Engage: A Deployment Management System

Jeffrey Fischer
genForma Corp, USA
jeffrey.fischer@genforma.com

Rupak Majumdar
MPI-SWS, Germany
rupak@mpi-sws.org

Shahram Esmaeilsabzali
MPI-SWS, Germany
shahram@mpi-sws.org

Abstract

Many modern applications are built by combining independently developed packages and services that are distributed over many machines with complex inter-dependencies. The assembly, installation, and management of such applications is hard, and usually performed either manually or by writing customized scripts. We present Engage, a system for configuring, installing, and managing complex application stacks. Engage consists of three components: a domain-specific model to describe component metadata and inter-component dependencies; a constraint-based algorithm that takes a partial installation specification and computes a full installation plan; and a runtime system that co-ordinates the deployment of the application across multiple machines and manages the deployed system. By explicitly modeling configuration metadata and inter-component dependencies, Engage enables static checking of application configurations and automated, constraint-driven, generation of installation plans across multiple machines. This reduces the tedious manual process of application configuration, installation, and management.

We have implemented Engage and we have used it to successfully host a number of applications. We describe our experiences in using Engage to manage a generic platform that hosts Django applications in the cloud or on premises.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management; K.6.2 [*Management of Computing and Information Systems*]: Installation management

General Terms Design, Languages, Management

Keywords Application deployment, declarative languages, cloud computing

1. Introduction

Many modern applications are built from components which are inter-dependent, distributed, and created independently. By combining existing components in a loosely-coupled architecture, applications delivering significant functionality in a scalable manner can be quickly created. However, the assembly, installation, and management of such systems in a consistent and effective way is currently a difficult and resource-intensive problem. Packages and services have dependencies that have to be satisfied, and misconfigurations between individual components and services can cause the application to become unstable or corrupted.

While there are tools that automate portions of this process for specific packages, e.g., package managers on Linux distributions, there are few tools that provide end-to-end functionality for the entire software stack, especially for distributed applications running on many nodes (e.g., in a private or public cloud infrastructure). In practice, large-scale software stacks are often managed using custom script-based tools and manual techniques. This process is error-prone, as upgrades to individual packages can break existing and implicit dependencies in the system. It is also labor-intensive, since the process must be performed manually by system administrators, developers, and end-users, and effort may be duplicated within the same organization. In fact, application administration overhead is mentioned as one main barrier to the adoption of free and open-source software [13].

The challenge of dependency management in software package installation is well-studied, and constraint-based tools to manage package installations on single machines have been developed [12, 17, 19]. However, none of the available tools consider additional factors that come up in managing a distributed application stack:

1. **Physical context**: Components may be distributed across multiple machines. Certain dependencies (e.g., libraries) have to be resolved within the context of a single machine (e.g., each machine running a Java component must install (possibly different versions of) the Java Runtime Environment), and certain dependencies must be resolved across machines (e.g., an application server and a back-end task processor may live on different servers but connect to the same database). Existing tools do not consider multi-machine dependencies.
2. **Configuration management**: Additionally, each component of an application may have its own collection of configuration settings. Such settings may be maintained in a configuration file or database or passed to a component through command line parameters or environment variables. These settings may determine how a component interacts with its dependencies. One must ensure that all selected components of an application have been configured to connect to each other as required by their dependency relationships. For example, an application depending on a database must be configured to connect to the host and port number on which the database is listening, and may need to know the database user's name and password. Currently, it is the user's responsibility to connect the configuration options between components (e.g., by manually setting the values in the application's configuration file).
3. **Application management**: Many application components use long-running processes (e.g. services or daemons) that need to be explicitly started, restarted, or stopped. Startup/shutdown of the full application stack can be deceptively difficult to automate due to timing issues between dependent services. If a component is started without first ensuring that all of its dependen-

PLDI'12, June 11–16, 2012, Beijing, China.

cies have completed their startup, it might intermittently fail due to connection errors. Most package management tools do not consider service startup/shutdown, and do not provide ways to express dependencies in the application stack.

We present Engage, a system for managing application stack configuration, installation, and maintenance. Engage has three components. The first is a declarative framework for specifying component metadata such as configuration parameters as well as dependencies between different components and services. The second is a constraint-based algorithm that takes a *partial installation specification* (a list of the main application components to be installed) and automatically produces a full installation specification (a list of all components to be installed). Finally, a runtime system co-ordinates the installation of components in an installation specification and manages the deployed system locally or on the cloud.

In Engage, a component, called a *resource*, is modeled using two parts: a *type* and a *driver*. A *type* for the resource describes configuration options and dependencies associated with a component. The configuration options specify required configuration parameters for the components, including input configuration options that are provided by components that the current resource depends on, and output configurations that the configured resource exports to downstream dependencies. The dependencies specify other components the current resource depends on, such as the machine in which it executes, other libraries and services that must be present on the same machine, and other services (possibly on diffferent machines) that it requires. A *driver* for the resource specifies a state machine that manages its lifecycle at runtime (e.g., installation, startup, shutdown, etc.). Together, the type and the driver specify completely the dependencies and the configurations required to install and use a resource. A subtyping mechanism ensures that each resource can be developed independently and reused in many different contexts.

A *resource instance* is a specific instantiation of a resource that is deployed, for instance, a specific server or a specific instance of a MySQL database. An application is described by compositions of resource instances (an *installation specification*), and it can be checked statically to ensure that all dependencies are met, there are no circular or conflicting dependencies, and that configuration parameters between dependent components are passed correctly. For example, we can check that, if an application component has a dependency on a database, there is a resource instance in the installation specification corresponding to a database component, and the application component gets its input configuration options for the database from the outputs of the database component. Since resources can model the physical context of a machine, we can statically reason about the physical context; for instance, checking that certain libraries exist on the same machine. Additionally, the resource drivers ensure that the runtime system has enough information to start up and shut down services according to their dependencies. Thus, the resource types enable static checking of application stacks for dependencies and conflicts, for configuration and physical context management, and the types together with the drivers enable runtime application management.

Rather than writing the entire installation specification, which can be long and tedious, Engage allows the user to only specify a *partial installation specification*, which outlines the main application components and the machines they should be installed on. A *configuration engine* takes the partial installation specification and produces a *full installation specification* by computing the transitive closure of dependencies of the components, resolving choices between components through Boolean constraint solving, and adding component dependency links to form a directed acyclic graph of resource instances.

The Engage runtime system then takes this full installation specification and deploys it (locally or on the cloud), using the dependency order on resource instances and the actions in the resource drivers to order the sequence of installations and startups. The drivers specify the order in which operations can be performed on a resource instance, and also implement each operation using an underlying programming language. The runtime system monitors the application as it runs, and can manage the application stack (e.g., to stop the application) based on the dependency order specified in the install specification. At the end of this process, the user should have a full application installed, configured, and running, potentially across multiple machines.

We have implemented the Engage system and have used Engage as the deployment and management tool for a number of case studies. The core of Engage contains about 26K lines of Python code, 6K lines of OCaml code, and about 5K lines of metadata. Engage is integrated with both Rackspace and Amazon Web Services and can deploy applications on these services. Engage has been used as the core technology behind a commercial platform-as-a-service company for application stacks written in Django and using standard components such as Python packages, databases (MySQL), key-value stores (Redis, MongoDB), message queues (RabbitMQ, Celery), and web caches (Memcached). It has efficiently managed the configuration, deployment, and upgrades of complex third-party applications. In addition, we have used Engage to configure and manage Java-based and pure Python applications.

