A Metamodel for Distributed Ensembles of Virtual Appliances

Xabriel J. Collazo-Mojica and S. Masoud Sadjadi
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
Email: {xcoll001, sadjadi}@cs.fiu.edu

***Abstract*—We present our work on modeling distributed ensembles of virtual appliances (DEVAs) on Infrastructure as a Service (IaaS) clouds. Designing solutions on IaaS providers require a good understanding of the underlying details such as the software installation or the network configuration. We propose the use of DEVAs, a modeling approach built on top of the notion of virtual appliances, that allows easy-to-compose and ready-to-use cloud application architectures that are IaaS-agnostic, and that abstract away unnecessary details for web application developers. In this paper, we extend the definition of a DEVA from previous work by presenting an underlying metamodel and how that metamodel can be transformed to an actual deployment. We also present a case study where we model a web application architecture and we discuss how we can instantiate it in an IaaS cloud. We argue that the DEVA modeling approach is suitable for typical cloud use cases.**

***Index Terms*—virtual appliance, non-functional requirements, cloud computing architectures.**

I. Introduction

In this paper, we extend our work on modeling groups of interdependent virtual machines in the cloud. Designing these compositions require a good understanding of the underlying details such as the software installation or the network configuration. They are typically deployed in a cloud layer called Infrastructure as a Service (IaaS). Each IaaS provider has its own API to configure the virtual machines and its connections, requiring users to learn the details of yet another API. Similarly, manually installing software stacks in these virtual machines is a tedious task. In [1], we presented a visual modeling approach to make these architectures easy-to-compose and ready-to-use. We call these models *distributed ensembles of virtual appliances* (DEVAs)[1]. We now extend our work by presenting an underlying meta-model for DEVAs.

Sapuntzakis et al. presented in 2003 the idea of a virtual appliance, effectively treating full stacks of software applications and OS as updatable image files [2]. These virtual appliance files could then be cloned in bare metal computers, or instantiated in virtual machines. Still, when the time comes to compose multi appliance systems with interdependencies, Sapuntzaki's implementation relied on defaults from software vendors for most of the configurations. This may not be the case for web applications (e.g., when trying to configure interdependencies between a database server and its clients). Similarly, the configuration of the network connection between appliances had to be done by hand. Various recent attempts at automatically configuring virtual appliances and their network dependencies have been presented [3], [4], [5], but they all rely on having experts on appliance configurations, or on specific IaaS providers.

In [1], we proposed that with proper modeling of these kind of scenarios, the configuration problems could be abstracted away. By designing DEVAs, non-expert users can easily architect interdependent virtual appliances. DEVA models include quality of service (QoS) constraints, which can account for the non-functional requirements of the modeled architecture. In this paper, we extend our modeling approach by 1) presenting an underlying meta-model, and 2) how a model generated from the meta-model can be instantiated to an actual deployment.

The main technical challenge for this work was to come up with a meta-model with sufficient details to be able to instantiate DEVAs on the spectrum of current IaaS providers. To demonstrate our modeling approach, we present a case study where we model a web application architecture and discuss how we can instantiate it in a current IaaS provider.

We argue that the DEVA modeling approach is suitable for typical cloud use cases, and that having a model of a cloud architecture can simplify co-allocations and migrations across-different IaaS providers. In Section II, we present background information on the methodology used and on our previous work. In Section III, we present our metamodeling framework. In Section IV-A, we discuss a system that can transform DEVA models into instances, and in Section IV-B, we present a simple case study. In Section V, we discuss related work, and finally, in Section VI, we enumerate some concluding remarks.

II. Background

In this section, we present background information on model-driven engineering, the methodology used for this work, and present details on previous work including the definition of DEVAs.

[1]Note that in [1] we called our models *virtual environments* instead of DEVAs. We have desisted of the previous name as it is already used in other CS areas.

A. Model-Driven Engineering

MDE is a methodology that aims to effectively apply models to software development. Instead of looking at models as simple diagrams, MDE captures the problem domain in a modeling language. This language can later be used to generate solutions that can be automatically generated. MDE has many goals, including accelerated development, automated transformations, and platform independence [6].

