Engage Architecture Guide

Revision History

Date	**Author**	**Description**
2011-09-02	jfischer	Created from internal design docs.

Contents

Notice

1 Introduction

Engage is an open source platform for installing, configuring, managing, and maintaining application software stacks. This document describes the Engage platform and the declarative language it uses to describe software components and their dependencies.

1.1 Background

Modern software is frequently built from collections of independently-developed but inter-related components (e.g. packages, libraries, services). This provides re-usability benefits, but complicates the installation, configuration, and management of software systems. In particular, one must 1) select a set of software components which are compatible with each other and satisfy all required dependencies, 2) install these components in an order that ensures that all installation dependencies are met, and 3) configure each individual component to connect properly to its dependent components.

The unique technology behind the Engage platform is 1) a representation for software component inter-dependencies, and 2) a method for using this representation to coordinate the installation and configuration of software components. The representation captures each component's dependency requirement (the allowable sets of components which satisfy its dependencies), compatibility requirements (e.g. a component may only be compatible with specific versions of another component), the physical relationships (e.g. nesting, software linking, and independent services), and configuration dependencies (a given configuration variable in one component depends on the value of a configuration variable in another component). The dependency representation (metadata) can be stored in files or a database.

Given a collection of software components to be installed (called an *install specification*), Engage uses the dependency metadata to determine a (potentially larger) set of software components to be installed which satisfies all dependency requirements. It also specifies the configuration variables for each component, based on user input and any configuration variable dependencies between components. In addition, Engage proscribes an ordering for executing the installation, configuration, and startup operations on each component, taking into account all dependencies between components for these operations (e.g. an program may need to be installed only after any libraries that it uses).

1.2 Glossary

Here, we define any specific terms that we will use in the rest of the document.

Configuration Engine
This is the backend component which, given a library of r*esource definitions* and a user-specified i*nstall specification*, finds an *install solution* that satisfies the specification and any constraints in the resource definitions. The install solution can then be passed to the *deployment engine* to perform the actual install. If no solution is possible, the configuration engine returns an error.

Dependencies
We define three types of dependencies between resources (software packages as well as the OS/hardware):

Inside -- A software package A has an *inside dependency* on package B if it must be installed/run inside of package B. Examples include OpenMRS inside Tomcat, and Tomcat inside a particular OS.
Environment -- A software package A has an *environment dependency* on package B if it depends on package B being installed in an enclosing resource. As an example, Tomcat has an environment dependency on Java. It is not installed inside of Java, but requires that Java be installed on the machine.
Peer -- A software package A has a *peer dependency* on package B if it calls B through some kind of interprocess communication mechanism (e.g. Web Services or TCP/IP). For example, OpenMRS has a peer dependency on MySQL.

Driver
A driver is the combined metadata and code needed to install an application component. This includes a *resource definition*, one or more *package locators*, and a *resource manager*.

Deployment Engine
This is the backend component which, given an *install solution* (a set of configured resource instances), builds a global install script which will call the install scripts of the individual software packages and start any required services, all in dependency order. This code should also manage recover and rollback from errors.

Install Solution
A description of how a collection of software packages (*resources*) should look when installed and configured. It must satisfy a particular *install specification* and the constraints defined by a library of *resource definitions*. The install solution is represented as a collection of *resource instances*.

Install Specification
The install specification is created by the user to indicate the desired software configuration. This is done by providing a JSON file or through a command line installer. The specification is then passed to a solver which computes a complete configuration (resolving all dependencies) and then creates a global *install solution*.

Package Locator
A package locator is a metadata entry indicating where the software for a resource (application component) may be obtained. A package locator typically references a local file or a URL for downloading the software. It may also be a package name used by a package manager (e.g. apt or macports).

Resource
A *resource* is a generic term for a software package, the OS, a virtual machine, or a machine.

Resource Definition
A *resource definition* specifies how a resource may be instantiated. It includes a name, version, names and types of configuration properties, and constraints on dependent resources.

Resource Instance
The metadata describing how a specific application component (resource) will be / has been installed.. A resource instance includes an id, name, version number, configuration properties, and links to specific dependent resources. Once can consider resource definitions to be like classes and resource instances to be like objects.

Resource Package Library
The resource package library lists a set of *package locators* for each resource definition. These package locators indicate where the actual software may be obtained for the application component. This may be a file on the local machine or a remote website.

2 System Architecture

Figure 1 shows the system architecture of Engage. Specific terms are defined in the Glossary of the previous section.

