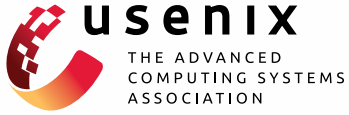

Compiling Abstract Specifications into Concrete Systems—Bringing Order to the Cloud

Ian Unruh, Alexandru G. Bardas, Rui Zhuang, Xinming Ou, and Scott A. DeLoach, *Kansas State University*

https://www.usenix.org/conference/lisa14/conference-program/presentation/unruh

This paper is included in the Proceedings of the 28th Large Installation System Administration Conference (LISA14).

November 9–14, 2014 • Seattle, WA

ISBN 978-1-931971-17-1

Open access to the Proceedings of the 28th Large Installation System Administration Conference (LISA14) is sponsored by USENIX

Compiling Abstract Specifications into Concrete Systems – Bringing Order to the Cloud

Ian Unruh
Kansas State University
iunruh@ksu.edu

Alexandru G. Bardas
Kansas State University
bardasag@ksu.edu

Rui Zhuang
Kansas State University
zrui@ksu.edu

Xinming Ou
Kansas State University
xou@ksu.edu

Scott A. DeLoach
Kansas State University
sdeloach@ksu.edu

Abstract

Currently, there are important limitations in the abstractions that support creating and managing services in a cloud-based IT system. As a result, cloud users must choose between managing the low-level details of their cloud services directly (as in IaaS), which is time-consuming and error-prone, and turning over significant parts of this management to their cloud provider (in SaaS or PaaS), which is less flexible and more difficult to tailor to user needs. To alleviate this situation we propose a high-level abstraction called the *requirement model* for defining cloud-based IT systems. It captures important aspects of a system's structure, such as service dependencies, without introducing low-level details such as operating systems or application configurations. The requirement model separates the cloud customer's concern of *what* the system does, from the system engineer's concern of *how* to implement it. In addition, we present a "compilation" process that automatically translates a requirement model into a concrete system based on pre-defined and reusable knowledge units. When combined, the requirement model and the compilation process enable repeatable deployment of cloud-based systems, more reliable system management, and the ability to implement the same requirement in different ways and on multiple cloud platforms. We demonstrate the practicality of this approach in the *ANCOR* (Automated eNterprise network COmpileR) framework, which generates concrete, cloud-based systems based on a specific requirement model. Our current implementation[1] targets OpenStack and uses Puppet to configure the cloud instances, although the framework will also support other cloud platforms and configuration management solutions.

[1]Current ANCOR implementation is available and is distributed under the GNU (version 3) General Public License terms on: https://arguslab.github.io/ancor/

Tags: cloud, modeling networking configuration, configuration management, deployment automation

1 Introduction

Cloud computing is revolutionizing industry and reshaping the way IT systems are designed, deployed and utilized [3]. However, every revolution has its own challenges. Already, companies that have moved resources into the cloud are using terms like "virtual sprawl" to describe the mess they have created [38]. Cloud services are currently offered in several models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). While these options allow customers to decide *how much* management they want to perform for their cloud-based systems, they do not provide good *abstractions* for effectively managing those systems or addressing diverse user needs.

IaaS solutions such as Amazon Web Services (AWS) and OpenStack allow cloud users to access the raw resources (compute, storage, bandwidth, *etc.*); however, it forces users to manage the software stack in their cloud instances at a low level. While this approach gives users tremendous flexibility, it also allows the users to create badly configured or misconfigured systems, raising significant concerns (especially related to security) [5, 7]. Moreover, offering automatic scalability and failover is challenging for cloud providers because replication and state management procedures are application-dependent [3]. On the other hand, SaaS (also known as "on-demand software") provides pre-configured applications to cloud users (*e.g.*, SalesForce and Google Apps). Users typically choose from a set of predefined templates, which makes it difficult to adequately address the range of user needs. PaaS (*e.g.*, Google App Engine, Heroku, and Windows Azure) is somewhere in the middle, offering computing platforms with various pre-installed operating systems as well as services and allowing users to deploy their own applications as well.

As PaaS is a compromise between IaaS and SaaS, it also inherits the limitations of both to various degrees. For example, users can be easily "locked in" to a PaaS vendor, like in SaaS, and the configuration of applications is still on the users' shoulders, like in IaaS.

We observe that existing cloud service models suffer from the lack of an appropriate *higher-level abstraction* capable of capturing objectives and functionality of *the complete IT system*. Such an abstraction, if designed well, can help both the creation and the long-term maintenance of the system. While there have been attempts at providing abstractions at various levels of cloud-based services, none have provided an abstraction that both separates user requirements from low-level platform/system details and provides a global view of the system. This has limited the usefulness of those solutions when it comes to long-term maintenance, multi-platform support, and migration from one cloud provider to another. We believe to be effective, the *abstraction* should exhibit the following properties.

1. It must be capable of representing *what* a user needs instead of low-level details on *how* to implement those needs. A major motivation for using cloud infrastructures is to outsource IT management to a more specialized workforce (called *system engineers* hereafter). Communicating *needs* from users to engineers is better served using higher-level abstractions as opposed to low-level system details.

2. It must support automatic compilation into valid concrete systems on different cloud infrastructures. Such compilation should use well-defined knowledge units built by the system engineers and be capable of translating a specification based on the abstraction (*i.e.*, an *abstract specification*) into different concrete systems based on low-level implementation/platform choices.

3. It should facilitate the long-term maintenance of the system, including scaling the system up/down, automatic fail over, application update, and other general changes to the system. It should also support orchestrating those changes in a secure and reliable manner and aid in fault analysis and diagnosis.

We believe such an abstraction will benefit all three existing cloud service models. For IaaS, an abstract specification will act as a common language for cloud users and system engineers to define the system, while the compilation/maintenance process becomes a tool that enables system engineers to be more efficient in their jobs. Re-using the compilation knowledge units will also spread the labor costs of creating those units across a large number of customers. In the SaaS model the system engineers will belong to the cloud provider so the abstract specification and the compilation/maintenance process will help them provide better service at a lower cost. In the PaaS model we foresee using the abstraction and compilation process to stand up a PaaS more quickly than can be done today. This could even foster the convergence to a common set of PaaS APIs across PaaS vendors to support easier maintenance and migration between PaaS clouds.

There are multiple challenges in achieving this vision. The most critical is whether it is feasible to design the abstraction so that it can capture appropriate system attributes in a way that is meaningful to users and system engineers while being amenable to an automated compilation process that generates valid concrete systems.

The ***contributions*** of our work are:

- We design an abstract specification for cloud-based IT systems that separates user requirements from system implementation details, is platform-independent, and can capture the important aspects of a system's structure at a high level.

- We design a compilation process that (1) translates the high-level specification into the low-level details required for a particular choice of cloud platform and set of applications, and (2) leverages a mature configuration management solution to deploy the resulting system to a cloud.

- We show that maintaining an abstract specification at an appropriate level enables users to address automatic scaling and failover even though these processes are highly application-dependent, and supports a more reliable and error-free orchestration of changes in the system's long-term maintenance.

To demonstrate the efficacy of our approach, we implemented and evaluated a fully-functional prototype of our system, called *ANCOR* (Automated eNterprise network COmpileR). The current implementation of ANCOR targets OpenStack [45] and uses Puppet [20] as the configuration management tool (CMT); however, the framework can also be targeted at other cloud platforms such as AWS, and use other CMT solutions such as Chef [34].

The rest of the paper is organized as follows. Section 2 explains the limitations of current solutions as well as the enabling technologies used in this work. Section 3 presents an overview of the framework, including the proposed abstraction and the compilation workflow. Section 4 describes the implementation of the ANCOR framework and its evaluation. We discuss some other relevant features of the approach and future work in Section 5, followed by related work and conclusion.

2 Background

2.1 Limitations of Available Automation and Abstraction Technologies

Recent years have seen a proliferation of cloud management automation technologies. Some of these solutions (*e.g.,* AWS OpsWorks) tend to focus on automation as opposed to abstraction. They include scripts that automatically create virtual machines, install software applications, and manage the machine/software lifecycle. Some are even able to dynamically scale the computing capacity [36, 38, 39]. Unfortunately, none of these solutions provide a way to explicitly document the dependencies between the deployed applications. Instead, dependencies are *inferred* using solution-specific methods for provider-specific platforms. Not only is this unreliable (*e.g.*, applications may have non-standard dependencies in some deployments), but it lacks the capability to *maintain* the dependency after the system is generated. Ubuntu Juju [41] is a special case that is described and discussed in Section 6 (Related Work).