In this work, we utilize the MDE methodology to present a modeling approach that simplifies cloud architecture design, as well as to achieve platform independence from IaaS providers. In addition, we use MDE's metamodeling approach as presented in [6] to realize an abstract syntax definition for our models. Static semantics are presented in Object Constraint Language (OCL) notation [7].

B. Distributed Ensembles of Virtual Appliances

A *distributed ensemble of virtual appliances* is a model of a logical architecture of interdependent virtual appliances. From the point of view of users of DEVAs, these models should present the following properties:

1) **Easy to understand**: Views for the design, deployment, change management, and monitoring of these architectures should only present what is strictly necessary to realize them. Advanced options should be available but normally hidden.
2) **Self-configurable**: Once the user has specified a description of the architecture needed, our solution should be able to instantiate the model and configure all the details automatically by following the constraints and policies specified.
3) **Present deployment choices**: Modeling should be abstract enough to allow for an implementation to present deployment choices. Given the heterogeneity of current IaaS APIs, this is one of the main challenges of our approach.

C. Visual Concrete Syntax for DEVAs

Research has shown the benefits of visual aids when designing system architectures, claiming a gain of over 60% in comprehension [8]. Since our target users are web developers, which may not be experts on architectures for the cloud, a graphical representation is desirable. We model virtual appliances with boxes with *service endpoints*. The boxes represent all the necessary software, the OS and the configuration necessary to support the services provided or consumed. Figure 1 presents an example of a DEVA composed of two virtual appliances. The box entitled 'RoR Node' is a representation of an appliance provisioned with the Ruby on Rails web framework (and all other needed software). Similarly, the box entitled 'MySQL DB' is an appliance provisioned with a MySQL database. These appliances have been interconnected with a 'db' link by joining the corresponding endpoints. This connection assumes that any interdependent configuration will be resolved by our solution. For example, to be able to provide the db service, a username and password must be agreed by the db provider and db consumers. In our approach, there is no need to configure IP addresses, ports or configuration files.

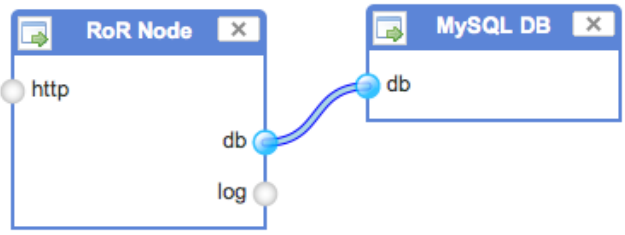


Fig. 1. An example of a simple DEVA as presented in [1].

III. A Metamodeling Framework for DEVAs

In [1], we investigated the first property of our model by presenting an uncomplicated visual concrete syntax to design DEVAs. In this paper, we investigate the second property. We realized that for our solution to be able to model arbitrarily complex cloud architectures, we needed a formal definition of our modeling approach. The third property, to be able to present deployment choices, remains as future work.

Having an underlying metamodeling framework for DEVAs facilitates the steps necessary to go from a model to an instance in a similar way that the Meta-Object Facility (MOF) defines the UML language. Figure 2 presents an overview of the framework. Note that to keep our modeling as simple as possible, we do not currently use the MOF, or UML Profiles to define our framework.

At the top of the figure, we have the DEVA Metametamodel. This level defines what the valid constructs are for our modeling approach. Based on this constructs, we propose two different DEVA metamodels with the main difference being that one allows the instantiation of resource-independent (RI-DEVA) models, and the other allows resource-dependent (RD-DEVA) models. Resources in this paper mean computational resources as we would typically obtain from a IaaS provider, such as CPU allocations, RAM memory, and network connection speed and IP addresses.

We make this resource dependency distinction for two main reasons. First, to present the user with a simple designing tool that separates the concern of modeling a DEVA, and the concern of modeling the resources needed to run such DEVA. By having a RI-DEVA, a user can let our framework allocate the required resources based on the specified high-level policies and constraints. We can achieve this by transforming an RI-DEVA to a RD-DEVA using model to model (M2M) transformations [6].