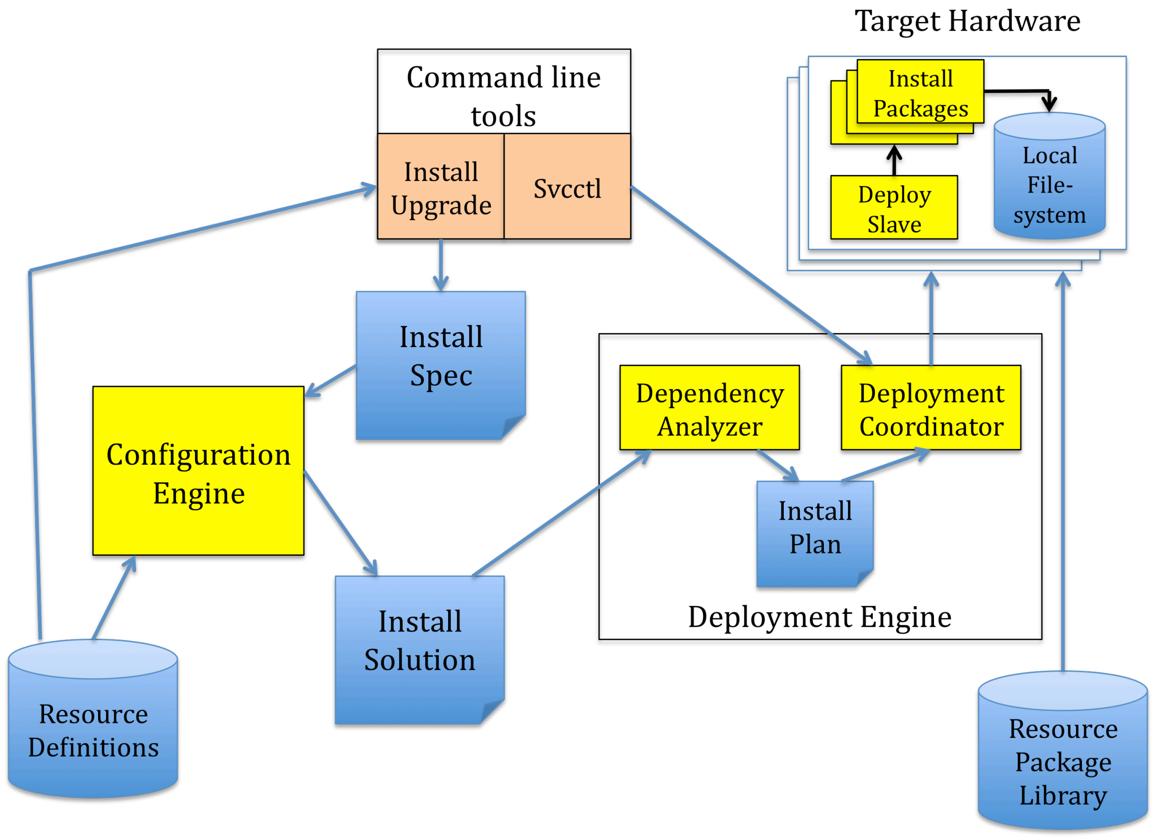

Figure 1: System Architecture of Engage

To explain the details of the Engage's architecture, we will describe it from the perspectives of a developer and then an user.

Developer Perspective
To add support for an application to Engage, one first creates a *driver* for each software component used by the application. Drivers are used to implement the installation, configuration, upgrade, and management of each component in isolation. Each driver includes a *resource definition, package locators,* and a *resource manager* class. Resource definitions are JSON descriptions of a component and are used to determine the valid component selections for the overall application, the settings that need to be supplied by the user (and their default values), and the relationships of between the settings of different components (e.g. for a service provided by one component and consumed by another). Package locators tell the system where to find the actual software to be installed (e.g. from the local filesystem, from a remote website, or via a package manager). The resource manager is a Python class with methods to install, uninstall, start, stop, and upgrade the component. Each driver can be developed independently and reused across applications and environments. It is expected that, as the library of drivers for Engage expands, adding support for a new application will largely involve the reuse of existing drivers.

User Perspective
A user of Engage first starts the install command line tool and selects the configuration they wish to install from a list. Any required configuration values are then requested from the user. The installer next calls the *configuration engine*. This engine uses the user-provided configuration values and the metadata in the resource definitions to determine a set of resources to be installed and their configuration values. This result, called the *install solution*, must satisfy all dependency constraints and configure the individual components so that they can work together (e.g. a database client must be configured to talk to its corresponding database server). The install solution is represented as a list of *resource instances*. Each resource instance defines a unique id, the component's associated driver, configuration values, and dependencies on other resource instances.

The install solution is passed to the *deployment engine*, which is responsible for coordinating the installation. The deployment engine coordinates, through *deployment slaves* running on each of the target machines, the execution of the drivers for each component. Each driver is provided with the configuration settings/dependencies for that component (extracted from the install solution). The deployment engine calls these drivers in dependency order. The driver's *resource manager* class is responsible for determining the source of the software package from the *resource package library*, downloading it (if necessary), installing the software, and finally configuring it. After a component is installed, the deployment engine verifies that the installation was successful and starts the component (if it involves a standalone process) by calling the appropriate methods on the associated resource manager.

At the end of this process, the user should have a full application installed, configured and running, potentially across multiple machines.

This same infrastructure is used to implement upgrade, backup, and uninstall functionality. The user can also perform basic application management through the *svcctl* command line tool. This tool leverages the application metadata to support dependency aware startup, shutdown, and monitoring of the application and its components. Finally, Engage can integrate with system management tools and automatically provide the required metadata for such tools to monitor and manage a deployed application.

3 Installation Constraints: Theory

3.1 Background

At the core of the Engage is a declarative framework for representing component dependencies. Its starting point is the simple, Boolean dependency declarations used in [Mancinelli06, Tucker07], which specify Boolean dependency relations between components. Essentially, the language of constraints can specify properties of the form *in order for package* a *to be installed, either* b *and* c *must be installed, or* e *and* f *must be installed and* g *must not be installed*, i.e., the Boolean constraint: $a \rightarrow (b \wedge c \vee e \wedge f \wedge \neg g)$.