Recent years have also seen a general movement towards more abstractions at various levels of cloud services, especially in PaaS. Examples include Windows Azure Service Definition Schema (.csdef) [57] and Google AppEngine (GAE) YAML-based specification language [16]. These abstractions are focused on a particular PaaS, thus they have no need to separate the platform from user requirements. Rather, they simply abstract away some details to make it easier for users to use the particular platform to deploy their apps. The abstractions only capture applications under the users' control and do not include platform service structures. As a result the abstractions *cannot* support compiling abstract specifications to different cloud platforms[2].

Systems like Maestro [22], Maestro-NG [23], Deis [9], and Flynn [15] are based on the Linux Containers [21] virtualization approach (specifically the Docker open-source engine [10]). Some of the description languages in these systems (specifically Maestro and MaestroNG) can capture dependencies among the containers (applications) through named channels. However, these specifications abstract *instances* (virtual machines), as opposed to the whole system. There is no formal model to define a globally consistent view of the system, and as a result once a system is deployed it is challenging to perform reliable configuration updates. Current Docker-based solutions are primarily focused on the initial configuration/deployment; maintenance is usually not addressed or they resort to a re-deployment process.

The lack of a consistent high-level abstraction describing the complete IT system creates a number of challenges in configuring cloud-based systems: network deployments and changes cannot be automatically validated, automated solutions are error-prone, incremental changes are challenging (if not impossible) to automate, and configuration definitions are unique to specific cloud providers and are not easily ported to other providers.

[2]Indeed, it appears that these abstractions will likely make it harder for users to move to other cloud providers as they are platform-specific.

2.2 Enabling Technologies

Several new technologies have facilitated the development of our current prototype. In particular, there have been several advancements in the configuration management tools (CMT) that help streamline the configuration management process. This is especially beneficial to our work, since those technologies are the perfect building blocks for our compilation process. To help the reader better understand our approach, we present a basic background on the state-of-the-art CMTs.

Two popular configuration management solutions are Chef [34] and Puppet [20]. We use Puppet but similar concepts exist in Chef as well. Puppet works by installing an agent on the host to be managed, which communicates with a controller (called the master) to receive configuration directives. Directives are written in a declarative language called Puppet *manifests*, which define the *desired configuration state* of the host (*e.g.*, installed packages, configuration files, running services, *etc*.). If the host's current state is different than the manifest received by the Puppet agent, the agent will issue appropriate commands to bring the system into the specified state.

In Puppet, manifests can be reused by extracting the directives and placing them in *classes*. Puppet classes use parameters to separate the configuration data (*e.g.*, IP addresses, port numbers, version numbers, *etc*.) from the configuration logic. Classes can be packaged together in a Puppet module for reuse. Typically, classes are bound to nodes in a master manifest known as the *site manifest*. Puppet can also be configured to use an external program such as External Node Classifier (ENC) [18] or Hiera [35] to provide specific configuration data to the classes that will be assigned to a node.

In the current prototype we use Hiera [35], which is a key/value look-up tool for configuration data. Hiera stores site-specific data and acts as a site-wide configuration file, thus separating the specific configuration information from the Puppet modules. Puppet classes can be populated with configuration data directly from Hiera, which makes it easier to re-use public Puppet modules "as is" by simply customizing the data in Hiera. Moreover, users can publish their own modules without worrying about exposing sensitive environment-specific data or clashing variable names. Hiera also supports module

```
class role::work_queue::default {

  $exports = hiera("exports")

  class { "profile::redis":
    port => $exports["redis"]["port"]
  }
}
```

Figure 1: Puppet Worker Queue Class

```
classes:
  - role::work_queue::default

exports:
  redis: { port: 6379 }
```

Figure 2: Hiera Configuration Data

customization by enabling the configuration of default data with multiple levels of overrides.

Figure 1 is an example of a Puppet class for a `worker_queue` based on Redis [47]. Puppet classes can be reused in different scenarios without hard-coding parameters: in this particular example there is only one parameter, `port`. The concrete value of this parameter (`$exports["redis"]["port"]`) is derived from Hiera (Figure 2), which is shown as `6379` but can be computed automatically by a program at runtime. This allows us to calculate parameters based on the up-to-date system model, as opposed to hardcoding them. We use this technology in the compilation process described later.

We should also emphasize that while our current prototype uses Puppet, ANCOR can work with many mature CMT solutions such as Chef, SaltStack [49], Bcfg2 [43], or CFEngine [44]. Two important properties are required for a CMT to be useable by ANCOR. First, the directives an agent receives dictates a *desired state* as opposed to commands for state changes, which allows configuration changes to be handled in the same way as the initial configuration. Second, there is a mechanism for reusable configuration modules (*e.g.*, Puppet classes) that become the building blocks, or the "instruction set," into which ANCOR can compile the abstract requirement model. Depending on the specific CMT features, an orchestrator component might also be needed (especially in case the CMT employs only a pull-configuration model). An orchestrator component can be used on the CMT master node to trigger different actions on the CMT agents (achieve a push-configuration model).

3 The ANCOR Framework

Figure 3 shows the three major components of the ANCOR framework: the Operations Model, the Compiler, and the Conductor. The arrows denote information flow.

The key idea behind our approach is to abstract the functionality and structure of IT services into a model that is used to generate and manage concrete systems.

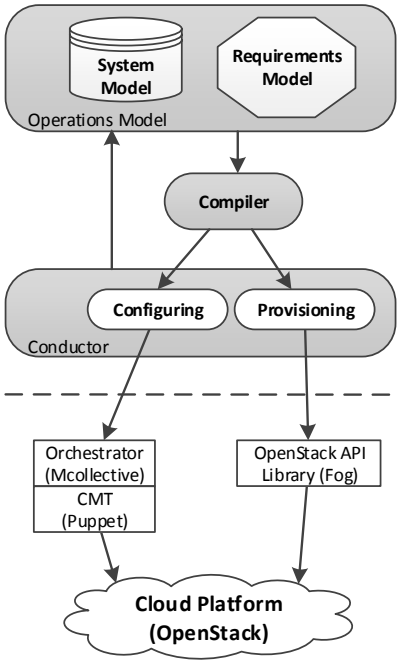

Figure 3: ANCOR Framework

We call this abstraction the *requirement model*. We also maintain the details of the concrete system in the *system model*. The two constitute the Operations Model. When ANCOR compiles a requirement model into a concrete, cloud-based system, the system model is populated with the details of the cloud instances and their correspondence to the requirement model. When the system changes, the system model is updated to ensure it has a consistent and accurate view of the deployment. Figure 14, part of the Appendix, shows the complete entity-relation diagram for the operations model.

The Compiler references the requirement model to make implementation decisions necessary to satisfy the abstract requirements and to instruct the conductor to orchestrate the provisioning and configuration of the instances. It can also instruct the conductor to perform user-requested configuration changes while ensuring the concrete system always satisfies the requirement model.

The Conductor consists of two sub-components, Provisioning and Configuring, which are responsible for interacting with the cloud-provider API, the CMT and orchestration tools (shown below the dashed line).

The ANCOR framework manages the relationships and dependencies between instances as well as instance clustering. Such management involves creating and deleting instances, adding/removing instances to/from clusters, and keeping dependent instances/clusters aware of configuration updates. The ANCOR framework simplifies network management as system dependencies are formalized and automatically maintained. Moreover, tra-

```
goals:
 ecommerce:
    name: eCommerce frontend
    roles:
      - weblb
      - webapp
      - worker
      - work_queue
      - db_master
      - db_slave

roles:
 weblb:
   name: Web application load balancer
   min: 2
   is_public: true
   implementations:
     default:
       profile: role::weblb::default
   exports:
     http:{ type: single_port, protocol: tcp, number:80 }
   imports:
     webapp: http

 webapp:
   name: Web application
   min: 3
   implementations:
     default:
       profile: role::webapp::default
   exports:
     http: { type: single_port, protocol: tcp }
   imports:
     db_master: querying
     db_slave: querying
     work_queue: redis

worker:
 name: Sidekiq worker application
 min: 2
 implementations:
   default:
     profile: role::worker::default
 imports:
   db_master: querying
   db_slave: querying
   work_queue: redis

work_queue:
 name: Redis work queue
 implementations:
   default:
     profile: role::work_queue::default
 exports:
   redis: { type: single_port, protocol: tcp }

db_master:
 name: MySQL master
 implementations:
   default:
     profile: role::db_master::default
 exports:
   querying: { type: single_port, protocol: tcp }

db_slave:
 name: MySQL slave
 implementations:
   default:
     profile: role::db_slave::default
 min: 2
 exports:
   querying: { type: single_port, protocol: tcp }
 imports:
   db_master: querying
```

Figure 4: eCommerce Website Requirements Specification

ditional failures can also be addressed, thus increasing network resiliency. Next, we explain the requirement model, the way it is compiled into a concrete system, and how it is used to better manage that system.