Second, the resource performance from IaaS providers has been shown to have high variability [9]. Similarly, different IaaS providers have different metrics for specifying available resources. Thus, we believe it is desirable to model application architectures by quantifiable SLA constraints (*i.e.,* "able to do 100 transactions per second"), rather than subjective resource

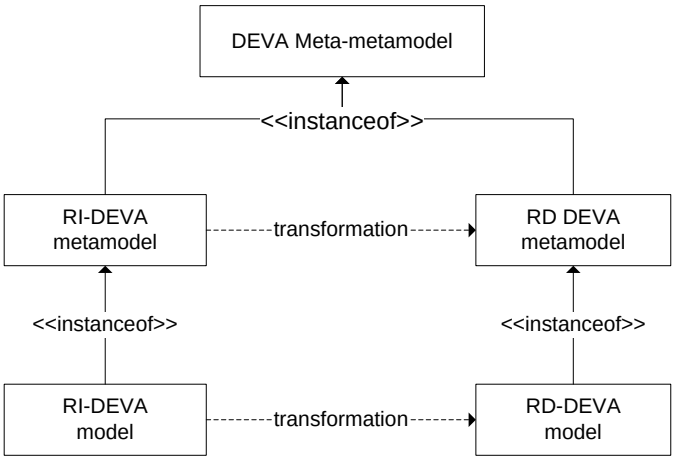
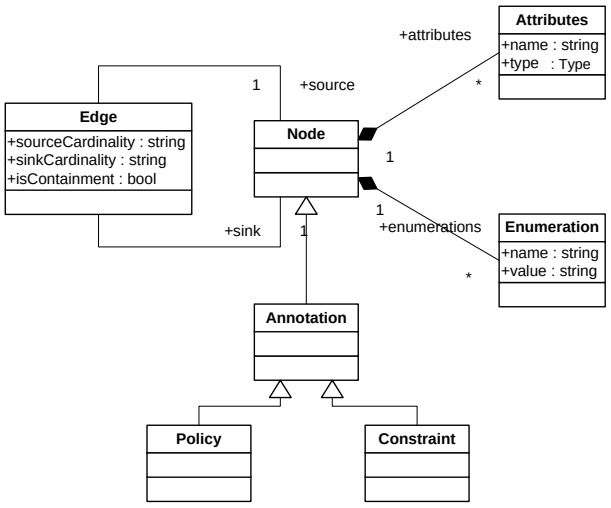


Fig. 2. Metamodeling Framework for DEVAs.



(a) Abstract syntax for DEVA meta-metamodel.

```
context Node
inv: self.attributes.size == 0 or
     self.enumerations.size == 0

context Edge
inv: self.source != self.sink
```

(b) Static semantics for DEVA meta-metamodel.

Fig. 3. Meta-metamodel for DEVAs.

metrics (*i.e.,* "having an equivalent of a 1GHz CPU with 256MB RAM") as IaaS providers currently offer [10]. Our main concern is to allow developers to specify RI-DEVAs that are transformed to RD-DEVAs by our system.

A. DEVA Meta-metamodel

Our proposed DEVA meta-metamodel abstract syntax is presented on Figure 3(a), and the static semantics on Figure 3(b). At this level, we can construct simple graphs, with Nodes and Edges. This metamodel is a simple extension of a graph which allows Annotations against Nodes. These Annotations can be either Policies or Constraints. Each of the Nodes can have a list of Attributes, or a list of Enumerations, but not both. Edges have a source Node and a sink Node, which cannot be the same Node. An attribute is composed by a name and a type. Enumerations have names and literal values. With these basic constructs, we can generate the metamodels for both RI-DEVAs and RD-DEVAs, which we describe below.