Related Work Engage is different from an *operating-system-level package manager* (OSLPM), such as dpkg for Debian, RPM, or apt. These tools manage dependencies among packages on a single machine, but do not provide support for configuration management, application management (e.g., server startup), or multi-machine deployments. However, they form the building blocks for Engage: a driver for a resource can use an OSLPM to install the required packages on a machine. Thus, Engage is complementary to OSLPM, and uses OSLPM's to provide support for higher-level issues in application deployment and maintenance for each platform.

Constraint-based tools [11, 17–19] manage individual software packages at a single machine level, but do not provide support for configuration management, multi-machine deployments, or application management. A constraint usually specifies pre-requisites (and conflicts) for a package, but does not model configuration parameters or service startup/shutdown requirements. The constraint-based configuration in Engage is inspired by these tools, but in addition, Engage provides support for configuration and application management across multiple physical contexts (machines). The distribution editor of the EDOS project [17] provides a means to check the consistency and installability of a set of interdependent packages. However, in the distribution editor, this information is not used for analysis or for application-level installation. Model-based formalisms are used to detect compatibility and upgrade problems [10, 11], but not across physical contexts, nor targeted toward application stack management.

The input/output/configuration ports of Engage's resource types were inspired by module systems for more general purpose languages, including Units [14], nesC [15], and ArchJava [7]. Unlike these systems, the connections between modules (resources) in Engage are implicit, using dependency constraints to instantiate resources and establish connections. Engage's resource definition language provides a combination of traditional module systems (importing/exporting definitions, subtyping of individual values and structures) and constraint languages.

Engage is similar to *configuration management* systems such as Puppet [1], Chef [2], LCFG [8], and CfEngine [9], which provide a domain-specific language to model the "desired state" for a machine, e.g., files that should be present and services that should be

running. These systems use a client-server model in which a server holds the model of the desired state for a machine, and the client ensures this configuration is met. Engage improves upon the design of existing configuration management systems in the following ways. First, Engage separates the declarative specification of resources (resource types) and the imperative details of their installation (instances and drivers). This enables type-checking for configurations and to detect configuration or dependency problems statically. Second, through the constraint-based configuration engine, Engage allows the compact specification of application stacks, usually over an order of magnitude smaller than the full installation specification required by other systems. In practice, we have found the combination of instance/type separation and a contraint-based dependency model result in very flexible install specifications. We can take an application and deploy it on multiple platforms (e.g. MacOSX and Linux) and in multiple configurations (e.g. development, testing, and production) without significantly more work than is required for a single configuration. This is not possible with traditional configuration management systems, which are optimized for keeping large numbers of nearly identical systems in a given standard configuraiton.

Finally, while configuration management systems can be used to support resource management over multiple servers, each client is usually managed in isolation with no coordination between servers. Engage provides a means to coordinate and configure between services across machines according to their dependencies.

SmartFrog [16] supports distributed application deployment by modeling the components of a system and providing a framework for *lifecycle managers* to manage individual components and their coordination. Engage's deployment engine uses a similar approach, with drivers in lieu of lifecycle managers. Engage additionally uses the dependency relationship information it obtains from the configuration phase to determine the sequencing of deployment actions. The use of constraint-based configuration is also novel.

2. Overview

We now describe Engage through an example. As noted before, Engage has three main components:

1. A declarative language to describe *resources*. A resource is a generic term for a software package (e.g., the Apache web server), operating system (e.g., Ubuntu 10.4), or a virtual or physical machine. Its description consists of a *type* (the configuration parameters and dependencies) and a *driver* (a state machine for managing the component).
2. A *configuration engine* that takes a database of resources and a *partial installation specification* and produces a *full installation specification* that describes how a collection of software components should look when installed and configured.
3. The Engage runtime, consisting of a *deployment engine* and runtime services that install and manage components in a deployment. It calls the drivers of each component in the full installation specification and starts services in dependency order.

OpenMRS We demonstrate how Engage can be used to install and manage OpenMRS, an open source enterprise medical records system [20]. OpenMRS is written in Java, and runs as a servlet contained within the Apache Tomcat webserver. It depends on Java, version 5 or greater. Either the Java Developer Kit (JDK) or Java Runtime Environment (JRE) may be used. The Tomcat distribution should be at least version 5.5 but before 6.0.29. OpenMRS talks to a MySQL database in the backend, where the MySQL distribution should be version 5 or greater. In a production setting, the database will run on a separate machine from the application server. OpenMRS itself does not have any specific operating system dependen-

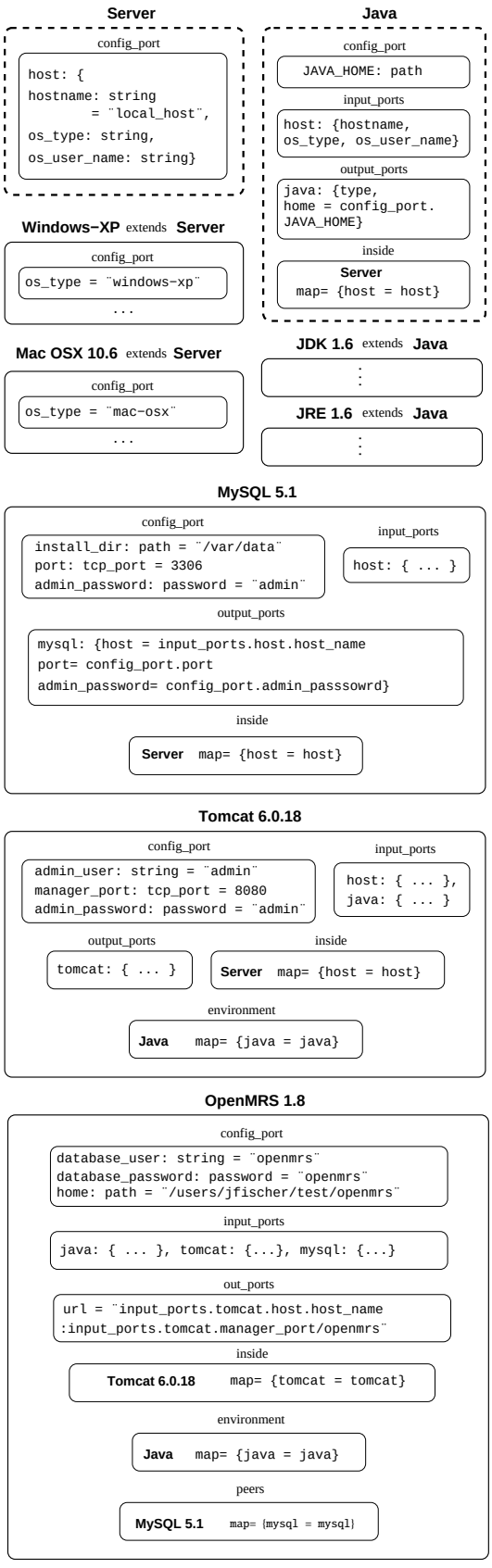

Figure 1: Resource types for OpenMRS installation

```
{ "id": "server", "key": "Mac-OSX 10.6",
  "config_port": {
    hostname="localhost", os_user_name="root"
  }
},
{ "id": "tomcat", "key": "Tomcat 6.0.18",
  "inside": { "id":"server" }
},
{ "id": "openmrs", "key": "OpenMRS 1.8"
  "inside": { "id":"tomcat" }
}
```

Figure 2: Partial installation specification for OpenMRS

cies. Thus the operating systems are limited by the availability of Java and MySQL. This includes Windows XP/Vista, Linux, Solaris, and Mac OSX. Different executables for Java and MySQL are required for each platform.