3.1 Requirement Model

3.1.1 The ARML language

We specify the requirement model in a domain-specific language called the ANCOR Requirement Modeling Language (ARML). ARML's concrete syntax is based on YAML [12], which is a language that supports specification of arbitrary key-value pairs. The *abstract syntax* of ARML is described in the Appendix (Figure 13). Figure 4 shows an example ARML specification for an eCommerce website in Figure 5.

The example is a scalable and highly available eCommerce website on a cloud infrastructure, which adopts a multiple-layer architecture with the various clusters of services shown in Figure 5: web load balancer (Varnish [55]), web application (Ruby on Rails [48]) with Unicorn [30], HAProxy [17] and Nginx [29]), database (MySQL [26]), worker application (Sidekiq [40]), and messaging queue (Redis [47]). Arrows indicate dependency between the clusters. Each cluster consists of multiple instances that offer the same services. Clustering supports scaling via cluster expansion (adding more instances to the cluster) or contraction (removing instances from the cluster). The clustering strategies employed by these applications fall into two main categories: homogeneous and master-slave. In a homogeneous cluster all

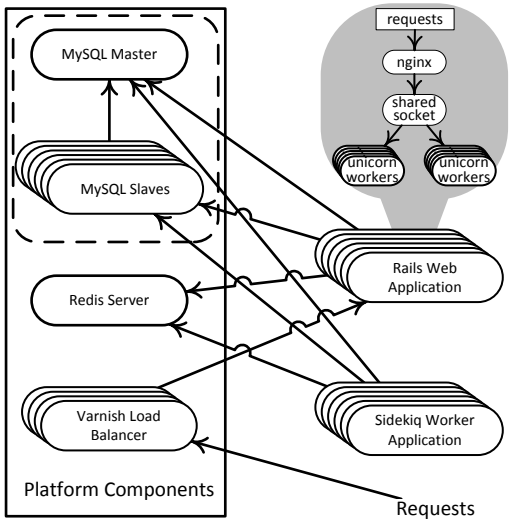

Figure 5: eCommerce Website

cluster members have the same configuration. If one of the instances stops working, another instance takes over. In master-slave, the master and slave instances have different configurations and perform different functions (*e.g.*, write versus read). If the master fails, a slave is promoted to be the master. In this example system, the web load balancer, web application, and the worker application employ the homogeneous clustering while the database employs master-slave (thus MySQL master and MySQL slaves form one cluster). Redis is used as a messaging queue. The clustering approach, mainly replication, supported by Redis is not suited for high-throughput queues.

A requirement model contains the specifications of system *goals* and *roles*. A *goal* is a high-level business goal (*e.g.*, blog website, eCommerce website, *etc.*) whose purpose is to organize the IT capabilities (*roles*) around business objectives. In Figure 4 there is a single system goal `ecommerce` that is supported by six roles.

A *role* defines a logical unit of configuration. Examples include a database role, a web application role, a message broker role, and so on. In essence, a role represents a group of similarly configured instances that provide the same functionality. In our model we use a single role to represent all the instances that achieve that functionality. For example, the web application instances in Figure 5 are configured identically (except for network addresses) and multiple load balancers dispatch incoming web requests to the instances in the web application cluster. We have a single role `webapp` for all the web application instances, and a `weblb` role for all the load balancer instances. The role-to-instance mapping is maintained in the system model.

A role may depend on other roles. A role uses a *channel* to interact with other roles. A channel is an interface *exported* (provided) by a role and possibly *imported* (consumed) by other roles. Channels could include a single network port or a range of ports. For instance, the `webapp` role exports an `http` channel, which is a TCP port (*e.g.,* 80). The `weblb` role imports the `http` channel from the `webapp` role. A role is a "black box" to other roles, and only the exported channels are visible interfaces. Using these interfaces the requirement model captures the dependencies between the roles.

The `webapp` role also imports three channels from various other roles: `querying` from `db_master`, `querying` from `db_slave`, and `redis` from `work_queue`. This means the `webapp` role depends upon three other roles: `db_master`, `db_slave`, and `work_queue`. The `min` field indicates the minimum number of instances that should be deployed to play the role. If `min` is not specified it's default value is 1. In the current prototype implementation the count of the instances will be equal to `min`. The requirement model addresses instance clustering naturally by requiring multiple instances to play a role. For homogeneous clusters this is easy to understand. For master-slave clusters, at least two roles are involved in the cluster, the master and the slave. The dependency information captured in the export/import relationship is sufficient to support calculating configuration changes when, for example, the master is removed from the cluster and a new node is promoted to be the master. So far we have not found any real-world clustering strategies that require explicitly modeling the cluster structure beyond the dependency relationship between the roles that form the cluster. If more general clustering strategies are needed, we will extend our requirement model to support this.

3.1.2 Role Implementation

Role names are system-specific and are chosen by the user or system engineers to convey a notion of the role's purpose in the system; there are no pre-defined role names in ARML. However, to automatically compile and maintain concrete systems, system engineers must provide the *semantics* of each role, which is specified in the role specification's *implementation* field. The implementation field defines how each instance must be configured to play the role. The conductor then uses this implementation information to properly configure and deploy the concrete instances. The value of the implementation field is thus dependent on the CMT being used. This process is similar to traditional programming language compilers where abstract code constructs are compiled down to machine code. The compiler must contain the semantics of each code construct in terms of machine instructions for a specific architecture. The analogy between our ANCOR compiler and a programming language compiler naturally begs the question: "what is the *architecture-equivalent* of a cloud-based IT system?" In other words, is there an interface to a "cloud runtime" into which we can compile an abstract specification?

It turns out that a well-defined interface between the requirement model and the "cloud runtime" is well within reach if we leverage existing CMT technologies. As explained in Section 2.2, there has been a general movement in CMT towards encapsulating commonly-used configuration directives into reusable, parameterized modules. Thus, one can use both community and custom modules to implement roles and populate those reusable knowledge units with parameters derived from our high-level requirement model. Potential role implementations must be specified in a role's "implementations" field (see Figure 4). A role may have multiple implementations since there could be more than one way to achieve its functionality. The compiler then selects an

```
class role::weblb::default {

  $exports = hiera("exports")
  $imports = hiera("imports")
---
  class { "profile::varnish":
    listen_port => $exports["http"]["port"] }

  $backends = $imports["webapp"]

  file { "default.vcl":
    ensure  => file,
    content =>
     template("role/weblb-varnish/default.vcl.erb"),
    path    => "/etc/varnish/default.vcl",
    owner   => root,
    group   => root,
    mode    => 644,
    require => Package["varnish"],
    notify  => Exec["reload-varnish"], }
}
```

Figure 6: Web Load Balancer Role Implementation

appropriate role implementation from those that satisfy all constraints levied by existing role implementations in the system. (The current prototype simply selects the first role implementation in the "implementations" field; we leave the constraint specification and implementation selection problems for future work.)

An important challenge was structuring the knowledge units so that they could be easily reused in different requirement models. Failing to have a proper role implementation design model would lead to rewriting every single role implementation from scratch. We adopted an approach similar to that used by Dunn [11]. We name role implementations based on their functionality and/or properties (*e.g.*, weblb) and use "profiles" to integrate individual components to embody a logical software stack. The software stack is constructed using community and custom modules as lower-level components[3].

For instance, in case of the load balancer: The `weblb` role is assigned the `default` implementation in the ARML specification (see Figure 4). Figure 6 is a Puppet class that shows the implementation that was defined as `default` for the `weblb`. Figure 7 pictures a sample of possible parameters that Puppet is getting through Hiera from the compiler for configuring one of the `weblb` instances. There are two parts in each role implementation (see Figure 6). The code before "`---`" imports operations model values from Hiera (*e.g.*, see Figure 7). The statements `hiera("exports")` and `hiera("imports")` query Hiera to find all the channels the web balancer will consume (*imports*) and the channels that it will make available to other roles (*exports*). These channels will be stored in two variables,

[3] All `default` role `implementations` used with ANCOR are available on GitHub: https://github.com/arguslab/ancor-puppet

```
{
 "exports": {
   "http": {
     "port": 80,
     "protocol": "tcp"
   }
 },
 "imports": {
   "webapp": {
     "webapp-ce66a264": {
       "ip_address": "10.118.117.16",
       "stage": "undefined",
       "planned_stage": "deploy",
       "http": {
         "port": 42683,
         "protocol": "tcp"
       }
     },
     "webapp-84407edd": {
       "ip_address": "10.118.117.19",
       "stage": "undefined",
       "planned_stage": "deploy",
       "http": {
         "port": 23311,
         "protocol": "tcp"
       }
     },
     "webapp-1ce1ce46": {
       "ip_address": "10.118.117.22",
       "stage": "undefined",
       "planned_stage": "deploy",
       "http": {
         "port": 10894,
         "protocol": "tcp"
       }
     }
   }
 },
 "classes": [
   "role::weblb::default"
 ]
}
```

Figure 7: Specific `weblb` Paramaters Sample Exposed to Hiera by ANCOR

`"exports"` and `"imports"`. The web balancer will be instructed to expose an `http` channel on the particular port (in this case port 80, see "exports" in Figure 7), and will be configured to use all instances that are assigned to play the role `webapp`, from which it imports the `http` channel.

