well defined and established concepts at a higher level of abstraction although the DSL provides a more developer friendly representation. The development effort will focus only on the DSL. Although the generated artifacts are platform dependent, they are at a layer transparent to the developers.

Considering the use cases presented in Section 2, if all the applications were indeed built using a platform agnostic DSL, porting the code would be have been effortless given the presence of the relevant generators. To port data, one would have to generate a transformation using the generic data format. These transformations will be using the lifting-lowering mechanism where the data is first transformed to a common format (lifted) and then transformed back to the target format (lowered) [13].

### Manageability

Using the generator based strategy a quadratic explosion of application combinations can be avoided. In the case of MobiCloud the total number of combinations ( $T_c$ ) is

$$T_c = \sum_{i=0}^m \{MV_i\} \times \sum_{j=0}^c \{CV_j\} \quad (1)$$

Where  $m$  is the number of mobile platforms,  $c$  is the number of Cloud platforms,  $MV_i$  is the number of versions of the  $i$ th mobile platform and  $CV_j$  is the number of versions of the  $j$ th Cloud platform.

```

recipe(:todolist) do
  metadata({:id => 'todo-app'},
    {:deployment_profile => "http://www.elastra.com/sites/
    default/files/ExampleApacheStack-full.ttl" })

  model(:todoitem, (:name=>:string, :description =>
    :string,:time => :date, :location => :string))
  model(:user, {:ref => "foaf:Person"})

  controller(:todohandler) do
    action :create,:todoitem,
      {:security_profile => `ssl`}
    action :retrieve,:todoitem
    ...
  end

  view :todo_add, (:models =>[:todoitem],:controller =>
    :todohandler,:action => :create)
  view :todo_show, (:models =>[:todoitem],:controller =>
    :todohandler,:action => :retrieve)
end

```

Reference to an external semantic deployment configuration

Reference to an external data definition

Reference to a Non-functional semantic model  
 Indicating the action needs to be secure

**Figure 4:** Semantically annotated DSL script

However the number of generators that need to be maintained ( $T_g$ ) is

$$T_g = \sum_{i=0}^m \{MV_i\} + \sum_{j=0}^c \{CV_j\} \quad (2)$$

Approximating the real world numbers, assuming there are 4 mobile platforms with 2 versions each and 3 Cloud platforms with 2 versions each, the total combinations that exist is 48 (Equation 1). The total number of generators required is 14 according to (Equation 2). Adding one more Cloud platform with 2 versions increase the combinations by 8 but adds only 2 extra generators. The number of needed generators may be far less than this theoretical maximum since many platform versions are backward compatible. The number of generators grows linearly with the number of new platform additions in a manageable fashion.

## Developer Convenience

The todo list program presented in Figure 3 is in fact a working program and can be compiled using the online tools. Table 2 shows a preliminary program metric comparison for the generated artifacts. Some generated artifacts such as the XML configuration files are not considered for these metrics.

These results clearly indicate that there is a significant amount of relieved effort in creating these applications via the DSL. For example, generating the combination of Android and Google Appengine, the developers only provide approximately 3% of the code they would have written otherwise.

Apart from the amount of generated code, the generators

Application	DSL LoC <sup>a</sup>	Target platform	LoC <sup>b</sup>	$N_C$ <sup>c</sup>	$N_M$ <sup>d</sup>
Todolist	12	Android	225	10	6
		Blackberry	324	8	19
		Amazon EC2	215	5	27
		Google Appengine	158	5	22

- <sup>a</sup>Lines of Code in DSL
- <sup>b</sup>Lines of Code Generated
- <sup>c</sup>Number of Classes
- <sup>d</sup>Number of Methods

**Table 2:** Comparison of Code Metrics for the Generated Application in the Todo list program

also provide the proper organization of files and optionally a build script. This also attributes to developer convenience since it facilitates a convenient way to generate the final executables.

## Efficiency of the Generated Code

DSLs by nature are focused on specific domains and do not offer flexibility when deviating from the target domain. In this case, the code is generated via templates and is functional but may not be optimal for the given platform.

We follow the argument that it is economical to upgrade the hardware than using human effort to optimize the code [2]. This has long been the subject of debate. Many practitioners agree that although there are some cases where code optimization is necessary, it is indeed cheaper to just upgrade the hardware for many of the cases. In this case, even though the generators do not provide the optimum code for

a platform, increasing the available computing power to the given type of application is always economical. The convenience the DSL provides far out weighs the drawbacks of the generated code.