For example, suppose that (ignoring version numbers) in order to install Apache, the system must already have libc6 and perl, have either debconf or debconf-2.0, but not have jserv. This gives the constraint: apache1.3.34-2 → libc6 ∧ perl ∧ (debconf ∨ debconf-2.0) ∧ ¬jserv.

An *install solution* is a subset of software packages. An install solution *P* is *valid* if every Boolean dependency constraint on packages in *P* is true. Given a set of Boolean dependency constraints on software packages, and a set *P* of target packages to install, the *install problem* asks what set of packages *P'* must be installed such that (1) $P \subseteq P'$ (i.e., every target package is installed), and (2) *P'* is a valid installation profile. For Boolean constraints, the install problem reduces to Boolean satisfiability: intuitively, given a set of target packages *P*, and a set of dependency constraints *C* on packages, any satisfying assignment to the Boolean formula: $(\bigwedge_{p \in P} p) \wedge (\wedge\{c \mid c \in C\})$ is a valid installation profile. This idea has been implemented in several tools (debcheck [Mancinelli06] and Opium [Tucker07]).

Unfortunately, the Boolean view is somewhat restrictive for full application installation. For example, we cannot express configuration information for packages, such as the TCP/IP port number for a service, and we cannot express relationships such as the value of a port used by a client to connect to the server is the same as the port for the server. Second, existing install tools based on the Boolean view work for *single* machines. For installations spanning multiple machines, the constraint generation and solving does not take into account physical context or sharing. However, this is crucial for large-scale cluster or cloud deployments.

3.2 Resource definitions and instances

To address these limitations, we extend the basic Boolean constrain model. Our vehicle for this extension is the *resource definition library*. Each resource definition corresponds to a software package and contains properties uniquely identifying the software component (called its *key*), definitions of configuration properties, and dependency constraints. A resource key is a set of key/value pairs that identify a specific package, such as package name and version.

A *resource instance* is creating from a resource definition by assigning concrete values to its configuration properties and replacing dependency constraints with directional links to other resource instances, such that the associated resource definitions for those instances satisfy the original constraints. Each resource instance contains a *resource id*, which is an arbitrary string uniquely identifying the resource and a key, which refers to the associated resource definition. Given a set *P* of resource instances, solving the install problem yields a set *P′* of resource instances which satisfy the dependency constraints of the corresponding resource definitions. This set is known as an *install solution*.

3.3 Ports

The configuration property definitions are divided into three groups, which are called *ports*:

- A single *configuration port* for configuration properties which should be specified/overridden by the user and are used by the associated install package.

- Zero or more *input ports* whose values come from dependent software components and are used by the associated install package.
- Zero or more *output ports* which provide values for any client software components which depend on the component being defined.

Input ports are associated with output ports of other install components through dependency constraints. These constraints reference other component definitions, potentially combined through logical *and* and *exclusive-or* (one-of) operators. When the configuration engine resolves a given dependency constraint to a specific resource definition, it then maps the associated output ports of the dependent component to the input ports of the client component.

3.4 Constraint types and semantics

The constraints in a resource definition are a Boolean expression over *atomic constraints*. Each atomic constraint is a pair of the form *(target resource, constraint type)*. The target resource references a specific resource definition's key. The constraint type indicates the semantics of the constraint and may introduce additional *side constraints* (see below). The atomic constraint thus represents a dependency of the specified type on the target resource. In the current implementation, three constraint types are modeled:

- *Environment* constraints represent the dependency on another software package in the same machine. This type of constraint is typically used for libraries. For example, in OpenMRS [OpenMRS], the dependencies of Tomcat and OpenMRS itself on Java are represented as environment constraints.
- *Inside* constraints represent that a package will be installed inside of another package. For example, the dependency of OpenMRS on the Tomcat application server is represented as an inside constraint. One may represent physical and virtual machines as resources and any machine/OS dependencies as inside constraints.
- *Peer* constraints represent a dependency on another software package that is not necessarily on the same machine, typically connected via a network protocol. For example, the dependency of OpenMRS on a MySQL database is represented as a peer constraint.

Other constraint types are possible.

A satisfying solution to the Boolean constraints for a collection of resources can be interpreted as follows: each atomic constraint which is assigned *true* represents a dependency of the specified type between two resource instances and requires that, if the source resource instance is present in the install solution, the target resource instance is also present. Each atomic constraint that is assigned *false* represents a dependency that is not in the install solution.

3.5 Side constraints

Each constraint type may optionally introduce side constraints into the set of constraints implied by the source resource. The atomic predicates of these constraints may be either 1) variables representing resource instances, where r is *true* if resource r is present in the install solution, or 2) predicates of the form $c(r_1, r_2)$, where $c(r_1, r_2)$ is *true* iff there exists a constraint of type c from resource instance r_1 to instance r_2, or 3) predicates of the form $c^*(r_1, r_2)$, where $c^*()$ is the transitive closure of the $c()$ predicate. The following side constraints are introduced by the three constraint types described above:

- If $env(r_1, r_2)$ (there is an environment constraint from resource r_1 to resource r_2), then there exists an r_3 such that $inside^*(r_1, r_3)$ and $inside^*(r_2, r_3)$.
- If $inside(r_1, r_2)$ (there is an inside constraint from resource r_1 to resource r_2) and $inside(r_1, r_3)$, then $r_2 = r_3$. In other words, a resource can have at most one inside constraint.
- Peer constraints do not introduce any side constraints.