A single-machine installation for OpenMRS is described on the OpenMRS web page, and consists of a 10-page document. This document gives a list of the software components which must be installed in order to install OpenMRS, in their dependency order. It points to a different document for installing and configuring MySQL. Moreover, the document gives separate conditions for Windows and for Ubuntu (after 10.10 and before 10.10). The installation requirements and dependencies in the installation are given as comments in the document. We give two examples:

- Java must be installed before installing Tomcat.
- [When starting OpenMRS] Ensure that Tomcat is started by checking to see if icon in the tray is green (Windows install).

It is the user's responsibility to troubleshoot the installation if some of these conditions are not met. When moving from one deployment (e.g., test) to another (e.g., production), the user must manually reconstruct all the installation steps (or write ad hoc scripts to automate the steps). Moreover, the instructions are sometimes incomplete (for example, how to check if Tomcat is started on Ubuntu? How to check it programmatically?). Our intent is not to deride the OpenMRS installation document (indeed, OpenMRS has a very readable set of instructions), but to point out that these are common issues in many installation specifications.

Engage We now describe how the OpenMRS installation is managed in Engage. For simplicity, we only describe the configuration for a Mac OSX 10.6 server with Tomcat 6.0.18. Software or hardware components, called *resources*, are represented in Engage using two parts: a *resource type* describes the configuration metadata and dependencies on other resources, and a *resource driver* describes a state machine that manages the lifecycle of the resource. We assume resource types and drivers are written by the component developer, not the end user installing the application.

We first describe resource types. Figure 1 shows the resource types that are relevant to the OpenMRS installation (simplified for readability). (We omit describing a concrete syntax for resources.) Each resource description has a (unique) key identifying the resource, sets of typed configuration, input, and output ports, and dependency specifications on other components. Typically, the key consists of the name of the package and its version. The ports specify attributes (key-value pairs) internal to the component's configuration (the *configuration* ports), attributes derived from other components on which the component depends (*input* ports), and attributes that the component exports to downstream components (*output* ports). For example, the resource type for key `MySQL 5.1` (MySQL version 5.1) has a configuration port called `port` that specifies the default port (3306), and has an output port, also called `port`, that copies its value from the configuration port for downstream components connecting to it.

The dependencies come in three types. An *inside dependency* specifies the container resource in which the current resource must execute. In general, this is a physical or virtual machine, or a "container" like Tomcat which contains other packages. If a resource does not have an inside dependency, it corresponds to a virtual or physical machine. For example, the (abstract) resource `Server` and its subclasses `Mac OSX 10.6` and `Windows-XP` refer to machines. On the other hand, the resource `Tomcat 6.0.18` has an inside dependency on a `Server`, meaning that in a deployment, `Tomcat` runs inside a sub-class of resource type `Server` (since `Server` is abstract and cannot be instantiated). The resource `OpenMRS 1.8` has an inside dependency on `Tomcat 6.0.18`, since it executes as a servlet inside the `Tomcat` server.

An *environment dependency* specifies other resources that must be present (and possibly execute) on the same machine as a precondition for the installation and execution of the current resource. For example, both `Tomcat` and `OpenMRS` have environment dependencies on `Java`, which means that an instance of a subclass of resource type of `Java` must be installed on a machine before `Tomcat` or `OpenMRS` can be installed on that machine. In addition, an environment dependency also specifies how configuration options from the output ports of a resource flow into the input ports of the current resource. For example, the output port `java` in the `Java` resource type is an input to `OpenMRS 1.8`

A *peer dependency* specifies resources must be present, but not necessarily on the same machine. For example, `OpenMRS 1.8` has a peer dependency on the `MySQL 5.1` database, meaning that an instance of `MySQL 5.1` must be deployed before `OpenMRS 1.8` can be deployed, but possibly on a machine different from that of OpenMRS.

When a resource type *A* depends on a resource type *B*, the resource *A* can obtain the value of an output port of *B* by using a *port mapping*. A port mapping relates an output port of a resource type to an input port of another. For example, `OpenMRS 1.8` maps the output port `mysql` of `MySQL 5.1` into its input port that has the same name.

Well-formed sets of resource types ensure that the union of all dependency relations is acyclic, and that each input port is mapped exactly once by the set of dependencies. Given a resource, we call the set of resources that it transitively depends on its set of *upstream dependencies* and the set of resources that transitively depend on it its set of *downstream dependencies*.

Resource *types* correspond to the classes in an OO language, resource *instances* represent specific resources instantiations (e.g. a copy of `MySQL 5.1` installed on server `demotest` and listening on port 3306) and correspond to objects. An instance of a resource has a globally unique identifier, fills in concrete values for all ports, and replaces each dependency in the resource type with a concrete instance. A full installation specification provides a list of each instance required to deploy an application. For example, given *resource types* for each component required by OpenMRS, the user can write an installation specification describing all the components that are required to install OpenMRS. Engage's type system can check the installation specification to make sure all required dependencies are present in the correct physical context and that each instance is correctly configured.

Writing full installation specifications can get tedious. Engage's configuration engine allows the end user to provide *partial installation specifications*, and automatically determines what other resources must be instantiated to deploy the system. Figure 2 shows the partial installation specification written by a user for OpenMRS. It consists of three resource instances: an instance `server` for a machine running Mac OSX version 10.6, an instance `tomcat`

of Tomcat running inside `server`, and an instance `openmrs` for the OpenMRS application running inside `tomcat`. In particular, the user does not have to explicitly give the other dependencies on Java and MySQL. The partial installation specification may also define values for individual configuration port properties. In our example, the `hostname` and `os_user_name` properties have been assigned values. Unassigned configuration properties will take the default values defined in the associated resource types.

The configuration engine takes this partial installation specification and expands out all the dependencies to generate a set of Boolean constraints such that the Boolean constraints are satisfiable iff there is a full installation specification which includes all of the resource instances mentioned in the partial installation specification. The atomic propositions in the Boolean constraints consist of instances of resources: the proposition is true in a satisfying assignment iff the corresponding resource instance must be deployed. For the partial installation specification of Figure 2, the configuration engine generates the following constraints:

`server` $\wedge$	from install spec
`tomcat` $\wedge$	from install spec
`openmrs` $\wedge$	from install spec
`openmrs` $\rightarrow \boxtimes\{$`jdk`, `jre`$\} \wedge$	env dep
`tomcat` $\rightarrow \boxtimes\{$`jdk`, `jre`$\} \wedge$	env dep
`openmrs` $\rightarrow$ `mysql` $\wedge$	peer dep
`tomcat` $\rightarrow$ `server` $\wedge$	inside dep
`openmrs` $\rightarrow$ `tomcat` $\wedge$	inside dep
`mysql` $\rightarrow$ `server` $\wedge$	inside dep
`jdk` $\rightarrow$ `server` $\wedge$	inside dep
`jre` $\rightarrow$ `server`	inside dep

where $\boxtimes S$ is the "exactly one" predicate that asserts that exactly one proposition from the set S is true. The first three constraints arise from the partial installation specification (each instance there must be deployed). The next three arise out of environment (and peer, for the last one) dependencies, and state that the deployment of the l.h.s. implies the deployment of the r.h.s. The final five arise out of inside dependencies. Note that the peer dependency of OpenMRS on MySQL could be resolved by "creating" a new machine instance and installing MySQL on that machine. However, our constraint generation process assumes that no new machines should be created. Thus, unless explicitly specified, a peer dependency is deployed at the same machine as the machine of its dependent.