The default `weblb` implementation is based on the reusable Puppet "Varnish profile" (`profile::varnish` - see Figure 8). The `profile::varnish` Puppet class uses the necessary specified parameters to customize the Varnish installation. The parameters in `profile::varnish` (*e.g.*, `$listen_address`, `$listen_port`, *etc.*) are initialized with default values. These values will be overwritten in case they are specified in `role::weblb::default`. In the current example, `$listen_port` is the only parameter that will be overwritten (see Figure 6), the other parameters will keep their default values defined in

```
class profile::varnish(
  $listen_address = "0.0.0.0",

# $listen_port's default value "6081" will be
# OVERWRITTEN with the value passed
# from role::weblb::default
  $listen_port = 6081,

  $admin_listen_address = "127.0.0.1",
  $admin_listen_port = 6082) {

  apt::source { "varnish":
    location =>
      "http://repo.varnish-cache.org/ubuntu/",
    release => "precise",
    repos => "varnish-3.0",
    key => "C4DEFFEB",
    key_source =>
    "http://repo.varnish-cache.org/debian/GPG-key.txt",
  }
  package { "varnish":
    ensure => installed,
    require => Apt::Source["varnish"], }

  service { "varnish":
    ensure => running,
    require => Package["varnish"], }

  Exec {
   path => ["/bin", "/sbin", "/usr/bin", "/usr/sbin"]
  }

  exec { "reload-varnish":
    command => "service varnish reload",
    refreshonly => true,
    require => Package["varnish"] }

  file { "/etc/default/varnish":
    ensure => file,
    content =>
      template("profile/varnish/default.erb"),
    owner => root,
    group => root,
    mode => 644,
    notify  => Service["varnish"],
    require => Package["varnish"], }
}
```

Figure 8: Web Load Balancer Role Profile

```
# Configuration file for varnish
START=yes
NFILES=131072
MEMLOCK=82000
VARNISH_VCL_CONF=/etc/varnish/default.vcl
VARNISH_LISTEN_ADDRESS=<%= @listen_address %>
VARNISH_LISTEN_PORT=<%= @listen_port %>
VARNISH_ADMIN_LISTEN_ADDR=<%= @admin_listen_address %>
VARNISH_ADMIN_LISTEN_PORT=<%= @admin_listen_port %>
VARNISH_MIN_THREADS=1
VARNISH_MAX_THREADS=1000
. . .
```

Figure 9: Web Load Balancer, Varnish, Initialization Script: *default.erb* (used in *profile::varnish*)

```
<% @backends.each do |name, backend| %>
backend be_<%= name.sub("-", "_") %> {
  .host = "<%= backend["ip_address"] %>";
  .port = "<%= backend["http"]["port"] %>";

  .probe = {
    .url = "/";
    .interval = 5s;
    .timeout = 1s;
    .window = 5;
    .threshold = 3;
  }
}
<% end %>

director webapp round-robin {
  <% @backends.each_key do |name| %>
  {
    .backend = be_<%= name.sub("-", "_") %>;
  }
  <% end %>
}

sub vcl_recv {
  set req.backend = webapp;
}
```

Figure 10: Web Load Balancer, Varnish, Configuration File: *default.vcl.erb* (used in *role::weblb::default*)

`profile::varnish`. The parameters values (initialized in `role::weblb::default` or in `profile::varnish`) are passed to Figure 9 and Figure 10 to generate the customized Varnish configuration files, and this is all done by Puppet automatically at runtime.

Thus, a role implementation definition specifies *a concrete way to implement the intended functionality embodied by a role* by describing the invocation of predefined configuration modules with concrete parameters computed from the operations model. These role implementations are not only useful when generating the system, but also for modifying the system as it changes over time. For example, if a new instance is deployed to play the `webapp` role, the dependency structure in the operations model allows ANCOR to automatically find all the other roles that may be impacted (those depending on the `webapp` role) and use their role implementation to direct the CMT to reconfigure them so that they are consistent with the updated operations model.

ANCOR leverages existing CMT to define the role implementations, to minimize additional work that has to be done by the system engineers. For example, only information in Figure 6 is what one needs to write for ANCOR; Figure 7 is generated automatically by ANCOR; Figure 8, 9, and 10 are what one would have to specify anyway using Puppet.

3.2 The ANCOR Workflow

There are four main phases involved in creating and managing cloud-based systems using ANCOR.

1. Requirements model specification
2. Compilation choices specification
3. Compilation/Deployment
4. Maintenance

The first two phases result in the creation of the requirement model while the next two phases perform the actual deployment and maintenance of the cloud-based system.

3.2.1 Requirement Model Specification

In this phase, the user and system engineers work together to define the goals of the system, which may require significant input from various stakeholders. Next, they determine the roles required to achieve each goal and the dependencies among the roles. This task could be handled by the system engineers alone or in consultation with the user. The high-level requirement language ARML provides an abstract, common language for this communication.

3.2.2 Compilation Choices Specification

In this phase, system engineers define role semantics using pre-defined CMT modules. In our current prototype this is accomplished by defining the role implementations that invoke Puppet classes as described in Section 3.1.2. If no appropriate CMT modules exist, system engineers must define new profiles and place them into the repository for future use. In general, system engineers could specify multiple implementation choices for each role to provide the ANCOR compiler flexibility in choosing the appropriate implementation at runtime. One of the options available to system engineers is the specification of the desired operating system for each instance. Here again, different operating systems may be used for each implementation of a role. With a wide variety of choices available to systems engineers, a constraint language is needed to specify the compatibility among the various implementation choices; we leave this for future research.

3.2.3 Compilation/Deployment

Once the requirement model has been defined, the framework can automatically compile the requirements into a working system on the cloud provider's infrastructure. This process has seven key steps:

1. The framework signals the compiler to deploy a specific requirement model.
2. The compiler makes several implementation decisions including the number of instances used for each role and the flavors, operating systems, and role implementations used.
3. The compiler signals the conductor component to begin deployment.
4. The conductor interacts with the OpenStack API to provision instances and create the necessary security rules (configure the cloud's internal firewall). The provisioning module uses a package such as *cloud-init* to initialize each cloud instance, including installing the CMT and orchestration tool agents (*e.g.*, the Puppet agent and MCollective [19] agent).
5. Once an instance is live, the message orchestrator (*e.g.*, MCollective) prepares the instance for configuration by distributing its authentication key to the CMT master (*e.g.*, Puppet master).
6. The configuration is pushed to the authenticated instances using the CMT agent and, if needed, the orchestrator (*e.g.*, Puppet agent and MCollective).
7. System engineers may check deployed services by using system monitoring applications such as Sensu [50] or Opsview [46] , or by directly accessing the instances.

In step 6, configuration is carried out via the Hiera component while configuration directives (node manifests) are computed on the fly using ANCOR's operations model. This ensures that the parameters used to instantiate the Puppet modules always reflect the up-to-date system dependency information.

3.2.4 Maintenance

Once the system is deployed in the cloud, system engineers can modify the system. If the change does not affect the high-level requirement model, the maintenance is straightforward. The compiler will track the impacted instances using the operations model and reconfigure them using the up-to-date system information. A good example for this type of change is cluster expansion/contraction.

Cluster expansion is used to increase the number of instances in a cluster (*e.g.*, to serve more requests or for high-availability purposes).

1. System engineers instruct the compiler to add instances to a specific role. In future work we also would like to allow monitoring modules to make expansion decisions as well.
2. The compiler triggers the conductor component to create new instances, which automatically updates the ANCOR system model.

3. The compiler calculates the instances that depend on the role and instructs the configuration manager to re-configure the dependent instances based on the up-to-date ANCOR system model.

Cluster contraction is the opposite of cluster expansion. The main goal of cluster contraction is to reduce the number of instances in a cluster (*e.g.*, to lower cost).