3.6 Resource graphs

A *resource definition graph* is a directed graph obtained by mapping each resource definition in a set to a vertex and then creating directed edges between the vertices for each atomic constraint of each resource definition. The direction of an edge follows the direction of the dependency: the source vertex corresponds to the resource that has the constraint. We say that a set of resource definitions is *acyclic* if the associated resource definition graph is acyclic.

A *resource instance graph* is a directed graph obtained by mapping each resource instance in a set to a vertex and then creating a directed edge between vertices when there exists a dependency between the two associated resource instances. The direction of the edge follows the direction of the dependency: the source vertex corresponds to the resource that has the dependency. We call the resource instance graph of an install solution an *install graph*. We say that a set of resource instances is *acyclic* if the associated resource instance graph is acyclic. If a set of resource definitions is acyclic, then all install solutions that satisfy the install problem for the resource definition set are acyclic.

3.7 Partial orders on install solutions

Given a set of resource instances corresponding to an install solution, one can use the resource dependencies to determine valid operation sequences over the resource instance set for such operations as installation, configuration, patch, uninstall, startup, and shutdown. Given the three constraint types described above (environment, inside, and peer), it is sufficient to use a *dependency partial order* – a partial order obtained by traversing the install graph starting from the nodes with no outgoing arcs and traversing all arcs in the reverse direction. This is a *topological sort*, for which efficient algorithms exist (e.g. [Kahn62]). Install, configuration, patch, and startup can be accomplished by executing the associated operation on each resource in the order specified by the pre-order. Uninstall and shutdown can be accomplished by executing the associated operation on each resource in the reverse of the partial order.

Consider, for example, three resources: *OpenMRS*, *Tomcat*, and *MySQL*, where *inside(OpenMRS, Tomcat)* and *peer(OpenMRS, MySQL)*. Valid dependency preorders for these resources are (MySQL, Tomcat, OpenMRS) and (Tomcat, MySQL, and OpenMRS). It is feasible to install the components in the order (MySQL, Tomcat, OpenMRS). The installation sequence (MySQL, OpenMRS, Tomcat) is not legal. For shutdown and uninstall, can use the sequences (OpenMRS, Tomcat, MySQL) and (OpenMRS, MySQL, Tomcat).

One can generalize this approach to allow each constraint type to be treated differently. For example, for the startup operation, environment constraints do not need to be considered in the preorder. If *env*(r_1, r_2) then both (r_1, r_2) and (r_2, r_1) are valid startup sequences. One could also imagine a constraint type *c* where, if $c(r_1, r_2)$, then only (r_1, r_2) is valid for installation, but only (r_2, r_1) is valid for startup.

More formally, for each resource operation O, one can define a partial order function $\Phi_O(c, r_1, r_2)$, which is passed each constraint relationship $c(r, r')$ in the resource graph. The function Φ_O returns either:

- (r_1, r_2) if resource r_1 must appear before resource r_2 in the partial order for operation *O*
- (r_2, r_1) if resource r_2 must appear before resource r_1 in the partial order for operation *O*
- $\varnothing$ if there are no restrictions in the relative order of resources r_1 and r_2 in the partial order for operation *O*

4 Resource Definitions and Instances

We now look at a specific representation for resource definitions and instances. We use a representations based on JSON (JavaScript Object Notation [JSON]). The basic elements of the grammars include:

- Scalar constant values including strings ("this is a string") and integers (23).
- Maps, which are enclosed in braces and include a comma-separated list of property names strings and property values. For example: { foo: 5, bar: "str" }
- Lists, which are enclosed in brackets and include a comma-separated list of values. For example: [1, 2, "string"]

Resource definitions describe the valid configuration properties of software packages and their dependencies. These definitions are kept in a *resource definition library* and provided as input to the configuration engine.

Resource instances are used in two ways:

- A collection of *partial* resource instances is created by the configuration GUI and is used to provide an *install specification*. This specification tells (at a high level) what the user wants to install. It is partial in the sense that it does not necessarily provide all configuration values or resolve all dependencies. It is up to the constraint solver to complete the resource instances.
- An *install solution* is a collection of *complete* resource instances, for which all configuration values are specified and all constraints are resolved. In our implementation, the install solution is created by the constraint solver and given to the installer runtime.

See Section 7 for the full grammar of resource definitions and instances and Section 8 for a complete example of the resource definitions and install specification for OpenMRS.

4.1 Resource Definitions

A resource definition contains the following elements:

1. Key – the key is a collection of properties used to uniquely identify the resources. Typically, the key may contain a package name and version number. However, the set of properties is arbitrary, as long as the key uniquely identifies the resource definition. Here is an example key for an Apache Tomcat resource definition:
   { "name": "apache-tomcat", "version": "6.0.16" }

2. Configuration port – the configuration port defines a collection of properties whose values must be provided by the user in order to properly configure the software package.

3. Input ports – an input port is a collection of property definitions whose values are provided by a corresponding output port in a dependent resource. An input port definition can also provide values to be included in a list property defined by the output port (see Section 4.1.1 for details). For an input port to be *compatible* with an output port, each required property of the input port must have a corresponding property of the same type on the output port.

4. Output ports – an output port defines a collection of properties to be made available to other resources. Note that input-output port pairings are many-to-one: a given output port may be associated with the input ports of multiple resources. This is consistent with most software package and service dependencies.