B. RI-DEVA Metamodel

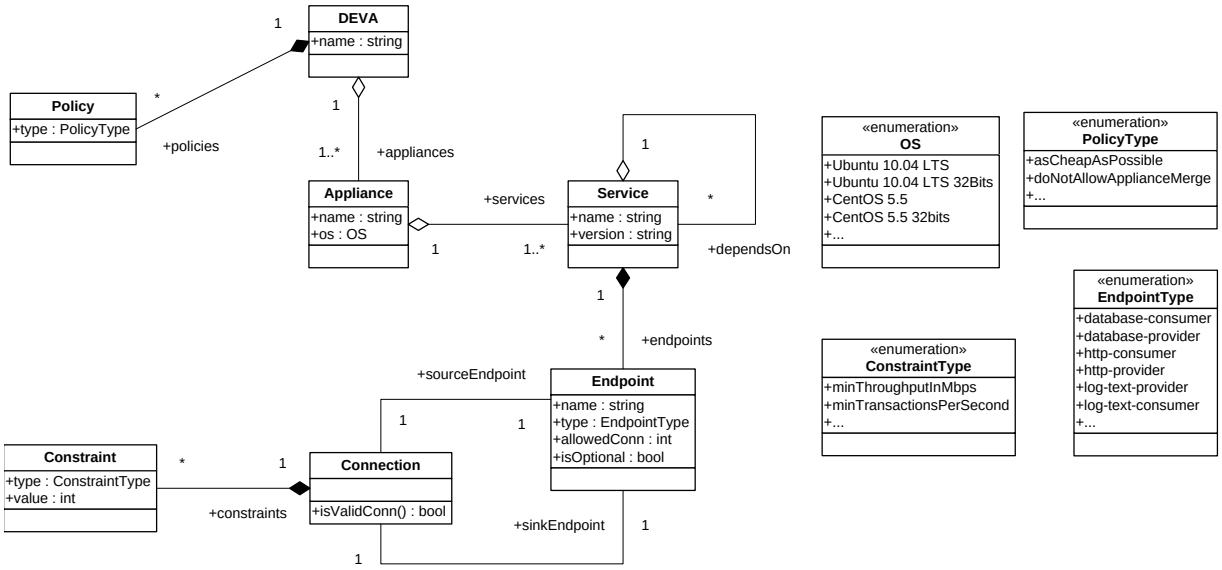
Our proposed RI-DEVA meta-model abstract syntax is presented in Figure 4(a), and the static semantics in Figure 4(b). An RI-DEVA is a named composition of zero or more global Policies and one or more virtual Appliances. In this context, Policies refer to high-level guidelines that a DEVA instance will try to meet. An example of a policy is to be able to instantiate a RI-DEVA with the least amount of resources possible (*i.e.* 'asCheapAsPossible' in our PolicyType enumeration). This means that economy will have a bigger weight whenever we try to realize this RI-DEVA. A RI-DEVA could include policies that are contradicting. In such cases, they would be accommodated in an arbitrary order.

At least one virtual appliance is necessary to convey a RI-DEVA. Note again that our Appliance definition is isolated completely from resource usage. These Appliances are represented by a name, a specific operating system, and a list of one or more services.

Each service is identified by a name, a version number, a list of the services it depends on (if any), and a set of zero or more endpoints. These endpoints are key to our approach, as can be seen in Figure 1. They allow to specify service production or consumption at the service level, rather than specifying network level details. This approach implicitly generates meta-data about the architecture that allows for easier appliance provisioning, as all the connection and service-to-service dependencies are known *a priori*.

Endpoints are represented by a name, a type, and an integer that specifies how many connections are allowed. A connection is composed of a source endpoint linked to a sink endpoint; sink and source cannot be the same endpoint. Connections should have valid type matches at each endpoint. Going back to Figure 1, a database-consumer can only be connected to a database-provider, and the provider endpoint can specify a maximum number of consumer connections. For this, we include a function $isValidConn : EndpointType \times EndpointType \rightarrow bool$ which determines Connection validity based on Endpoint type and their allowedConn property. Endpoints also specify if they can be left unused, or if a connection is necessary.

Each connection can have zero or more Constraints. These Constraints specify QoS non-functional requirements for the service-to-service connections. For example, in the case of a database connection, a constraint "at least 50 transactions per second" could be applied. Again, this meta-data could be used in the model transformation to try to guarantee the constrained connection performance.