1. System engineers instruct the compiler to mark a portion of a role's instances for removal.
2. The compiler calculates the instances that depend on the role and instructs the configuration manager to re-configure the dependent instances based on the up-to-date ANCOR system model.
3. The compiler triggers the conductor component to remove the marked instances.

If the change involves major modifications in the requirement model (*e.g.,* adding/removing a role), ANCOR will need to re-compile the requirement model. How to perform "incremental recompilation" that involve major structural changes without undue disruption will be a topic for future research.

4 Prototype Implementation

We built a prototype (Figure 3) in Ruby (using Rails, Sidekiq and Redis) to implement the ANCOR framework using OpenStack as the target cloud platform. The operations model is stored in MongoDB collections using Rails. ANCOR employs straight-forward type-checking to ensure that the requirement model is well-formed (*e.g.*, allowing a role to import a channel from another role only if the channel is exported by that role). The compiler references the MongoDB document collections that store the operations model and interacts with the conductor using a Redis messaging queue and Sidekiq, a worker application used for background processing. The conductor interacts with the OpenStack API through Fog (a cloud services library for Ruby) to provision the network, subnets and instances indicated by the compiler. Once an instance is live, the configuration module uses Puppet and MCollective to configure it using the manifest computed on the fly based on the operations model. The conductor also interacts with the system model and updates the provided system model database every time it performs a task (provisioning or configuration). Therefore, the system model stored in the MongoDB datastore will always have an updated picture of the system. Obviously, the different role implementation choices (*e.g.,* Sidekiq, Redis or Rails) used to build the eCommerce website example scenario (Figure 5) are independent from the components that leverage Sidekiq, Redis and Ruby on Rails in the ANCOR framework prototype (Figure 3)

In the current implementation we are using a workflow model that is based on chained and parallel tasks processing. Once the ARML specification is entered by the user, the specification will be parsed and the requirement model will be encountered. Next, the compiler steps in and based on the requirement model it chooses the number of instances that play a role, the role implementations, the IP addresses, the channels (port number and/or sockets that will be consumed or exposed), *etc.* Then the compiler populates the system model and creates various tasks that it passes to the worker queue. A task can be viewed as an assignment that is passed to a background (worker) process. In ANCOR, Sidekiq is used for background processing. Tasks are stored in the database and have several attributes (*e.g.*, type, arguments, state, context) A task can be related to provisioning (*e.g.*, using Fog) or to configuring an instance (*e.g.*, push configuration from Puppet master to Puppet agent). In case other tasks (*e.g.*, deploy instance) depend on the execution of the current task (*e.g.*, create network) a *wait handle* is created. Wait handles can be viewed as the mechanism used by the tasks to signal dependent tasks when they finished execution. A task creates a wait handle object that stores the ids of the tasks that wait for it to execute. Once the task finished the wait handle triggers all the dependent tasks to execute. The purpose of a wait handle is to start, resume or suspend the dependent tasks. Using this approach we can resume or suspend a task several times including tasks related to the orchestration tool (MCollective) and the CMT (Puppet). Independent tasks (*e.g.*, two different *deploy instance* tasks) will be executed in parallel employing locks on certain shared resources. The ANCOR prototype code, detailed instructions on how to deploy/run it and a detailed document containing specific implementation details are available online at https://github.com/arguslab/ancor.

4.1 Using ANCOR

The current framework implementation exposes a REST API [14] to it's clients. The current clients include a Command-Line Interface (CLI), a web-browser dashboard and also the Puppet master (specifically the Hiera module). Through the REST API, Hiera is obtaining the specific configuration details (*e.g.*, imported and exported channels - `$exports` and `$imports` arrays, see Figure 1) from the compiler in order to customize the Puppet modules that are part of the chosen role implementation (*e.g.*, see Figure 2). The CLI and the dashboard are used to deploy, manage, visualize (in case of the dashboard) and delete ANCOR deployments.

One can use the CLI to deploy, manage and delete the eCommerce website example using several key commands:

1. `ancor environment plan eCommerce.yaml` – plans the deployment (`eCommerce.yaml` is shown in Figure 4)
2. `ancor role list` – lists the current set of roles
3. `ancor instance list` – lists the current set of instances
4. `ancor environment commit` – deploys the environment on the cloud infrastructure
5. `ancor task list` – displays the current progress of the deployment
6. `ancor instance add webapp` – used to add another instance for the `webapp` role after all `tasks` are `completed`
7. `ancor environment list` – used to unlock the environment after all tasks are `completed`
8. `ancor environment remove production` – deletes the current deployment

More options and instructions on using the ANCOR CLI and the dashboard are available on the framework website.

4.2 Prototype Evaluation

The objective of our evaluation is to measure the dynamics of change management. This implies measuring how much the performance of a deployed IT system is influenced when adding and removing instances using ANCOR. Throughput and latency values are highly dependent on the applications' configurations and on the underlying cloud infrastructure. Thus, we are not focused on the throughput and latency themselves but on the *difference* between the baseline measurements and the measurements during system changes. We evaluated our prototype using two IT system setups: a basic blogging website (Figure 11) and the eCommerce website scenario (Figure 5). These scenarios are also available online. As we add other scenarios into the repository and integrate ANCOR with different cloud infrastructures (AWS will likely be the next infrastructure supported), we plan to expand this evaluation.

The current testing scenarios were deployed on a private cloud testbed in the Argus CyberSecurity Laboratory at Kansas State University. The cloud testbed consists of fifteen Dell PowerEdge R620 servers, each of which is equipped with 2 x Intel Xeon E-2660 v2 Processor@2.20GHz, 128GB of RAM, 4 x 1TB 7.2K HDD running in RAID0 (Stripe) and an Intel dual port 10GbE DA/SFP+ network card. Moreover, each server is connected to the main switch (Dell Force10 - S4810P switch) using two bounded 10GbE ports.

We deployed an OpenStack Havana infrastructure on these machines using Mirantis' open-source framework Fuel [24]. The infrastructure consists of one controller node and fourteen compute nodes.

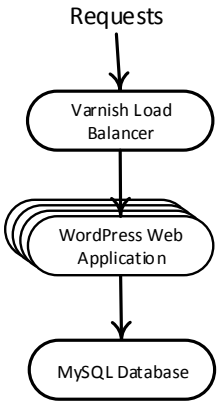

Figure 11: Blogging Website

4 threads and **40** connections				
	Avg	Stdev	Max	+/- Stdev
Latency	1.03ms	1.28ms	305.05ms	99.73%
5999.68 requests/sec, 1800296 requests in 5 min				
Timeouts: 5052				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 1: Benchmarking - Blogging Website Scenario (baseline) with Caching

wrk [58], an HTTP benchmarking tool, was used to test the system's availability and performance while managing various clusters (*i.e.,* adding/removing instances to/from a cluster). We ran the benchmarking tool on the initially deployed scenarios and established a baseline for every component in our measurements. *wrk* was launched from client machines that are able to access the websites (*i.e.,* connect to the load balancer). We targeted various links that ran operations we implemented to specifically test the various system components (*e.g.*, read/write from/to the database). After establishing the baseline, we started adding and removing instances to and from different clusters while targeting operations that involve the deepest cluster in the stack (database slave for the eCommerce setup). In case of the blogging website scenario we focused on the web application. The performance in the three cases are very close since we only added and removed one instance in the experiments. All caching features at the application level were disabled in both scenarios. Having caching features enabled would have prevented us from consistently reaching the targeted components; however the performance would be *greatly* improved. For instance, Table 1 exposes the baseline results for accessing a WordPress [56] posts from the blogging website setup (Figure 12) with caching enabled.

```
goals:
 wordpress:
   name: Wordpress blog
   roles:
     - db
     - webapp
     - weblb

roles:
 weblb:
   name: Web application load balance
   is_public: true
   implementations:
     default:
       profile: role::weblb::default
   exports:
     http: { type: single_port, protocol: tcp, number: 80 }
   imports:
     webapp: http

 webapp:
   name: Web application
   min: 2
   implementations:
     default:
       profile: role::weblb::default
   exports:
     http: { type: single_port, protocol: tcp }
   imports:
     db: querying

 db:
   name: MySQL database
   implementations:
     default:
       profile: role::weblb::default
   exports:
     querying: { type: single_port, protocol: tcp }
```

Figure 12: Blogging Website Requirements Model

4 threads and **40** connections				
	Avg	Stdev	Max	+/- Stdev
Latency	458.89ms	292.65ms	1.30s	73.86%
86.18 requests/sec, 25855 requests in 5 min				
Timeouts: 0				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 2: Benchmarking - Blogging Website Scenario - `webapp` cluster (baseline)