5. Inside constraint – this constrains the nesting of the resource in other resources. It may be an atomic constraint or a "one of" constraint. Thus, an inside constraint will resolve to exactly one "outer" resource. Details of constraints may be found in Section 3.1.2. As an example of an inside constraint, OpenMRS must be instantiated inside of Tomcat, where the Tomcat version is from 6.0 to 6.0.16.

6. Environment constraints – this describes a collection of dependent resources which must be present in the "environment", where the environment is one of the containing resources. An environment constraint can be a single resource constraint, a "one-of" constraint, or an "all-of" constraint.

7. Peer constraints – this describes a collection of resources which must be instantiated, not necessarily in the containing environment. Peer constraints are typically used for "services", such as a database, a web server, etc. A peer constraint can be a single resource constraint, a "one-of" constraint, or an "all-of" constraint.

4.1.1 Port Definitions

Port definitions define properties, their data types, and, optionally, default values for these properties. A port definition is a map whose keys are the property names. Values may be either scalar data types or maps.

Data types may be scalar types (e.g. string, integer, tcp_port, password, etc.), property maps, or property lists. Properties may have default values or values derived from other properties, according to the following rules (c.f. the grammar rule RD_TYPE_PROP in Section 7):

1. Configuration port properties may have default values, specified by the *default* keyword. These default values may be JSON constant values or *template strings* (see below). These values may be overridden by the user in the configuration wizard GUI.
2. Output port property definitions can specify a value for the property via the *fixed-value* keyword. These fixed values can be JSON constants or template strings.
3. Configuration and output port properties can derive their values from other properties via the *source* keyword, which is used to specify another property name. Output port properties can references configuration and input port properties. Configuration port properties can reference input port properties, except those input properties which use the *includes* keyword. The source properties are prefixed with their port type and, for input/output ports, their port name. For example: config_port.install_user, input_ports.host.hostname.
4. The default values for configuration port properties and fixed values for output port properties may be *string templates*: strings that include references to other property values. Property references are prefixed by the characters "${" and suffixed by the character "}". For example: "/Users/${config_port.install_user}/tomcat". The configuration engine will replace the property reference with the associated property value, once all dependent resource instances have been fully defined. . Output port string templates can references configuration and input port properties. Configuration port properties can reference input port string templates, except those input properties which use the *includes* keyword.
5. Given a resource r_o with an output port P_o and a resource r_i with a matching input port P_i (has all the same property names and types), if the output port P_o has a property p whose type is a list, then input port P_i can define p with the *includes* keyword. This indicates that, if resource r_o is linked as a dependency to resource r_i, the value for p specified in the input port definition should be added the list in the output definition. When multiple resources link to r_o as dependencies, their *includes* values are all added to the value of the list p, in an arbitrary order. The constant values used in an input property's *includes* reference may use string templates that reference values in the same resource's configuration port. For an example see the OpenMRS definitions of Section 8.1: the "environment-vars" output property of the "apache-tomcat" resource obtains its value from the "environment-vars" input property of the "OpenMRS" resource.

4.1.2 Constraint definitions

Peer and environment constraints may be *compound constraints* (c.f. the COMPOUD_CONSTRAINT grammar rule), while inside constraints are limited to *choice constraints* (c.f. the CHOICE_CONSTRAINT grammar rule), as they can only reference a single resource when resolved in the associated resource instance. Choice constraints can be atomic constraints or "one_of" (exclusive-or) expressions over atomic constraints. Compound constraints may be choice constraints or "all-of" (and) expressions over choice constraints.

Note that our concrete notation for constraints is more limited than the arbitrary Boolean expressions

described in the previous section. The concrete notation can be easily extended, if necessary.
Atomic resource constraints contain the following two elements:

1. *Key constraints* restrict resources based on the properties in their keys. Each key property can be restricted to a constant values or a range of constant values (c.f. the KEY_CONSTRAINT grammar rule.) This is useful for specifying a range of resources, which differ by version number. It can be interpreted as an "or" constraint over the specified resources.

2. Port mappings associate input ports of the resource being defined with output ports of the dependent resources. Each port mapping has the form INPUT_PORT: OUTPUT_PORT.

4.1.3 Well-formed Resource Definitions

An individual resource definition is *well-formed* if the following are all true:

1. The constraints, when taken together should, for every satisfiable instantiation, define exactly one mapping for each of the resource's input ports. With the limited constraint notation we will use in the first version, this implies that each resource referenced in a one_of constraint must define the same set of ports.
2. All configuration properties defined by output ports have values, either from a constant value, an associated input/config port property, or through a string template that references valid properties.

We also need to look at the collection of resource definitions as a whole. We say that a resource definition **library** is *well-formed* if the following are all true:

1. The set of inside and environment constraints do not form a cycle and the set of peer constraints do not form a cycle. These ensure that all resource instance graphs will be acyclic.
2. For each constraint, all the matching resource definitions in the library have output ports with the names specified in the port mapping and are compatible with the corresponding input ports mapped by the constraint.

4.2 Resource instances

Resource instances contain the following elements:

- Id – this is a string that uniquely identifies the resource instance.
- Key – the key of the resource definition of which this in an instance.
- Configuration port – lists the values of the configuration port properties.
- Input ports – lists the values of each input port's properties.
- Output ports – lists the values of each output port's properties.
- Inside – a single resolved constraint value (c.f. the RESOLVED_CONSTRAINT grammar rule and Section 4.2.1) corresponding to the resource instance containing this resource.
- Environment – a list of resolved constraint values corresponding to this resource's environment dependencies.
- Peer – a list of resolved constraint values corresponding to this resource's peer dependencies.