(a) Abstract syntax for RI-DEVAs.

```
context Connection
inv: sourceEndpoint != sinkEndpoint and isValidConn()

context Service
inv: self.dependsOn->forAll(dep: Service | dep != self)

context Constraint
inv: self.value >= 0

context Endpoint
inv: self.allowedConn > 0
```

(b) Static semantics for RI-DEVAs.

Fig. 4. Meta-metamodel for RI-DEVAs.

C. RD-DEVA Metamodel

Figure 5(a) presents a partial abstract syntax for RD-DEVAs, and Figure 5(b) the static semantics. These models have a similar metamodel as RI-DEVAs. The main difference is that we do not have policies nor constraints; rather, we have IaaS resources mapped to each Appliance. These Appliances must be mapped to 3 or more resources, and there should be at least one resource of CPU, RAM, and NIC, respectively.

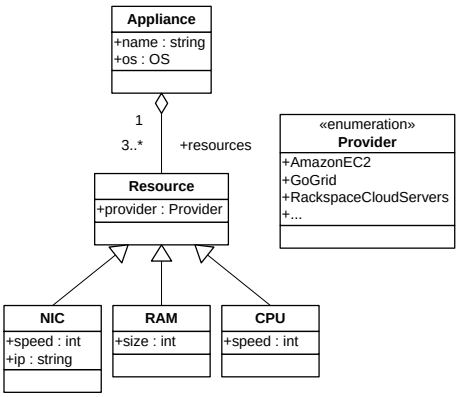
In this paper, we focus on how to construct RI-DEVAs and how to transform them to RD-DEVAs. Work on directly modeling RD-DEVAs is presented elsewhere [11].

IV. Transforming DEVAs

A. A model to instance reference system

Figure 6 presents an overview of a system that can transform DEVA models into instances. As input, the system would have a RI-DEVA model. This model would be received by a Model to Model (M2M) transformation engine aware of the metamodels for RI- and RD-DEVAs. The transformation between these models can be done in various ways. The task is to translate quantifiable constraints and policies to available resources in an IaaS cloud. This could be realized by having a mapping $Instantiate : ConstraintType \times Constraint.value \longrightarrow < R_{CPU}, R_{RAM}, R_{NIC} >$ applied to all of the services in a specific virtual appliance, where $< R_{CPU}, R_{RAM}, R_{NIC} >$ is a tuple representing the allocated resources in a specific IaaS cloud. This mapping could be implemented by using resource usage predictions based on empirical data in a black box manner, as in previous work where we model resources used by specific software [12], or in a white box approach as presented in [13]. The results of the mapping can then be aggregated to have the total needed resources for an Appliance.

Once we have a RD-DEVA, we need to have a DEVA model to instance (M2I) transformation engine that is aware of the IaaS resources. Again, we have various choices. Either the engine could be an IaaS provider which is aware of DEVAs, as in [11], or the engine could be a middleware which translates RD-DEVA models to IaaS API calls. To resolve interdependencies and to provision the appliances with the necessary software, this middleware would generate configuration scripts



(a) Partial abstract syntax for RD-DEVAs.

```
context Appliance
inv: self.resources->
          exists(c | c.oclIsKindOf(CPU))
inv: self.resources->
          exists(r | r.oclIsKindOf(RAM))
inv: self.resources->
          exists(n | n.oclIsKindOf(NIC))
```

(b) Partial static semantics for RD-DEVAs.

Fig. 5. Partial metamodel for RD-DEVAs. All other entities are as in RI-DEVAs if we eliminate the Policies and Constraints.

which would then be run on the instantiated Virtual Machines (VM) on top of the IaaS provider. Note that in our approach, a different M2I engine would be needed for each target IaaS provider. Nonetheless, some work from this engine can be modularized. For example, to make the software provisioning process repeatable, a software installation utility like Chef could be used [14].

B. A Simple Case Study

Because of space constraints, we present a simple case study of our approach based on Figure 1. A small composition like this can be instantiated as follows. First the user would define this RI-DEVA by using our prototype DEVA designer. We chose to have a web-based designer for DEVAs to provide a cross-platform solution. The model would then be sent to the M2M engine. Note that we can always compare the DEVA being designed against the metamodel to find discrepancies, and acknowledge the designer accordingly.