The constraints are satisfied, e.g., by setting `server`, `jdk`, `tomcat`, `mysql`, and `openmrs` to true, and `jre` to false. This corresponds to a deployment where a Java development kit, a Tomcat server, a MySQL database instance, and the OpenMRS application are all installed on a server running Mac OSX 10.6. Given this solution, we can also "tie together" the input and output ports by traversing the resource instances in topological order of dependencies, starting with the output ports of `server`, and using the definitions of output ports of preceding resource instances to get values of input ports according to the port mappings specified in the dependencies. Valuations to the input ports then determine the configuration and output ports of the instance. In this way, we can propagate configuration options along the application stack. The result of this process is a *full installation specification*, that details the components that must be installed, their configuration parameters, and the order of their installation. The last is obtained via the partial order imposed by the dependencies of the resource instances.

Finally, the Engage deployment system takes an install specification and deploys the components in the order of dependencies. The deployment system uses the corresponding driver for each resource instance. A resource driver is a state machine with special states *uninstalled*, *active*, and *inactive* (the latter two possibly the same state), with guarded actions between states. Figure 3 shows an example resource driver for Tomcat. An action (e.g., `install`) is implemented in an underlying programming language and performs some modification of the system state. For example, the `install` action can call an OS-level package manager to download and install a package. A guard describes a precondition for the action, and is omitted if an action can be performed at any time. A guard of the form $\uparrow s$ (respectively, $\downarrow s$) states that an action can be performed only when the state machines of all upstream (respectively, downstream) dependencies are in state $s \in \{$*uninstalled*, *active*, *inactive*$\}$. The actions on the state machines are performed by the Engage runtime system. The runtime system manages the state machines of all installed components and can check the status of guards. The runtime system can also monitor a deployment and shut it down (by shutting down services in the reverse order of dependencies).

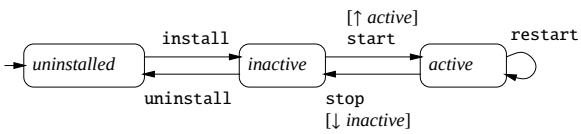

Figure 3: Resource driver for Tomcat

Engage ameliorates the problems of manual deployment in the following ways. First, dependencies among resource types ensure that required packages are installed when a resource instance is being installed (e.g., `Java` has already been installed before installing `Tomcat`). Second, the resource drivers and the runtime ensure that required services are already started when starting an application (e.g., `Tomcat` and `MySQL` daemons are already started when `OpenMRS` is started: their resource drivers are both in *active* when starting `OpenMRS`). While there is work involved in developing resource types and drivers, they are done once by the package developer, and any subsequent installation scenario involving the resource is completely automated. In contrast to ad hoc custom scripts, the declarative language enables static detection of configuration problems, e.g., cyclic dependencies between components, or unsolvable constraints in installation. In contrast to configuration management systems, the configuration engine significantly reduces user input: in our implementation, the (unsimplified) `OpenMRS` partial installation specification took 22 lines, and the full installation specification was 204 lines. Note that resource descriptions are reusable and can be used in different installations; e.g, the `MySQL` resource can be used in any deployment that requires a MySQL database.

3. Resource Types

In Engage, the fundamental abstraction for software and hardware components is a *resource*. A resource consists of a description of the metadata required to configure, install, or upgrade a component (its *resource type*) and a *driver* consisting of code that reads the metadata and manages the lifecycle of the component (installation, upgrade, rollback, etc.). We explain resource types here, and explain drivers in Section 5.

3.1 Resource Types

A *resource type* is an abstraction to model how a component may be instantiated. A *resource instance* is an instantiation of a resource type that describes how a specific resource will be (or has been) configured and installed. In analogy with object-oriented programming, a resource type is a class and a resource instance is an object of that class. Each resource type has a unique identifier (usually,

name of the component and its version) in a global name space. In addition, a resource type has ports and dependency constraints.

Ports Configuration data for a component are described in *ports*. A port has a *name* and a *type*. We assume an (unspecified) set of base types. For a port p, we write $p.\mathsf{name}$ and $p.\mathsf{type}$ to refer to the name and the type, respectively. Communication of configuration data between components is performed by connecting appropriate ports of two components.

Dependencies Dependencies between resource types specify ordering constraints between components, and determine how ports of a resource get their values. Engage specifies three types of dependencies.

An *inside dependency* specifies that an instance of a resource type can exist only inside an instance of another resource type; e.g., Tomcat can exist only "inside" a Server. Each resource type has either zero inside dependencies, in which case it must represent a physical machine, or exactly one inside dependency. Inside dependencies can be nested: e.g., a servlet can be inside Tomcat, which in turn is inside a server. For any resource type, one can walk the inside dependencies to eventually reach a physical machine, and this is the machine on which the component gets installed and run.

A resource type R_1 has an *environment dependency* on a resource type R_2 if it depends on a resource of type R_2 to be installed in an enclosing resource. As an example, Tomcat requires that the Java runtime environment be installed on the machine it is installed on. (However, Tomcat is not installed "inside" Java.)

A resource type R_1 has a *peer dependency* on a resource type R_2 if it calls services provided by an instance of R_2. In this case, unlike environment constraints, an instance of R_2 need not be installed on the same machine as the instance of R_1. For example, OpenMRS has a peer dependency on MySQL, and OpenMRS can make calls to the database over TCP/IP.

Formal Model Formally, a resource type

$$R = \langle \mathsf{key}, \mathsf{InP}, \mathsf{ConfP}, \mathsf{OutP}, \mathsf{Inside}, \mathsf{Env}, \mathsf{Peer} \rangle$$

consists of a (globally unique) *key* key (usually consisting of a name and a version), three disjoint sets of *input ports* InP, *configuration ports* ConfP, and *output ports* OutP, an optional *inside dependency* Inside or `null`, a set of *environment dependencies* Env, and a set of *peer dependencies* Peer. We assume each port $p \in \mathsf{ConfP}$ is either a *default* constant or defined as a function of the ports in InP, and each port $p \in \mathsf{OutP}$ is either a default constant or defined as a function of the ports in $\mathsf{InP} \cup \mathsf{ConfP}$. For a key key, we write $[\![\mathsf{key}]\!]$ to denote the (unique) resource type for that key. For a resource type R, we write $R.\mathsf{key}$, $R.\mathsf{InP}$, etc. to denote the components of the resource type.

The input ports are used to receive data from other resources. The output ports are used to affect the behavior of other resources by providing configuration data. The configuration ports are used to store resource-specific metadata used in configuration and installation. A configuration port of a resource can read from an input port of the same resource, and an output port of a resource can read from an input or config port of the resource.

Each dependency (inside, environment, or peer) is a pair $(\mathsf{key}', \mathsf{pmap})$, where key' is a key to a resource and pmap is a partial mapping from $[\![\mathsf{key}']\!].\mathsf{OutP}$ to $R.\mathsf{InP}$.

Inside, environment, and peer dependencies induce ordering relations on resource types. Let R and R' be resource types. We write $R \preceq_i R'$ (the *inside* ordering) if $R.\mathsf{Inside} = (R'.\mathsf{key}, \cdot)$. We write $R \preceq_e R'$ (the *environment* ordering) if $(R'.\mathsf{key}, \cdot) \in R.\mathsf{Env}$. We write $R \preceq_p R'$ (the *peer* ordering) if $(R'.\mathsf{key}, \cdot) \in R.\mathsf{Peer}$.

A *machine* is a resource whose inside dependency is `null`.

Well-formedness A finite set of resource types is called *well-formed* if the following conditions hold:

1. Each key appearing in an inside, environment, or peer dependency is mapped to a resource type in the set (no pending dependencies).
2. If a resource does not have an *inside* dependency, it does not have any input ports. Such resources usually correspond to physical machines.
3. Each input port is mapped exactly once in the port mappings of the inside, environment, and peer dependencies. Each output port is assigned a value. That is, the range of port mappings in two dependencies is disjoint, and the union of ranges is the set of all input ports.
4. The ordering $\preceq_i \cup \preceq_e \cup \preceq_p$ on resource types is acyclic.