A partial resource instance used in an install specification must contain an id and key. It may also contain the configuration port, inside, environment, and peer elements. A complete resource instance used in an install solution must contain all elements. Complete resource instances should contain enough information to instantiate the resource without referring to the corresponding resource definition.

4.2.1 Resolved constraint values

A resolved constraint value contains the following elements:

- Id – the id of the dependent resource instance
- Key – the key of the dependent resource instance
- Port mapping – map of ports from the enclosing resource's input ports to the dependent resource's output ports.

4.2.2 Well-formed install specifications

The resource instances that make up an install specification must satisfy the following requirements:

1. Any resource with an inside constraint must have that constraint mapped to a specific resource.

2. Any peer constraints must be mapped to specific resources.
3. Only one instance of a given resource key may appear (transitively) inside a single machine resource. This is a temporary limitation to simplify our implementation.

4.3 Deferred features

The following capabilities would be nice to support in our resource definition language, but left for a future release.

4.3.1 Nested resource constraints

Selecting a given dependent resource via a constraint would require a third resource, either inside the defining resource or the dependent resource. This would be useful for defining “connectors” between two resources. We would need a way of expressing that a resource must be “inside” of another dependent resource.

4.3.2 Pattern matching on existential constraints

When an input port defines an *includes* property, this is interpreted as an existential constraint on the corresponding output port’s list property. Currently, the value must resolve to a constant (either a literal value or a property reference to the resource’s configuration port). One could imagine allowing constants and variables in these constraints, where constants are used to find a matching list item in the output port and variables are then bound to the corresponding values defined by the output port.

Consider, for example the following input port definition:

```
    “server” : {
     “services”: {“type”: [{ name: string, protocol : string, port : tcp_port }],
                  “match”:
                   [ { “name”: “${config_port.service_name}”, “protocol:” “http/1.1”,
                       “port”: “${config_port.service_port_no}” } ]}
   }
```

When an output port is mapped to this input port, the solver looks for an entry in the services list which contains a protocol property whose value is “http/1.1”. This entry is then selected and the variables service_name and service_port_no bound to the values of the name and port properties of this entry.

4.3.3 Smart version comparisons

Each software package tends to define its own version number scheme. When writing key constraints, we’d like to be able to set bounds on versions. For example: “6.0” <= version <= “6.0.16”. This would require some kind of plugin framework to establish the ordering relation for each package’s version numbers. We could also adopt a standardized version numbering scheme. See, for example the Debian version numbering conventions [DebVer].

For now, we’ll just use lexical ordering to compare versions.

5 Configuration Engine

The configuration engine (a.k.a. constraint solver) takes as its input a collection of resource definitions and an install specification (collection of partial resource instances). Its output is an install solution, which is a collection of complete resource instances. The solution has the following properties:

- No reference is made to resource instances outside of the set.
- All inside, environment, and peer constraints are resolved.
- Each input port is mapped to exactly one output port.
- The existential constraints implied by input ports are resolved by the corresponding output ports. In other words, if an input port defines a list element, that element is present in the corresponding output port.

5.1 Minimal solutions and shared dependencies

In addition to the correctness requirements above, we would like install solutions to be "mimimal" in the sense that they do not install more software than necessary. In particular, packages installed on the same machine should share common dependencies. For example, both Tomcat and Ant depend on Java, either the JDK (Java Developer Kit) or the JRE (Java Runtime Environment). If both Tomcat and Ant are installed on the same machine, they should both use the same Java install.

Defining a minimal install in not straightforward. For example, consider a package A that depends on package X and a package B that depends on either packages X and Y or on package Z. If we choose to have package B share package A's install of X, then we have to install Y as well, which could be very large.

Also, forced sharing may cause constraints to become unsolvable. Consider packages A, B, and C, where package A depends on resource X, package B depends on one of X or Y, and package C depends on Y. If we introduce sharing constraints, we'll get a conflict, as package B cannot share its dependencies with both A and C.

Longer term, we may want to use pseudo-Boolean constraints and a cost function based on the number of packages installed.

For the first release, we'll limit environment and peer constraints to conjunctions of "one_of" constraints and inside constraints to one_of constraints. We'll use the following heuristic to promote sharing: if a resource is not referenced in the install spec, and all the resources depending on it have other potential targets, that resource is removed from consideration. This is implemented while building the resource instance graph.

5.2 Resolving instances for a resource type

It turns out that, given the resource definition language we've specified for the first release, only one instance of a given resource type (unique key) can be configured per machine. This is because input ports cannot place any restrictions on the configuration of an output port, only existential (additive) constraints. Thus, once a resource type is selected for an environment or inside dependency, the instance can be shared with all other resources on the machine requiring the same resource type.

For peer dependencies, we will require that the install specification explicitly maps the input port to a specific resource instance's output port. This will live on a particular machine and thus can resolve to only one instance.

5.3 Solver design

The solver computes a solution in four steps:

1. Build a *resource instance* graph.
2. Generate Boolean constraints from the graph.
3. Find a solution to the constraints and prune the graph to include only the selected instances.

4. Compute values for all configuration properties of the selected resource instances. First, ask the user to provide values for any unspecified configuration port properties. Then, compute the values of all input and output port properties. Each output property is a function of input port properties and configuration port properties. Input properties take their values from the corresponding output ports of dependent resources. The calculation of properties will terminate, as the resource instance graph is acyclic.