4 threads and **40** connections				
	Avg	Stdev	Max	+/- Stdev
Latency	418.76ms	268.63ms	1.88s	78.65%
96.63 requests/sec, 28989 requests in 5 min				
Timeouts: 0				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 3: Benchmarking - Blogging Website Scenario - `webapp` cluster (adding one instance)

4 threads and **40** connections				
	Avg	Stdev	Max	+/- Stdev
Latency	456.13ms	325.94ms	1.34s	73.53%
89.49 requests/sec, 26849 requests in 5 min				
Timeouts: 0				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 4: Benchmarking - Blogging Website Scenario - `webapp` cluster (removing an instance)

4 threads and **40** connections				
	Avg	Stdev	Max	+/- Stdev
Latency	16.75ms	18.92ms	333.76ms	91.12%
601.21 requests/sec, 180412 requests in 5 min				
Timeouts: 4728				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 5: Benchmarking - `db_slave` cluster (baseline)

4 threads and **40** connections				
	Avg	Stdev	Max	+/-Stdev
Latency	17.21ms	20.22ms	161.34ms	93.13%
771.71 requests/sec, 231563 requests in 5 min				
Timeouts: 4292				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 6: Benchmarking - `db_slave` cluster (adding one instance)

4.2.1 Blogging Website Scenario

The basic blogging website is pictured in Figure 11. When adding instances to the WordPress web application cluster, latency and throughput improve (Table 3), which is as expected. When instances are removed from the cluster the performance is slightly worse than the add-instance case, due to the decreased number of instances in the cluster. But it is still slightly better than the baseline, which has the less number of instances.

4.2.2 eCommerce Website Scenario

When adding instances to the MySQL slaves cluster, latency slightly increases but throughput is improved (Table 6). When instances are removed from the cluster both performance metrics were negatively affected (Table 7).

These results show that 1) the ANCOR system is reliable in generating a concrete IT system in the cloud from a high-level requirement model specification; and 2) the formal models in ANCOR help facilitate orchestrating changes in the cloud-based IT system such as scaling.

It is important not to disregard the fact that all benchmarking results were influenced by the way applications were configured (*e.g.*, disabled caching functionality and "hot-swap" feature). The "hot-swap" feature loads an updated configuration without restarting a service (*e.g.*, Unicorn), which could further improve performance if one further tunes these applications.

4 threads and **40** connections				
	Avg	Stdev	Max	+/-Stdev
Latency	15.91ms	23.18ms	417.68s	93.95%
500.59 requests/sec, 150202 requests in 5 min				
Timeouts: 4903				
Errors (non-2xx or 3xx HTTP responses): 0				

Table 7: Benchmarking - `db_slave` cluster (removing one instance)

5 Discussion and Future Work

Our requirements specification approach and the implemented ANCOR framework offer system engineers the same flexibility as in a typical IaaS model. This means that engineers can keep their workflow using their preferred configuration management tools (*e.g.,* Puppet and Chef) and orchestration tools (*e.g.,* MCollective). They have the option to do everything in their preferred ways up to the point where they connect the components (services) together. For example, system engineers have the option of using predefined configuration modules and of leveraging the contributions from the CMT community. Or they can write their own manifests or class definitions to customize the system in their own ways. ANCOR can leverage all of these and does not force the system engineers to use particular low-level tools or languages; rather it provides the ability to manage the whole system based on a high-level abstraction.

The high-level requirement model we developed could also facilitate tasks like failure diagnosis and system analysis to identify design weaknesses such as single point of failures or performance bottlenecks. The system dependency information specified in the requirement model and maintained in the operations model allows for more reliable and streamlined system updates such as service patching. It also allows for more fine-grained firewall setup (*i.e.*, only allows network access that is consistent with the system dependency), and enables porting systems to different cloud providers in a more organized manner (*e.g.*, one can take the up-to-date requirement model and compile it to a different cloud provider's infrastructure, and then synchronize data from the old one to the new one).

One of our future work directions is to construct a proactive change mechanism as part of the compiler. This mechanism would randomly select specific aspects of the configuration to change (*e.g.*, replacing a portion of the instances with freshly created ones with different IP addresses, application ports, software versions, *etc.*); the changes would be automatically carried out by the conductor component. Moreover, changes can be scheduled based on security monitoring tools (*e.g.*, Snort [51], Suricata [52], SnIPS [59], *etc.*) or on component behavior and interaction reports (using an approach similar to [1]). Our goal in this is to analyze the possible security benefits as well as to measure and analyze the performance, robustness, and resilience of the system under such disturbance. We are also considering adding SLA (Service Level Agreement) related primitives to the requirement model. Furthermore, we are looking into adding other CMT modules to support the use of multiple configuration management tools such as Chef. In addition, we are planning on developing built-in Docker support modules and switching to the OpenStack AWS compatible API that will enable us to use the same provider module on different cloud platforms.

6 Related Work

There has been a general trend towards creating more abstractions at various levels of cloud service offerings. Some of the solutions even use similar terminologies and features as those in ANCOR. For example, some solutions also use the term "role" in a similar way to ours [11, 57], and others have adopted named channels to describe dependencies in configuration files [22, 23, 57]. Thus it is important to describe the fundamental differences between the abstraction used in ANCOR and those in the other solutions, so that one does not get confused by the superficial similarities between them.

Abstractions used in the various PaaS solutions such as the Windows Azure service definition schema [57] and Google AppEngine YAML-based specifications [16] allow users to define their cloud-based applications. These abstractions, while useful for helping users to use the specific PaaS platform more easily, do not serve the same purpose as ARML. In particular, they only define user-provided applications and not the complete IT system in the cloud, since the important platform components are not modeled. Thus these abstractions cannot be used to compile into different implementation choices or platforms. They are tied to the specific PaaS platform and thus will never separate the user requirements from the platform details. Using these abstractions will lock the users in to the specific PaaS vendor, while ANCOR will give users complete flexibility as to implementation choices at all levels, including platforms.

Docker container-based solutions such as Maestro [22], Maestro-NG [23], Deis [9], and Flynn [15] provide management aid for deploying cloud instances using the Linux Containers virtualization approach. Some of them (Maestro and Maestro-NG) also have environment descriptions (in YAML) for the Docker instances that include named channels to capture dependencies. These solutions can automate initial deployment of cloud instances and make it easier to stand up a PaaS, but again they take a different approach and do not provide the same level of abstraction that supports the vision outlined in Section 1. Specifically, the abstractions provided by their environment descriptions are focused on instances as opposed to the complete IT system, the container is the unit of configuration, and maintenance tasks are done by progressing through containers. It is not clear how much the container-based solutions can help alleviate the long-term maintenance problem of cloud-based IT systems. Moreover, the container-based solutions are tied to Linux environments and Docker, which is still under heavy development and not ready for production use. We also summarized a few other features to differentiate ANCOR

from the other solutions in Table 8. ANCOR can be used to deploy systems using other orchestration tools such as Flynn and Deis in conjunction with traditional IT systems. As shown in Table 8, ANCOR is currently the most general and flexible management solution available.

Several companies are developing cloud migration technologies. While some appear to internally use abstractions to support migration [8, 13], no details are available for independent evaluation. Our approach is more fundamental in the sense that we *build* systems using the abstraction and, smoother and more reliable migration could be a future product of our approach. Rather than creating technology specifically to replicate existing systems, we aim to fundamentally change the way cloud-based systems are built and managed, which includes enabling dynamic and adaptive system changes, reducing human errors, and supporting more holistic security control and analysis.

Often, solutions like AWS CloudFormation [4], OpenStack Heat [31] or Terraform [53] may be, mistakenly, viewed as being at the same level of abstraction with ANCOR. These solutions are primarily focused on building and managing the infrastructure (cloud resources) by allowing the details of an infrastructure to be captured into a configuration file. CloudFormation and Heat manage AWS/OpenStack resources using templates (*e.g.*, Wordpress template [33], MySQL template [32], *etc.*), and they do not separate user requirements from system implementation details. The templates have the potential to integrate well with configuration management tools but there is no model of the structure and dependencies of the system. Thus, it cannot achieve one main objective of ANCOR which is to use the operations model to maintain the system, *e.g.*, updating dependencies automatically while replacing instances. Terraform similarly uses configuration files to describe the infrastructure setup, but it goes even further by being cloud-agnostic and by enabling multiple providers and services to be combined and composed [54].