For any resource type R in a well-formed set of resource types, one can follow inside constraints to a machine. This represents the physical context in which an instance of R will be installed.

3.2 Abstract Resources and Subtyping

We extend the core language with inheritance and abstract resource types. A resource type can be marked *abstract* or *concrete*. An abstract resource type cannot be instantiated, but can be used for inheritance. For example, in Section 2, we defined an abstract resource type `Server` for a server, and subclasses for Mac OSX and Windows servers. The generic `Server` cannot be instantiated, but its subclasses specifying the OS can be instantiated. Instances of concrete resource types can be deployed. By default, we assume resource types are concrete and omit the concrete qualifier.

Sub-resource types extend base resource type definitions by adding ports and dependencies. Fields from a super-resource type are implicitly replicated in the sub-resource type, or overridden as explained below.

Figure 4 presents the subtyping rules to ensure well-formedness of the sub-resource tree. The rules define subtyping relations $\leq_{in}$, $\leq_{conf}$, and $\leq_{out}$ for input, config, and output ports, respectively, their extensions to sets of ports and port mappings, and a $\leq_{RT}$ relation that checks for subtyping between resource types by checking subtyping of the corresponding resource types of their inside, environment, and peer dependencies. We assume an (unspecified) subtyping relation $\leq$ on the base types over which ports are defined. Note that the subtyping relation on base types go on opposite directions for input ports and for config and output ports. This is related to the usual co-variance and contra-variance of method arguments. Sub-resource types extend base resource types by defining more input, config, and output ports, subtyping the inside dependency, and adding additional environment and peer dependencies.

3.3 Resource Instances

A *resource instance* is created from a resource type by assigning concrete values to its configuration ports and by replacing dependency constraints with directional links to other resource instances. In addition to the components of a resource type, each resource instance has a globally unique *resource identifier* that uniquely identifies the resource instance. The identifier helps distinguish two instances of the same resource type in a deployment, e.g., two servers both running Mac OSX 10.6.

Deployment information for an application can be provided by giving a set of resource instances obtained from corresponding resource types, such that for every instance in the set, each dependency is instantiated with a concrete link to another resource instance, and the values in the output ports and mapped input ports for the resource instances match. Such a document is referred to as an *installation specification*.

$$\textsc{Input ports}\quad \frac{p'.\mathsf{name} = p.\mathsf{name} \qquad p'.\mathsf{type} \leq p.\mathsf{type}}{p' \leq_{in} p} \qquad \textsc{Config ports}\quad \frac{p'.\mathsf{name} = p.\mathsf{name} \qquad p'.\mathsf{type} \geq p.\mathsf{type}}{p' \leq_{conf} p} \qquad \textsc{Output ports}\quad \frac{p'.\mathsf{name} = p.\mathsf{name} \qquad p'.\mathsf{type} \geq p.\mathsf{type}}{p' \leq_{out} p}$$

$$\textsc{Sets of input ports}\quad \frac{\forall p \in P \cdot \exists p' \in P' \cdot p' \leq_{in} p}{P' \leq_{IN} P} \qquad \textsc{Sets of config ports}\quad \frac{\forall p \in P \cdot \exists p' \in P' \cdot p' \leq_{out} p}{P' \leq_{CONF} P} \qquad \textsc{Sets of output ports}\quad \frac{\forall p \in P \cdot \exists p' \in P' \cdot p' \leq_{out} p}{P' \leq_{OUT} P}$$

$$\textsc{Port mappings}\quad \frac{\forall (p,q) \in m \cdot \exists (p',q') \in m' \cdot (p' \leq_{out} p \wedge q' \leq_{in} q)}{m' \leq_{pm} m} \qquad \textsc{Refl}\quad \frac{}{R \leq_{RT} R} \qquad \textsc{Trans}\quad \frac{R \leq_{RT} R' \quad R' \leq_{RT} R''}{R \leq_{RT} R''}$$

$$\textsc{Resource types}\quad \frac{\begin{array}{c} R'.\mathsf{InP} \leq_{IN} R.\mathsf{InP} \quad R'.\mathsf{ConfP} \leq_{CONF} R.\mathsf{ConfP} \quad R'.\mathsf{OutP} \leq_{OUT} R.\mathsf{OutP} \\ ([\![R'.\mathsf{Inside.key}']\!] \leq_{RT} [\![R.\mathsf{Inside.key}']\!] \vee [\![R'.\mathsf{Inside.key}']\!] = [\![R.\mathsf{Inside.key}']\!] = \mathsf{null}) \wedge (R'.\mathsf{Inside.pmap} \leq_{pm} R.\mathsf{Inside.pmap}) \\ \forall (I,m) \in R.\mathsf{Env} \cdot \exists (I',m') \in R'.\mathsf{Env} \cdot ([\![I']\!] \leq_{RT} [\![I]\!] \wedge m' \leq_{pm} m) \qquad \forall (X,m) \in R.\mathsf{Peer} \cdot \exists (X',m') \in R'.\mathsf{Peer} \cdot ([\![X']\!] \leq_{RT} [\![X]\!] \wedge m' \leq_{pm} m) \end{array}}{R' \leq_{RT} R}$$

Figure 4: The rules for subtyping between two resource types.

3.4 Syntactic Sugar and Extensions

For brevity, we introduce some syntactic sugar in the language and in our examples. We allow a port to be a structure with named fields. We allow disjunctions in the descriptions of inside dependencies, and positive Boolean combinations of resource types in environment and peer dependencies. For example, we can express that a resource type R (e.g., Tomcat) is inside either resource type R_1 (e.g., Mac-OSX 10.6) or resource type R_2 (e.g., Windows 7), and a resource type R (e.g., OpenMRS) has an environment dependency on resource type R_1 (e.g., Java) as well as one of R_2 (e.g., MySQL) or R_3 (e.g., Postgres). The use of disjunctions is a convenient shorthand, and can be simulated by defining the appropriate super-resource types and their sub-resource types. To simplify the check for well-formedness, we require the ranges of two port mappings that are disjunctively combined to be identical.

To model a common idiom in which a resource depends on one of a number of versions of a different one (e.g., OpenMRS depends on versions of Tomcat before version 6.0.29), we add syntactic sugar to allow specifying ranges of versions for the same package, which are internally expanded to disjunctions of the different versions satisfying the range.

The implementation of Engage also has some extensions that we do not model here. Each port of a resource type in our implementation has a *static* or *dynamic* binding. A static port is assigned a value at the instantiation time of a resource of that type. For example, these can be default values. A dynamic port is assigned a value at the installation time of a resource. By default, we assume all ports are dynamic (and we omit mentioning the binding explicitly). Only configuration and output ports can have the static binding. A static configuration port must be a constant. A static output port is either a constant or a function of static configuration ports. Inside, environment, and peer dependencies are extended with two port maps: in a dependency of resource R on R', one port map maps output ports of R' to the inputs of R (as before), and the other maps static output ports of R to input ports of R'. Static ports are useful to flow configuration parameters in the "reverse direction" of dependencies. For example, when installing OpenMRS, we need to pass a server configuration file back to Tomcat. In our implementation, we use static ports to achieve this. However, by the nature of static ports, all configuration values flowing in the reverse direction can be resolved statically, and these configurations will be known when installing R' even though R has not been installed yet.