5.3.1 Resource instance graph

The resource instance graph is a Directed Acyclic Graph where vertices represent resource instances and edges represent dependencies. Vertices are labeled with a pair (m, k), where m is a machine id and k a resource key. Edges are labeled with a triple (d, i_g, i_c), where d is a dependency type (inside, environment, peer), i_g is a group index (the "one_of" group within an all_of constraint), and i_c is a constraint index (the index of a constraint within a one_of group). The indices start at 0. An edge from vertex (m_1, k_1) to vertex (m_2, k_2) represents a potential dependency of the resource instance represented by (m_1, k_1) on the resource instance represented by (m_2, k_2).

5.3.1.1 Example

As an example, consider the following (partial) resource definitions:

```
{
 "key": {"name": "mac-osx", "version": "10.5.1"}
 …
}

{
  "key": {"name": "windows-xp", "version": "sp2"}
  …
}

{
 "key": {"name": "java-runtime-environment", "version": "1.5.0_16-133"},
 …
 "inside": {
   "one_of": [{"name": "mac-osx", "version": {"greater-than-or-equal":"10.5.0"}},
              {"name": "windows-xp", "version":{"greater-than-or-equal": "sp2"}}]
  }
}
```

If we create an instance for each resource definition, all residing on machine M1, the associated resource instance graph should have three vertices, labeled as follows:

1. (M1, {"name": "mac-osx", "version": "10.5.1"})
2. (M1, {"name": "windows-xp", "version": "sp2"})
3. (M1, {"name": "java-runtime-environment", "version": "1.5.0_16-133"})

The inside constraint for java-runtime-environment will be represented by two outgoing edges, one to mac-osx, and one to windows-xp. These edges will be labeled as follows:

1. (inside, 0, 0)
2. (inside, 0, 1)

5.3.1.2 Graph generation algorithm

We generate the resource instance graph using the following steps:

1. For each resource instance in the install specification, create a vertex.
2. Go through all the resource definitions corresponding to the install spec instances. For each constraint, do the following:
    a. If the constraint is an "all_of" constraint, analyze all of the sub-constraints.
    b. If the constraint is a "one_of" constraint, look to see whether there exists a corresponding link in the install specification (applies only to inside and peer constraints). If so, create an edge

between the two vertices associated with the install specification resources. Otherwise, apply step c to each element of the one_of list.

    c. If the constraint is a single dependency, perform the following steps:
        i. Retrieve the keys of all resource definitions whose keys match the key constraint and generate a vertex label corresponding to each definition. . For inside and environment constraints, the machine is always the same as the source vertex. For peer constraints, the machine should be explicitly specified by the install specification.
        ii. If any of these keys correspond to an existing vertex in the instance graph, select that one as the target. Otherwise, create a new vertex corresponding to the label with the most recent version number in its key.[1] This vertex is then added to a queue called "PENDING_VERTEX_QUEUE".
        iii. Finally, create an edge between the source vertex and the target vertex corresponding to the constraint.

3. Go through all the vertices in PENDING_VERTEX_QUEUE, processing them as specified in step 2. Note that, in case b of step 2, there will be no corresponding links in the install specification, as the vertices in the queue do not have corresponding resources in the spec. Continue processing new vertices until the queue is empty.
4. Finally, we prune out some edges and vertices to encourage use of the resources in the install specification and to encourage dependency sharing. There are two situations where edges are removed:
    a. If there is a one-of constraint that includes references to resources included in the install spec, we remove all edges to references not in the install spec.
    b. For any vertex not referenced in the install spec where all incoming edges start at vertices that have other choices, we remove these vertices.

   After removing edges according to these rules, we remove any vertices not in the install spec that have no incoming edges.

5.3.2 Constraint generation

The atomic predicates in our constraint have one of the following forms:

- *resource*(m, k) is true if the resource (m, k) is present in the final solution
- *depends*(m_1, k_1, m_2, k_2) is true if resource (m_1, k_1) depends on resource (m_2, k_2)

In the actual generated constraints, we can use short names that are mapped to these longer names.

The following constraints are generated and conjoined:

- For each vertex in the resource instance graph corresponding to a resource instance in the install spec, we assert that the corresponding *resource* predicate is true.
- We create constraints corresponding to the edges in the graph. For each vertex, we look at the edges originating at that vertex. We group the edges into collections having the same first two components in their labels (e.g. (inside, 1, 0) and (inside, 1, 1) are in the same group). For each group, we add an implication that if the atomic *resource* predicate for the source resource is true, exactly one of the atomic *depends* predicates for the target vertices of the group is true.
- For each depends predicate which is generated, we create an implication of the form: *depends*(m_1, k_1, m_2, k_2) $\rightarrow$ *resource*(m_2, k_2).
- For each vertex in the resource instance graph not corresponding to a resource in the install spec, we create an implication that, if the associated resource predicate is true, then the disjunction of all the *depends* predicates targeting this resource must also be true. This ensures that resources are only included in the solution if either they are in the install spec or they satisfy another resource's dependency.

[1] This could prevent resource dependency sharing in situations where one package requires an earlier version of a resource than other packages. One solution is to find the set of keys that satisfy all constraints on a given machine and then pick the latest in that set.