## Limitations

### Deployment Complexity

Deploying the generated artifacts is a complicated work flow. Some of these work flows have been deliberately kept as human centric operations by the vendors. Even when there are Web APIs present, managing keys, security certificates and other deployment operations require the presence of a different layer of automation. Although such facilities are out of scope of this work, adding a middleware layer capable of managing deployments and subsequent management tasks would improve the reach and the usability of the DSL. IBM Altocumulus [11] is one such Cloud middleware that the authors have first hand experience in.

### Feature Abstractions

Many seasoned developers see a lack of flexibility in using a DSL based (model driven) approach. The primary concern is the limiting of the abstractions to the set of smallest common sub set of features. This means that fully portable applications can only be made at the level of the least capable platform. This becomes a serious limitation when the applications need to be well integrated to the target platform. One strategy is to use the DSL to generate the boiler plate code and then customize the generated code. The customization can be loosely attached to avoid overriding by subsequent updates.

In practice, almost all the modern Cloud and mobile platforms have comparable features. Hence the least common subset of features is not significantly different from the available features of a given platform.

## 7 Conclusion

This research clearly indicates that using a DSL shows promise in developing portable applications for the Cloud and other related platforms. We have identified a semantic separation of concerns for applications and demonstrated how these can be used with an experimental DSL. Although there are many details to iron out, the preliminary results are very encouraging and we would be investigating the use of semantics for Cloud application development further.

## References

- [1] D. Amrhein, P. Anderson, A. de Andrade, J. Armstrong, B. Arasan, R. Bruklis, K. Cameron, R. Cohen, A. Easton, R. Flores, et al. Cloud Computing Use Cases.
- [2] J. Atwood. Hardware is Cheap, Programmers are Expensive, 2008. Available online at <http://bit.ly/avyNiN> - Last accessed Spt 3rd 2010.
- [3] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
- [4] F. Chong, G. Carraro, and R. Wolter. Multi-Tenant Data Architecture. *MSDN Library, Microsoft Corporation*, 2006.
- [5] D. Durkee. Why cloud computing will never be free. *Communications of the ACM*, 53(5):62–69, 2010.
- [6] T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.
- [7] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM.
- [8] D. Hanselman and B. Littlefield. *Mastering MATLAB 5: A comprehensive tutorial and reference*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.
- [9] A. Manjunatha, A. Ranabahu, P. Anderson, S. S. Sahoo, M. Raymer, and A. Sheth. Cloud based scientific workflow for nmr data analysis, 2010. Conference poster.
- [10] A. Manjunatha, A. Ranabahu, A. Sheth, and K. Thirunarayan. A Domain Specific Language Based Method to Develop Cloud-Mobile Hybrid Applications. Technical report, Kno.e.sis Center, Wright State University, 2010. Available online at <http://knoesis.wright.edu/library/publications/MobiCloud.pdf> : Last accessed August 27th 2010.
- [11] E. Maximilien, A. Ranabahu, R. Engehausen, and L. Anderson. Toward cloud-agnostic middlewares. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 619–626. ACM, 2009.
- [12] P. Mell and T. Grance. NIST Definition of Cloud Computing v15. *National Institute of Standards and Technology*, 2009.
- [13] M. Nagarajan, K. Verma, A. Sheth, J. Miller, and J. Lathem. Semantic interoperability of web services-challenges and experiences. 2006.
- [14] N. Oldham, K. Verma, A. Sheth, and F. Hakimpour. Semantic WS-agreement partner selection. In *Proceedings of the 15th international conference on World Wide Web*, page 706. ACM, 2006.
- [15] A. Sheth. Semantic Web Process Lifecycle: Role of Semantics in Annotation, Discovery, Composition and Orchestration. In *Workshop on E-Services and the Semantic Web (ESSW 03) in 12th International World Wide Web (WWW) Conference*, Budapest, Hungary, 2003. Invited Presentation.
- [16] A. Sheth and A. Ranabahu. Semantic modeling for cloud computing, part 1. *IEEE Internet Computing*, 14:81–83, 2010.
- [17] K. Sivashanmugam, A. Sheth, J. Miller, K. Verma, R. Aggarwal, and P. Rajasekaran. Metadata and semantics for Web services and processes. *Databases and Information Systems*, 60:245–271, 2003.
- [18] F. Steimann. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497, 2006.