4. Configuration

In practice, writing the entire set of resource instances in an installation specification can be tedious. Instead, Engage allows the user to specify a *partial* installation specification. A partial installation specification is a set of partial resource instances, that is, a set of resource instances for which only a subset of dependencies are instantiated by concrete resource instances. For example, Figure 2 shows a partial installation specification with instance `server` obtained from a resource of type Mac OSX 10.6, a resource instance `tomcat` for a resource of type Tomcat 6.0.18, and a resource instance `openmrs` for OpenMRS 1.8. An installation specification *extends* a partial installation specification if each resource instance in the partial specification is also present in the installation specification. Engage automatically generates a *full* installation specification extending the partial specification, by using the dependency relations in the resource types and automated constraint solving.

The configuration engine takes as input a collection of resource types and a partial install specification (a collection of partial resource instances) and produces as output a set of complete resource instances (called the (full) *installation specification*) such that: (1) for every resource instance, and for every inside, environment, or peer dependency of its corresponding resource type, there exists a resource instance that satisfies the inside, environment, or peer dependency, respectively, and (2) each input port of a resource instance is mapped to exactly one output port of a different resource instance.

Given a partial instantiation specification, $\mathcal{I}$, the configuration engine computes a set of Boolean constraints using the following two steps. In the following, we assume that there is a fixed, well-formed set of resource types, $\mathcal{R}$, in the system and that every resource instance in the partial install specification is an instantiation of some resource type from this set.

Hypergraph Generation The hypergraph generation phase takes a partial install specification and constructs a directed *resource instance graph* whose nodes are resource instances, and whose hyperedges represent dependencies between resource instances. Without loss of generality, we assume that for each resource instance, the inside dependency is either `null` or a disjunction of resources, and the environment and peer dependencies are a set of disjunctions of resources.

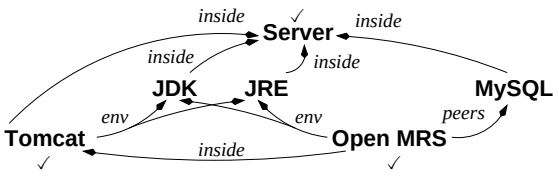

Figure 5: Hypergraph for partial installation specification in Figure 2

.

If a resource type has a dependency on an abstract resource r, we replace the dependency to r with a disjunction of concrete resources in the following way. We traverse the subtypes of r in the subclassing tree for r, starting at r and stopping whenever we see a concrete subtype of r. In this way, we get a "frontier" F of subtypes of r of concrete resources. We replace the dependency on r by a disjunction of the concrete resources in F. For example, if there is a dependency on the abstract resource `Java` (see Figure 1), it is replaced by the disjunction of the concrete resources `JDK 1.6` and `JRE 1.6`. After this transformation, there are no dependencies on abstract resources. (If such a frontier cannot be found, that is, if there is an abstract resource at the leaf of the subclassing tree rooted at r, we stop with an error.)

The hypergraph generation algorithm, GraphGen($\mathcal{R}, \mathcal{I}$), is a worklist-based algorithm to process instances. It proceeds as follows.

First, for every resource instance in the partial install specification, we create a node in the hypergraph and add it to the worklist. Recall that in addition to the key of the resource, each resource instance is globally uniquely identified by an additional identifier.

Second, we iteratively process partial resource instances from the worklist until the worklist is empty. Suppose we are processing resource instance r of resource R. We go through the dependencies of r.

We assume that the partial installation specification resolves inside dependencies of each resource instance in it by providing a resource instance on which r is inside-dependent (that is, the system does not generate new machines automatically). Given r, we check that there is an existing resource instance which matches the inside dependency. For each inside dependency, we create a directed edge labeled "*inside*" from the node in the graph that represents r to the node that represents its container.

Now consider environment dependencies of r. By assumption, the environment dependencies are a set of disjunctions. For each dependency in the set, we create a hyperedge with source r, and targets as follows. Suppose the dependency is a disjunction of resource keys k_1, ..., k_n, and consider the processing of key k_1. If we can already find a node r_1 in the graph with key k_1' such that $[\![k_1']\!] \leq_{RT} [\![k_1]\!]$ and such that r_1 is inside the same machine as r, we add the node r_1 as a target of the hyperedge. If there is no such node, we instantiate a new resource instance r_{new} with key k_1, inside the same machine as the machine of r, add the node for r_{new} to the graph as well as the worklist. (We add r_{new} to the worklist to ensure its dependencies are processed in the future.) At the end of this process, we construct a hyperedge with source r and n targets, one for each disjunct in the dependency, and label it "environment".

For peer dependencies of r, we proceed similarly. The only difference is that we look for a matching resource that is a subtype of the key, but need not be on the same machine. If we find such a node in the graph, we add it as the target of the hyperedge. If we do not find a matching resource, we add a new instance, but conservatively assume that the new instance resides in the same machine as r.

Figure 5 shows the dependencies generated when processing the partial instantiation specification of Figure 2. We have marked the resource instances that were present in the specification with a "✓".

LEMMA 1. *Let $\mathcal{R}$ be a set of well-formed resource types and $\mathcal{I}$ a partial install specification. Then procedure* GraphGen($\mathcal{R}, \mathcal{I}$) *creates a directed hypergraph $G = (V, E)$, such that: (i) for each resource instance $r \in \mathcal{I}$, we have $r \in V$, and for each resource instance $r \in V$, either $r \in \mathcal{I}$ or there is some resource instance $r' \in \mathcal{I}$ that is transitively dependent on the key of r; (ii) for each resource instance $r \in V$, if r.*Inside *is not* `null`*, then there is a resource instance $r' \in \mathcal{I}$ such that there is an inside edge from r to r'; (iii) for each resource key k such that there is an environment dependency from k to r, there is an hyperedge with source r containing a target resource instance with key k such that this resource instance is on the same machine as r; (iv) for each resource key k such that there is a peer dependency from k to r, there is an hyperedge with source r containing a target resource instance with key k.*

Intuitively, this lemma states that the generated graph correctly encodes the dependencies of all resource instances in $\mathcal{I}$.

Constraint Generation Given the directed hypergraph GraphGen($\mathcal{R}, \mathcal{I}$), we generate Boolean constraints as follows. An atomic proposition is of the form rsrc(id), where id is a resource instance identifier; it states that the resource instance with identifier id is installed in the machine obtained by following its inside dependencies.

We generate two kinds of constraints from the graph.

First, for each vertex in the constraint graph that represents a resource instance specified in the partial install specification, we add the constraint rsrc(m, id). This constraint ensures that each resource instance mentioned in the partial install specification is indeed instantiated in the deployment.

Second, we generate *dependency constraints* for each node v as follows. For each hyperedge e with source v and targets $\{v_1, \ldots, v_n\}$, we generate a constraint

$$\mathsf{rsrc}(v) \rightarrow \boxtimes\{\mathsf{rsrc}(v_1), \ldots, \mathsf{rsrc}(v_n)\} \quad (1)$$

where $\boxtimes S$ is the Boolean predicate that is true iff exactly one proposition from S is true. Formally,

$$\boxtimes S \equiv (\bigvee_{p \in S} p_i) \wedge \bigwedge_{p \in S} (p \rightarrow \bigwedge_{q \in S, q \neq p} \neg q)$$

We denote the conjunction of the set of all above predicates for a set of well-formed resource types $\mathcal{R}$ and a partial install specification $\mathcal{I}$ as Generate($\mathcal{R}, \mathcal{I}$).

THEOREM 1. *Let $\mathcal{R}$ be a set of well-formed resource types and $\mathcal{I}$ a partial install specification. There exists a full installation specification extending the partial install specification iff the formula* Generate($\mathcal{R}, \mathcal{I}$) *is satisfiable.*

A satisfying assignment to the Boolean constraints determines a full installation specification extending the partial installation specification. We can compute the values of all input, configuration, and output ports of all resource instances by a linear pass in topological order of dependencies, filling in the input ports of each resource instance based on the already-computed values of output ports.