Juju [41] is a system for managing services and works at a similar level as ANCOR. It resides above the CMT technologies and has a way of capturing the dependencies between software applications (services). It can also interact with a wide choice of cloud services or bare metal servers. The Juju client works on multiple operating systems (Ubuntu, OS X, and Windows) but Juju-managed services run primarily on Ubuntu servers, although support for CentOS and a number of Windows-based systems will be available in the near future [42]. While we were aware of the existence of Juju at the time of this paper's writing, the lack of formal documentation on how Juju actually works, the services running only on Ubuntu, and the major changes in the Juju project (*e.g.*, code base was completely rewritten in Go) kept us away from this project. We recently reevaluated Juju and discovered fundamental similarities between ANCOR and Juju. Even so, there are subtle differences that make the two approaches work better in different environments. For instance, the ANCOR approach adopts a more "centralized" management scheme in terms of deciding the configuration parameters of dependent services, while Juju adopts a negotiation scheme between dependent services (called relations in Juju) to reach a consistent configuration state across those services. Depending on the need for change in the system, the ANCOR approach may be more advantageous when it comes to a highly dynamic system with proactive changing (*e.g.*, in a moving target defense system).

Our approach has benefited from the recent development in the CMT technologies that have provided the building blocks (or "instruction sets") for our compiler. The general good practice in defining reusable configuration modules such as those advocated by Dunn [11] is aligned very well with the way we structure the requirement model. Thus our approach can be easily integrated with those CMT technologies.

Sapuntzakis *et al.* [37] proposed the configuration language CVL and the Collective system to support the creation, publication, execution, and update of virtual appliances. CVL allows for defining a network with multiple appliances and passing configuration parameters to each appliance instance through key-value pairs. The decade since the paper was published has seen dramatic improvement in configuration management tools such as Puppet [20] and Chef [34], which has taken care of specifying/managing the configuration of individual machines. Our work leverages these mature CMTs and uses an abstraction on a higher level. In particular, ARML specifies the dependency among roles through explicit "import" and "export" statements with the channel parameters, which are translated automatically to concrete protocol and port numbers by the integration of operations model and the CMT. While CVL does specify dependency among appliances through the "provides" and "requires" variables, they are string identifiers and not tied to configuration variables of the relevant appliances (*e.g.*, the "DNS host" configuration parameter of an LDAP server). In the CVL specification of the virtual appliance network, the programmer would need to take care in passing the correct configuration parameters (consistent with the dependency) to the relevant appliances. In ANCOR this is done automatically by the coordination between the CMT and the operations model (compiled from the high-level ARML specification). This also allows for easy adaptation of the system (*e.g.*, cluster expansion and contraction).

Begnum [6] proposed MLN (Manage Large Networks) that uses a light-weight language to describe a

Offering	Focus	Platform	Multiple Cloud Infrastructures	CMTs or other similar structures
OpenShift	Private PaaS	RHEL	Yes	None
Flynn	Private PaaS	Linux	Yes	Heroku Buildpacks, Docker
Deis	Private PaaS	Linux	Yes	Heroku Buildpacks, Chef, Docker
OpsWorks AWS	General	Ubuntu	No	Chef
Maestro	Single-host	Linux	Yes	Docker (based on LXC)
Maestro-NG	General	Linux	Yes	Docker (based on LXC)
Google AppEngine	PaaS	Linux	No	None
Heroku	PaaS	Linux	No	Heroku Buildpacks
Windows Azure	PaaS	Windows	No	None
Ubuntu Juju	General	Ubuntu, CentOS, Windows	Yes	Charms
ANCOR	General	Any	Yes	Puppet

Table 8: Current Solutions Comparison

virtual machine network. Like ANCOR, MLN uses off-the-shelf configuration management solutions instead of reinventing the wheel. A major difference between ANCOR and MLN is that ANCOR captures the instance dependency in the requirement model, which facilitates automating configuration of a complete IT system and its dynamic adaptation. ANCOR achieves this by compiling the abstract specification to the operations model, which is integrated with the CMT used to deploy/manage the instances.

Plush [2] is an application management infrastructure that provides a set of abstractions for specifying, deploying, and monitoring distributed applications (*e.g.,* peer-to-peer services, web search engines, social sites, *etc.*). While Plush's architecture is flexible, it is not targeted towards cloud-based enterprise systems and it is unclear whether system dependencies can be specified and maintained throughout the system lifecycle.

Narain pioneered the idea of using high-level specifications for network infrastructure configuration management in the ConfigAssure [27, 28] project. ConfigAssure takes formally specified network configuration constraints and automatically finds acceptable concrete configuration parameter values using a Boolean satisfiability (SAT) solver. While we adopt Narain's philosophy of using formal models to facilitate system management, the cloud problem domain is different from that of network infrastructure, thus requiring new models and methodologies.

Use of higher-level abstractions to improve system management has also been investigated in the context of Software-Defined Networking (SDN). Monsanto *et al.* introduced abstractions for building applications from independent modules that jointly manage network traffic [25]. Their Pyretic language and system supports specification of abstract network policies, policy composition, and execution on abstract network topologies. Our ANCOR language and system adopts a similar philosophy for cloud-based deployment and management.

7 Conclusion

Separating user requirements from the implementation details has the potential of changing the way IT systems are deployed and managed in the cloud. To capture user requirements, we developed a high-level abstraction called the requirement model for defining cloud-based IT systems. Once users define their desired system in the specification, it is automatically compiled into a concrete cloud-based system that meets the specified user requirements. We demonstrate the practicality of this approach in the ANCOR framework. Preliminary benchmarking results show that ANCOR can improve manageability and maintainability of a cloud-based system and enable dynamic configuration changes of a deployed system with negligible performance overhead.

8 Acknowledgments

This work was supported by the Air Force Office of Scientific Research award FA9550-12-1-0106 and U.S. National Science Foundation awards 1038366 and 1018703. Any opinions, findings and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the above agencies.

A Appendix

The *abstract syntax* of ARML is shown in Figure 13. We use $A \Rightarrow B$ to represent the key-value pairs (written as "$A : B$." in YAML). Upper-case identifiers represent non-terminal symbols, lower-case identifiers represent terminal symbols, and ARML keywords are distinguished by bold font.

Figure 14 shows the complete entity-relation diagram for the operations model. The arrows indicate the direction of references in the implementation (one-way or two-way references) and the associated multiplicity (1-to-1, 1-to-n, or n-to-n). For instance, one role may support multiple goals, and multiple roles could support one goal. Thus the multiplicity between goal and role is n-to-n.

ReqModel ::= **goals** ⇒ *GoalSpec*$^+$
roles ⇒ *RoleSpec*$^+$

GoalSpec ::= goalID ⇒
[**name** ⇒ string]
roles ⇒ roleID$^+$

RoleSpec ::= roleID ⇒
[**name** ⇒ string]
[**min** ⇒ integer]
[**exports** ⇒ *ChannelSpec*$^+$]
[**imports** ⇒ *ImportSpec*$^+$]
implementations ⇒ *ImplementationSpec*$^+$

ChannelSpec ::= chanelID ⇒
(**type** ⇒ channelTypeID, *ChannelAttr**)

ChannelAttr ::= attributeID ⇒ value

ImportSpec ::= roleID ⇒ channelID$^+$

ImplementationSpec ::= implementationID ⇒ value

goalID, roleID, channelID, attributeID, channelTypeID, strategyID, implementationID, clusterID, tag are symbols. *integer* and *string* are defined in the usual way.

Figure 13: ARML Grammar

The system model is a local reflection of the cloud-based IT system and, as previously mentioned, it is used for bookkeeping. This enables the user to track instances in terms of roles. Furthermore, the system model bridges the gap between the more abstract requirement model and the many different concrete systems that can implement it. The requirement model is considered read-only by the rest of the framework. On the other hand, the system model can be updated by every component in the framework.

An *instance* is a virtual machine that is assigned to play a role. A role can be played by more than one instance but an instance currently can play only one role. A role can have one or more ways of being implemented. This aspect is captured in *RoleImplementation*. *RoleImplementation* is equivalent to *Scenario* in the current prototype code. A *NIC* stores the MAC address(es) that belong to an instance and a *Network* stores the network(s) that an instance is connected to. Moreover, an instance has access to the ports that a role consumes or exposes (channels) through *ChannelSelection*. A *Channel* can be a single port or a range of ports. The cloud provider firewall configuration (known as "security groups" in OpenStack) is captured in *SecurityGroup*. One *SecurityGroup* can have multiple configuration entries, *SecurityRules*. *ProviderEndpoint* captures the cloud platform specific API. This component makes it easier to integrate ANCOR with different cloud providers.