Since the model in Figure 1 does not present any specific Constraints, our system would choose the simplest transformation, which could be a direct mapping to, say, two Virtual Machines in Amazon EC2 of the type "small instance". A small instance in Amazon EC2 provides one CPU allocation, with 1.7GB of RAM, and a basic network connection with two IPs, one public and one private [10]. This would be the transformation to an RD-DEVA.

Now that we have an RD-DEVA, we can send the model to the M2I engine. On this engine, we can instantiate the architecture by making the IaaS-specific API calls necessary to launch 2 VMs, and then provisioning the software of each virtual appliance. After this, we would need to resolve the interdependencies (such as username and password for the database) by running configuration scripts on the instantiated VMs. Note that the meta-data of the Connections could be used in this step. For example, on Amazon EC2, the private IP could be used to make the DB connection, instead of using the public one. Similarly, we also know that there is only one connection possible to the database server. Therefore, the instance could be configured to reject any connection other than to the DB port.

V. Related Work

We have identified the need for better abstractions from the current IaaS implementations provided by vendors such as Amazon [10] or GoGrid [15]. Thus, we propose a modeling approach that is abstract enough to allow these interdependent systems to be easy-to-design, fast-to-deploy, and that limits the effects of IaaS vendor lock-in. We do not currently use a standardized IT information model, such as DMTF's Common Information Model [16], as current IaaS providers do not support them.

Commercial applications implementing a similar modeling approach are available [17], [18]. They only offer closed-source implementations and only work on their proprietary cloud platforms. IBM has worked on a similar project, but their implementation assumes that users are experts in the domains of virtual image provisioning, image composition, and composition deployment [3]. While they target enterprise customers, we target non-expert cloud users.

Platform as a Service (PaaS) providers, such as Google AppEngine [19], abstract away the underpinnings of a fully working web application. Of course, this means that the user has to learn the vendor's API, and that migrating the application to other PaaS provider implies changing most of the implementation. Our work envisions models that once specified do not need to be changed because of a vendor switch.

Recently, Amazon started to offer a service called CloudFormation. Customers can now specify groups of virtual machines with provisioning scripts that provide repeatable architecture instantiation. Compared to our solution, Amazon's offering only work on their IaaS service and of course the details of their API need to be well known. Nonetheless, this is a welcomed addition to their cloud offering, and validates the current and future importance of easier-to-compose solutions for the cloud.

VI. Concluding Remarks

In this paper, we presented a modeling approach for DEVAs. First, we defined a metamodeling framework to be able to formulate two metamodels, one for RI-DEVAs, and the other for RD-DEVAs. We argued that RI-DEVAs are more desirable for describing cloud architectures, and that an automatic transformation between these two metamodels allows developers to specify DEVAs with quantifiable QoS constraints rather than

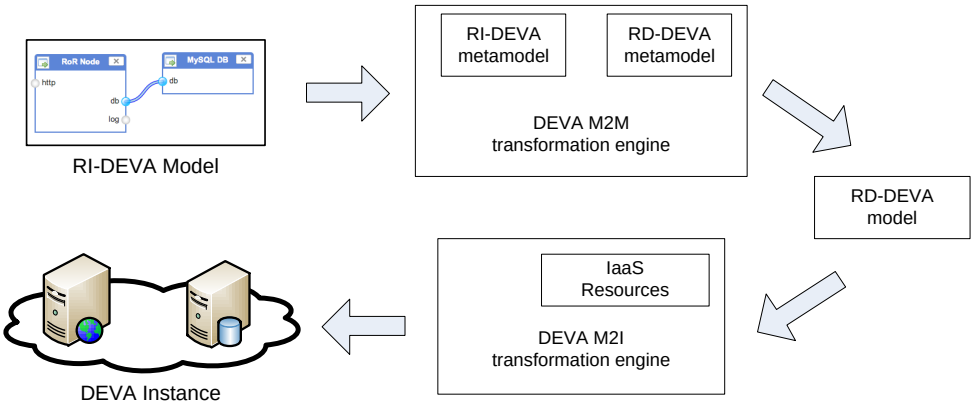


Fig. 6. Transforming DEVA models to DEVA instances.

by subjective performance metrics provided by current IaaS providers.