5. Deployment

Engage's *deployment engine* takes a full installation specification and automatically deploys the application. It provides runtime support for provisioning servers, co-ordinating installations, as well

as generic monitoring, shutdown, and upgrade services for applications. The deployment engine uses *drivers* for each resource to co-ordinate its actions.

5.1 Resource Drivers

In addition to a type, a resource specifies a *driver* that implements a state machine for the lifecycle management of the instances of that type. A driver state machine consists of a set of *states* and a set of *transitions* between them. Each state machine has three special states, *uninstalled*, *inactive*, and *active*, called the *basic states*. Initially, a state machine resides in its *uninstalled*.

Each transition has a *guard* and an *action*. The guard of each action is either *true* or a conjunctive condition about the basic states of all the resource instances that the driver's associated resource instance depends on (i.e., the upstream resource instances) or the resource instances that depend on the driver's resource instance (i.e., the downstream resource instances). Basic state predicates are of the form $\uparrow s$ or $\downarrow s$, where $s \in \{uninstalled, inactive, active\}$. A basic state predicate $\uparrow s$ (respectively, $\downarrow s$) is true if the state machines for all upstream (respectively, downstream) resource instances are in state s. Each guarded action is implemented in an underlying programming language (Python in our implementation).

A resource instance starts its life in its initial state, *uninstalled*, and proceeds through various states in the state machine depending on transitions invoked by the deployment engine. If the guard of a transition is *true*, then the action can be performed at any time. Otherwise, the transition blocks until the guard becomes true, at which point the action is executed and the state is updated.

Formally, the driver for a resource type R is a state machine $(Q, uninstalled, inactive, active, \mathcal{A}, \delta)$ consisting of a set of *states* Q with $\{uninstalled, inactive, active\} \subseteq Q$, a set of *actions* $\mathcal{A}$, and a (partial) transition function $\delta : Q \times (BasicStatesPred \times \mathcal{A}) \rightharpoonup Q$, where *BasicStatesPred* is a conjunction of basic state predicates.

Figure 3 shows the state machine for the Tomcat server. The server starts in the initial *uninstalled* state. The action `install` takes it to the *inactive* state. From the *inactive* state, the action `start` takes it to the *active*, and has the precondition that all upstream dependencies are active. From the *active* state, the `stop` action takes it back to the *inactive* state and has the precondition that all downstream dependencies are inactive. In the *active* state, the server can be restarted. Each guarded action carries out system changes and commands required to install, start, or stop the server, as well as additional book-keeping.

5.2 Runtime Services

Provisioning The starting point of a deployment is the set of physical or virtual servers on which the application software will be installed. Engage provides a set of runtime tools to determine properties of servers, such as hostname, IP address, operating system, CPU architecture, etc. These tools automatically create a resource instance for the server, and in practice, are used to start writing a new partial installation specification when the servers are known. In addition, Engage provides integration with cloud servers through the *libcloud* APIs [3]. If a machine resource instance in the partial installation specification does not include configuration details, and Engage is being run in a cloud environment, a new virtual server is provisioned to perform the role of that machine in the deployment. In this case, the additional host configuration details (obtained from the cloud infrastructure) are added to the installation specification before passing it to the configuration engine. Engage currently integrates with Rackspace cloud services and Amazon AWS.

Installation, Monitoring, and Shutdown Given a full installation specification, the deployment engine executes commands on the resource drivers for each resource instance in the specification such that every driver state machine is in its active state. At this point, the system is defined to be deployed.

The deployment engine orders resource instances according to dependencies. Since the dependency ordering is acyclic, this is always possible. In the beginning, the resource driver for each resource instance is in its initial state. The deployment engine executes driver transitions to bring each driver to its active state. The process can be performed in parallel, as long as the dependency ordering is met, and the precondition for each transition is satisfied. Since the deployment engine tracks the states of each resource instance, it can co-ordinate the calls.

In the multi-host case, we must coordinate the install across multiple hosts. The implementation of a multi-host install can be simplified if one can partially order the machines such that for every two machines m_1 and m_2, m_1 is before m_2 if there is some resource instance to be installed in m_2 that depends on some resource instance in m_1 but no resource instance to be installed in m_2 depends on some instance in m_1. We have found this assumption to be valid for all the systems that we have modeled thus far, and our implementation makes this simplifying assumption. In this case, we can break the overall install specification into per-node specifications and run a slave instance of Engage on each target host. The entire deployment is then coordinated from a master host, with each slave running with no awareness of the others. Slave deployments can run in parallel when the slaves have no inter-dependencies.

The runtime includes a plugin framework for the automatic integration with monitoring and management tools. For example, Engage integrates with *monit* [4], a process monitoring/restart service. The runtime system adds an instance of monit to the installation specification for each target host, ensuring that monit itself will be installed. After the deployment completes, the resource type is used by the plugin to generate a monit configuration file registered with monit. Users can view the status and resource usage of each installed service. If the process associated with a service fails, it will be automatically restarted by monit using a set of runtime services provided by Engage.

Shutting down an application goes in the reverse dependency order, and applies transitions of each resource driver to bring each state machine to an inactive state. Again, the runtime co-ordinates this activity since it knows the dependencies and tracks the states of the resource drivers.

Upgrades Initial deployment of an application stack is only one part of its lifecycle. Changes after initial deployment may incude bug fixes, enhancements, infrastructure updates, security patches, and updates to Engage itself. Given that Engage has a full description of the deployed system, multiple upgrade strategies are possible.

Currently, we require that the user (or a utility program, as in the case of Django) provide a *partial install specification* describing the desired new state of the system. This is used to compute a full install specification for the deployed system. The current system is then backed up, and any components that will be removed or that cannot be upgraded in-place are uninstalled. The new system is now deployed, per the install specification, upgrading and adding components as needed. If the upgrade fails, the partially installed components are uninstalled and the old version restored from the backup. This upgrade approach is relatively easy to implement, supports rollback in the case of errors, and can handle changes to any layer of the stack, including Engage. However, all upgrades using this approach experience the worst case upgrade time, even if there are only minor differences between the old and new configurations. We leave optimizations of the upgrade framework as future work, possibly extending [11].

App name	Description	Source	Comments
Areneae	Simple test app	Beta tester	Deployed by a third party
Buzzfire	Twitter bookmark and ranking app	Open source	Uses Redis key-value store
Codespeed	Web application performance monitor	Open source	
Django-Blog	Blogging platform	Beta tester	Deployed by a third party; Installs 18 Python package dependencies
Django-CMS	Content Management System	Open source	
FA	Manage faculty, student, and postdoc applications	Beta tester	Application used in production
Feature Collector	Gather software feature requests	Developed internally	
WebApp	Run production web site for Django hosting company	Developed internally	Application used in production

Table 1: Django applications. "Beta testers" are independent application developers who used the service.

6. Evaluation

We have implemented the Engage system and we have used it in active deployment in providing platform-as-a-service support for Django applications. The front-end for the declarative framework and the configuration engine are written in about 6K lines of Ocaml. We use the the MiniSat satisfiability solver to solve Boolean constraints. The runtime system (provisioning and deployment engine) and resource drivers are written in about 26K lines of Python. In addition, we have about 5K lines of resource types in our resource library. We allow deployments to be performed either on a given set of servers or on dynamically provisioned servers from Rackspace or Amazon Web Services (AWS). The entire system, including the resource types, is open source and (for Django applications) available through a web service.

We have installed several open source applications, including OpenMRS (described in Section 2). In the following, we give two case studies that we have performed using the system. The first (JasperReports) was used as a case study to compare times for a manual install versus the time needed to automate an installation by developing the required resource types and drivers in Engage. The second (Django applications) describes our experience hosting third party applications on the cloud.