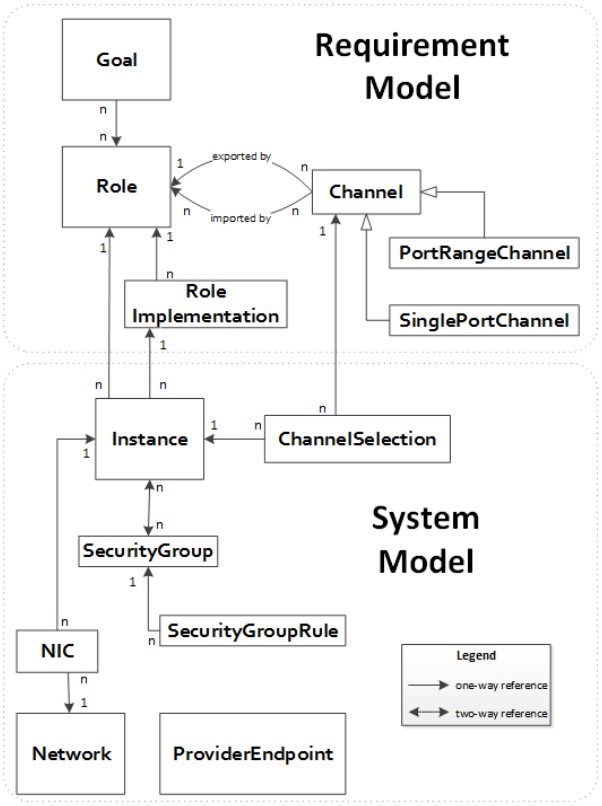

Figure 14: Operations Model

References

[1] A.J.Oliner and A.Aiken. A query language for understanding component interactions in production systems. *In Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, pages 201–210, 2010.

[2] J. Albrecht, C. Tuttle, R. Braud, D. Dao, N. Topilski, A.C. Snoeren, and A. Vahdat. Distributed Application Configuration, Management, and Visualization with Plush. *In ACM Transactions on Internet Technology (Volume: 11, Issue: 2, Article No. 6)*, 2011.

[3] M. Armbust, A. Fox, R Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, pages 50–58, 2010.

[4] AWS. CloudFormation - accessed 7/2014. https://aws.amazon.com/cloudformation/.

[5] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A Security Analysis of Amazon's Elastic Compute Cloud Service. *In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 1427–1434, 2012.

[6] K.M. Begnum. Managing Large Networks of Virtual Machines. *In Proceedings of the 20th USENIX conference on Large Installation System Administration (LISA)*, 2006.

[7] S. Bugiel, S. Nürnberger, T. Pöppelman, A. Sadeghi, and T. Schneider. AmazonIA: When Elasticity Snaps Back. *In Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*, pages 389–400, 2011.

[8] CloudVelocity. One hybrid cloud software. Technical report, CloudVelocity Software, 2013.

[9] Deis website - accessed 3/2014. http://deis.io/overview/.

[10] Docker. Learn What Docker Is All About - accessed 1/2014. https://www.docker.io/learn_more/.

[11] Craig Dunn. Designing Puppet: Roles and Profiles - accessed 10/2013. http://www.craigdunn.org/2012/05/239/.

[12] C. C. Evans. YAML - accessed 8/2013. http://yaml.org/.

[13] William Fellows. 'Run any app on any cloud' is CliQr's bold claim. 451 Research, Jan 2013. http://cdn1.hubspot.com/hub/194983/Run_any_app_on_any_cloud.pdf.

[14] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[15] Flynn website - accessed 4/2014. https://flynn.io/.

[16] Google Developers - accessed 12/2013. https://developers.google.com/appengine/docs/python/config/appconfig.

[17] HAProxy website - accessed 2/2014. http://haproxy.1wt.eu/.

[18] Puppet Labs. External Node Classifiers - accessed 9/2013. http://docs.puppetlabs.com/guides/external_nodes.html.

[19] Puppet Labs. What is MCollective and what does it allow you to do? - accessed 9/2013. http://puppetlabs.com/mcollective.

[20] Puppet Labs. What is Puppet? - accessed 9/2013. http://puppetlabs.com/puppet/what-is-puppet.

[21] LXC - Linux Containers - accessed 1/2014. http://linuxcontainers.org/.

[22] Maestro Github repository - accessed 1/2014. https://github.com/toscanini/maestro.

[23] Maestro-NG Github repository - accessed 4/2014. https://github.com/signalfuse/maestro-ng.

[24] Mirantis Software Website - accessed 3/2014. http://software.mirantis.com/.

[25] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. *In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2013.

[26] MySQL website - accessed 9/2013. http://www.mysql.com/.

[27] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[28] Sanjai Narain, Sharad Malik, and Ehab Al-Shaer. Towards eliminating configuration errors in cyber infrastructure. In *Proceedings of 4th IEEE Symposium on Configuration Analytics and Automation (SafeConfig)*, pages 1–2. IEEE, 2011.

[29] Nginx website - accessed 1/2014. http://nginx.org/.

[30] Unicorn! Repository on GitHub. What it is? - accessed 4/2014. https://github.com/defunkt/unicorn.

[31] OpenStack. Heat - accessed 7/2014. https://wiki.openstack.org/wiki/Heat.

[32] OpenStack. Heat/CloudFormation MySQL Template - accessed 5/2014. https://github.com/openstack/heat-templates/blob/master/cfn/F17/MySQL_Single_Instance.template.

[33] OpenStack. Heat/CloudFormation Wordpress Template - accessed 5/2014. https://github.com/openstack/heat-templates/blob/master/cfn/F17/WordPress_With_LB.template.

[34] Opscode. Chef - accessed 3/2014. http://www.opscode.com/chef/.

[35] PuppetLabs Docs - accessed 2/2014. http://docs.puppetlabs.com/hiera/1/.

[36] RightScale. White Paper - Quantifying the Benefits of the RightScale Cloud Management Platform - accessed 8/2013. http://www.rightscale.com/info_center/white-papers/RightScale-Quantifying-The-Benefits.pdf.

[37] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, and M. Rosenblum. Virtual Appliances for Deploying and Maintaining Software. *In Proceedings of the 17th USENIX conference on Large Installation System Administration (LISA)*, pages 181–194, 2003.

[38] Service-now.com. White Paper - Managing the Cloud as an Incremental Step Forward - accessed 8/2013. http://www.techrepublic.com/resource-library/whitepapers/managing-the-cloud-as-an-incremental-step-forward/.

[39] Amazon Web Services. AWS OpsWorks - accessed 4/2014. http://aws.amazon.com/opsworks/.

[40] Sidekiq Repository on GitHub - accessed 2/2014. https://github.com/mperham/sidekiq.

[41] Ubuntu. Juju - accessed 7/2014. https://juju.ubuntu.com/.

[42] Steven J. Vaughan-Nichols. Canonical Juju DevOps Tool Coming to CentOS and Windows. http://www.zdnet.com/canonical-juju-devops-tool-coming-to-centos-and-windows-7000029418/.

[43] BCFG2 Website. What is Bcfg2? - accessed 4/2014. http://docs.bcfg2.org/.

[44] CFEngine Website. What is CFEngine? - accessed 3/2014. http://cfengine.com/what-is-cfengine.

[45] OpenStack Website. Open Source Software for Building Private and Public Clouds - accessed 4/2014. http://www.openstack.org/.

[46] OpsView Website. OpsView - accessed 4/2014. http://www.opsview.com/.

[47] Redis Website. Redis - accessed 4/2014. http://redis.io/.

[48] Ruby Website. Ruby is... - accessed 4/2014. https://www.ruby-lang.org/en/.

[49] SaltStack Website. What is SaltStack? - accessed 4/2014. http://saltstack.com/community.html.

[50] Sensu Website. Sensu - accessed 4/2014. http://sensuapp.org/.

[51] Snort Website. What is Snort? - accessed 4/2014. http://www.snort.org/.

[52] Suricata Website. - accessed 4/2014. http://suricata-ids.org/.

[53] Terraform Website. Terraform - accessed 9/2014. http://www.terraform.io/intro/index.html.

[54] Terraform Website. Terraform vs. Other Software - accessed 9/2014. http://www.terraform.io/intro/vs/cloudformation.html.

[55] Varnish Cache Website. About - accessed 4/2014. https://www.varnish-cache.org/about.

[56] WordPress Website. WordPress - accessed 4/2014. https://wordpress.org/.

[57] Windows Azure Service Definition Schema (.csdef) - accessed 3/2014. http://msdn.microsoft.com/en-us/library/windowsazure/ee758711.aspx.

[58] wrk Github Repository - accessed 4/2014. https://github.com/wg/wrk.

[59] L. Zomlot, S. Chandran Sundaramurthy, K. Luo, X. Ou, and S.R. Rajagopalan. Prioritizing intrusion analysis using Dempster-Shafer theory. *In Proceedings of the 4th ACM workshop on Security and artificial intelligence (AISec)*, pages 59–70, 2011.