We then presented the details of our framework, by describing RI-DEVAs and RD-DEVAs with their abstract syntax and static semantics. We also described our key modeling abstraction, which consist of making service-to-service connections between Appliances, so that the architecture developer does not have to specify network level details. This abstraction aids in two ways: network level details may not be the developer's expertise area, and also its configuration can vary between different IaaS APIs.

We also presented how a system could implement our modeling approach, by describing the various stages needed to transform a RI-DEVA modeled by a developer to an actual instance in an IaaS Provider. Finally, a simple case study of how a typical web architecture could be modeled with our approach was discussed.

Designing solutions on top of IaaS providers is a growing problem domain in the cloud. A modeling approach such as the one presented can potentially make these architectures easy-to-compose and ready-to-use. We are in the process of prototyping a system that follows the details presented in Section IV. We are also investigating how to expand our approach to monitor and autonomically enforce Constraints and Policies of instantiated DEVAs.

Acknowledgment

We appreciate the discussions held with David Villegas. This work was supported in part by a GAANN Fellowship from the US Department of Education under P200A090061 and in part by the National Science Foundation under Grant No. OISE-0730065.

References

[1] X. J. Collazo-Mojica, S. M. Sadjadi, F. Kon, and D. D. Silva, "Virtual environments: Easy modeling of interdependent virtual appliances in the cloud," *SPLASH 2010 Workshop on Flexible Modeling Tools*, Aug 2010.

[2] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software," *LISA '03: Proceedings of the 17th USENIX Large Installation Systems Administration Conference*, pp. 181–194, Aug 2003.

[3] A. Konstantinou, T. Eilam, M. Kalantar, A. Totok, W. Arnold, and E. Snible, "An architecture for virtual solution composition and deployment in infrastructure clouds," *VTDC '09: Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing*, Jun 2009.

[4] T. Chen, Y. Wang, Y. Ren, C. Luo, D. Qian, and Z. Luan, "R-ECS: reliable elastic computing services for building virtual computing environment," *ICIS '09: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, Nov 2009.

[5] K. Keahey and T. Freeman, "Contextualization: Providing one-click virtual clusters," *IEEE Fourth International Conference on eScience*, pp. 301–308, 2008.

[6] T. Stahl and M. Voelter, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[7] "Object Constraint Language," March 2011. [Online]. Available: http://www.omg.org/spec/OCL/2.2/

[8] J. Knodel, D. Muthig, and M. Naab, "Understanding software architectures by visualization–an experiment with graphical elements," *WCRE '06: 13th Working Conference on Reverse Engineering*, pp. 39–50, 2006.

[9] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, *A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2010, vol. 34, pp. 115–131.

[10] "Amazon Elastic Compute Cloud," March 2011. [Online]. Available: http://aws.amazon.com/ec2/

[11] D. Villegas and S. M. Sadjadi, "DEVA: Distributed ensembles of virtual appliances in the cloud," Florida International University - School of Computer and Information Sciences, Tech. Rep. FIU-SCIS-2011-03-22, March 2011.

[12] S. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo-Mojica, "A modeling approach for estimating execution time of long-running scientific applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1 –8.

[13] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 87 – 92, 2010.

[14] "Chef Systems Integration Framework," March 2011. [Online]. Available: http://wiki.opscode.com/display/chef/Home

[15] "GoGrid Cloud Hosting," March 2011. [Online]. Available: http://www.gogrid.com/cloud-hosting/cloud-servers.php

[16] "DMTF's Common Information Model," May 2011. [Online]. Available: http://www.dmtf.org/standards/cim

[17] "3Tera Inc." March 2011. [Online]. Available: http://www.3tera.com/

[18] "Elastra Corporation," March 2011. [Online]. Available: http://www.elastra.com/

[19] "Google App Engine," March 2011. [Online]. Available: http://code.google.com/appengine/