6.1 JasperReports

The Jasper Reports Server [5] provides a web user interface and web service APIs for generating, viewing, and managing reports. It runs in a servlet container (e.g., Apache Tomcat) and requires a database (e.g., MySQL). The installation guide is 77 pages long, with 8 pages dedicated to the specific approach we used for installing Jasper.

We compared the time for installing Jasper manually with the time to automate the installation of Jasper in Engage. The manual installations were performed by one of the authors, who had not attempted to install Jasper previously. It took him 5 hours the first time, 2 hours 15 minutes the second time, and converged to around 1 hour in subsequent tries. Much of the learning curve was due to imprecise instructions that led to incorrect actions (followed by debugging sessions and rolling back), environment issues, and locating/downloading the required prerequisite software.

Engage already included resources for Tomcat and MySQL. To automate Jasper installation, we created two new resources: one for Jasper Reports Server itself and one for a MySQL JDBC connector required as a dependency for Jasper. Specifying the resource type for the JDBC connector required 40 lines. No additional Python code was required for the driver as we were able to reuse existing generic driver code for downloading and extracting archives. The resource for Jasper was 69 lines of types and 201 lines of Python code (including comments).

The automated installation of Jasper Reports Server via Engage performs environment checks (e.g., required TCP/IP ports are available), downloads required application packages, installs the components in dependency order, and starts the database, web server, and reports server. After installation, the reports server can be managed (start, stop, status check) via Engage, or automatically integrated with 3rd party management tools.

The development time for Engage support of Jasper Reports Server was 3 hours 56 minutes. This time was broken down into 47 minutes for resource type design, 81 minutes for driver development, and 108 minutes for debugging and testing.

Running the automated install of Jasper Reports Server takes 17 minutes if the required software packages are downloaded from the internet and 5 minutes if they are obtained from a local file cache. The automation of the Jasper install via Engage ensures a repeatable process, supports both local and cloud-based targets, and can be used as a part of a larger deployment (e.g., where the reports server can be used as a service to a web application). The partial installation specification for Jasper is 26 lines long. The full installation specification generated by the configuration engine is 434 lines long.

6.2 Django Applications

We implemented resources and drivers for installation and management of applications developed in Django [6], a web application framework written in Python. A goal of our Django support was to enable Django developers to deploy their existing applications on different environments (local test environments and production environments on the cloud) with little changes and no need to understand the internals of Engage. To achieve this, we built an *application packager* that validates a Django application, extracts some metadata used by Engage, and packages the application into an archive with a pre-defined layout. This application can then be deployed by Engage to the cloud or a local machine. We provide pre-defined partial installation specifications for the same application to be deployed in different configurations (e.g., debug or production, local or cloud), supporting the migration of changes through the full development lifecycle: from development to QA to staging to production.

Django support involves 37 resources, of which 14 are specific to Django applications. Engage allows the following (independent) configuration choices for Django applications:

- **OS**: MacOSX (two versions) or Ubuntu Linux (two versions)
- **Web server**: Gunicorn or Apache HTTP server
- **Database**: SQLite or MySQL
- **Optional components**: RabbitMQ/Celery (message queuing), Redis (key-value store), and memcached (caching)
- **Optional monitoring**: Monit

Thus, we currently support 256 distinct deployment configurations on a single node. Furthermore, many topologies of these components are permitted by the Engage architecture: everything deployed to the same node, a separate node for each component, multiple application nodes, etc. Finally, our Django driver supports the declarative enumeration of Python packages available from the Python package index (http://pypi.python.org) to be installed as a prerequisite to the Django application.

Evaluating installs We have evaluated Engage's Django support by testing it with Django applications developed by us, obtained as open source applications, and provided by third party beta testers. Table 1 summarizes eight of the applications we deployed. All eight applications were deployable by Engage without requiring any application-specific deployment code. Areneae and Django-Blog are particularly interesting, as they were developed and deployed by beta customers (who had no prior contact with the paper authors), using only our documentation and a few email exchanges. WebApp is the production infrastructure that runs a commercial PaaS (Platform as a Service) site. It is about 4K lines of code, and uses asynchronous messaging, cron jobs, and caching. The partial installation specification for the production site is 61 lines long and has seven resources. The full installation specification generated by the Engage configuration engine from this spec is 1,444 lines long and has 29 resources.

Evaluating upgrades To evaluate application upgrades, we took two snapshots of the FA application about four months apart. Both snapshots represented production versions of the application. Between the two snapshots, the user interface, application logic, and database schema all changed. We use South, a *database migration* framework, in the Engage Django driver to support application upgrades involving database schema changes. Using South, we were able to automatically upgrade from the old version to the new version of the application, while preserving the content in the database. If we introduce an error in the second application version that causes the upgrade to fail, Engage automatically rolls back to the prior application version.

7. Conclusion

We have presented Engage, a novel system for deployment of complex multi-component, distributed applications. Our contribution in this paper is twofold. First, we introduce a domain-specific language that provides the necessary syntax to impose a set of common structural dependencies between a set of inter-dependent resources succinctly and explicitly. Second, we present a tool suite that exploits the language: (i) to provide effective static checks to identify specification flaws and incompatibilities; (ii) to identify a full installation specification of a system from an input partial installation specification; (iii) to carry out an installation specification automatically; and (iv) to manage the life cycles of installed resources automatically.

References

[1] Puppet at Puppet Labs: http://www.puppetlabs.com/.

[2] Chef at Opscode: http://www.opscode.com/chef/.

[3] Apache Libcloud: http://libcloud.apache.org/.

[4] Monit: http://mmonit.com/monit/.

[5] JasperReports Server: http://www.jaspersoft.com/reporting-server/.

[6] Django: https://www.djangoproject.com/.

[7] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *ECOOP '02*, pages 334–367. Springer, 2002.

[8] P. Anderson and A. Scobie. LCFG: the next generation. In *UKUUG Winter Conference. UKUUG*, 2002.

[9] M. Burgess. A site configuration engine. *Computing Systems*, 8(2):309–337, 1995.

[10] A. Cicchetti, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchirol. A model driven approach to upgrade package based software systems. In *Evaluation of Novel Approaches to Software Engineering*, pages 262–276. Springer, 2010.

[11] R. Di Cosmo, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Supporting software evolution in component-based FOSS systems. *Science of Computer Programming*, 2010.

[12] R. Di Cosmo and J. Vouillon. On software component co-installability. In *SIGSOFT FSE '11*, pages 256–266. ACM, 2011.

[13] R. Di Cosmo, P. Trezentos, and S. Zacchiroli. Package upgrades in FOSS distributions: Details and challenges. In *HotSWup '08*, 2008.

[14] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI '98*, pages 236–248. ACM, 1998.

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03*, pages 1–11. ACM, 2003.

[16] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog configuration management framework. *Operating Systems Review*, 43(1):16–25, 2009.

[17] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE '06*, pages 199–208. IEEE Computer Society, 2006.

[18] P. Trezentos, I. Lynce, and A. L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-Boolean optimization. In *ASE '10*, pages 427–436. ACM, 2010.

[19] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *ICSE '07*, pages 178–188. IEEE Computer Society, 2007.

[20] B.A. Wolfe, B.W. Mamlin, P.G. Biondich, H. Fraser, D. Jazayeri, C. Allen, J. Miranda, and W.M. Tierney. Cooking Up an Open Source EMR for Developing Countries: OpenMRS — A Recipe for Successful Collaboration. In *American Medical Informatics Association Annual Symposium*, pages 529 – 533. AMIA, 2006.