Given a satisfying assignment to this generated predicate, we can determine the set of selected resources by looking at the values of the atomic r*esource* predicates. We can then determine the mapping of input to output ports from the *depends* predicate values. Note that this algorithm assumes that each feasible constraint selection binds each input port to exactly one output port. We achieve this through restrictions on the form of resource constraints (see Section 4.1.3). If this were not the case, we would have to generate additional constraints to ensure this 1-to-1 mapping.

6 Deployment Engine

The deployment engine includes the following components:

- The **dependency analyzer** converts an *install solution* to an *install plan*. It does this by building a dependency graph, performing a (depth-first) traversal of this graph, and outputting calls to the *resource API* for each individual package encountered in the traversal. The install script is designed to run within the *install coordinator.*
- The **resource manager API** defines an interface to be implemented for each package. It provides calls to install the package, configure it, start services, etc.
- The **install coordinator** runs the steps of an install plan. In the single-node case, this just involves calling the appropriate resource manager methods in the order specified by the install plan. In the multi-node case, there are two levels of install coordinators: the *global coordinator* instantiates *slave deployment engines* on the individual target nodes. The execution of the slaves is sequenced according to the inter-node dependencies of the application stack being installed.
- The **upgrade coordinator** sequences the actions to perform an application upgrade.

6.1 Dependency Analyzer

This is component is pretty straightforward. It just creates a dependency graph representation of the resource instances and linearizes the graph using a topological sort. The install plan is just the resulting ordered list, with resources that have no dependencies at the head of the list. For multi-node installs we partition the list by node than then graph a *global install plan*. This global plan is just an ordering of the nodes, based on dependencies between nodes. Given two server nodes A and B, node A has a dependency on node B if and only if node A contains resource instance that has a dependency on a resource instance contained on node B.

6.2 Resource API

Each resource definition has an associated *resource manager* class, which defines the following calls:

validate_pre_install()
Should raise an error if there is a problem in the configuration that would prevent an install (e.g. target directory does not exist or is not writable). We separate this from the install() method so that it can be called as early as possible in the overall deployment. Should not change any state.

is_intalled()
Returns true if the resource has already been installed, false otherwise.

install()
Given the specified version of the software and the configuration data provided in the resource instance, install the package and configure it.

validate_post_intall()
Validate that the install is correct. Called if the package is already installed and we want to see if it is really setup correctly. Should raise an error if there is a problem with the install. Should not change any state. This is not called explicitly by sequencer after install() -- we assume that install() will do its own sanity checks (perhaps by just calling this function). This should NOT be called if we aren't doing the install -- some of the checks may only be valid if we are install the software (e.g. checking that target dir is writable).

is_service()
Return true if the associate resource is also a service (can be started and stopped).

uninstall()
Remove the software and its data files from the system.

upgrade()
Upgrade the resource. Currently, this assumes that the resource has been backed up and uninstalled. It is given a reference to the backup of the previous resource version.

backup()
Backup the resource to the specified location on the local filesystem
restore()
Given a previous backup, restore the resource to the system.

Some of these APIs may be implemented as calls to other utilities (e.g. package managers on Linux).

Services implement the following additional APIs:

start()
Start the service

stop()
Stop the service.

is_running()
Return true if the service is running, false otherwise.

6.3 Install coordinator

The install coordinator executes the install plan, by instantiating resource managers for each resource in the plan, by finding the appropriate package locator for each resource, and then invoking the individual resource manager methods. Currently, a failure in a resource manager invocation causes the install to stop.

The following code implements a simple install coordinator, which calls the appropriate methods to install and start each resource, without any special error handling. It takes two parameters: an *install plan*, which is a list of resource instances sorted in dependency order and a *package library* which provides the package locators and references to the associated resource manager classes for each resource.

```
def run_install(install_plan, library):
    mgr_pkg_list = [get_manager_and_package(instance_md, library)
                    for instance_md in install_plan]
    for (mgr, pkg) in mgr_pkg_list:
        get_logger().info("Processing resource '%s'." % mgr.id)
        if mgr.is_installed():
            mgr.validate_post_install()
            get_logger().info("Resource %s already installed." % mgr.package_name)
        else:
            mgr.install(pkg)
            get_logger().info("Install of %s successful." % mgr.package_name)
        if mgr.is_service():
            if mgr.is_running():
                get_logger().info("Service %s already running." % mgr.package_name)
            else:
                mgr.start()
                get_logger().info("Service %s started successfully." %
                                  mgr.package_name)
```

6.4 Upgrade Coordinator

The upgrade coordinator sequences the upgrade process. The goals for this coordinator are:

1. Be able to perform upgrades of any component in the system, including the application code, database schema/data, dependent components, and Engage itself.
2. If an upgrade fails, roll back to the previous working version of the system. This includes undoing any database schema changes (either by a reverse migration or by restoring from a backup).
3. Optimize the upgrade process, avoiding unnecessary changes.

The current implementation achieves the first two goals, but not the third. In future releases, we will enhance the upgrade process to include optimizations.

Here is how the current upgrade process works:

1. Backup the entire application stack (via the backup() methods of the resource managers classes).
2. Uninstall each component, except for those that were installed globally on the system (e.g. via a package manager). This is done via the uninstall() resource manager method. In most cases, uninstall involves just deleting a directory.
3. Install the new version of the application stack. This is done by calling the upgrade() or install() method of each resource manager, depending on whether there was a corresponding resource in the old configuration. Each upgrade() call is given the new resource's configuration, the old resource's configuration, and the old resource's backup data.
4. If a step of the upgrade/install fails, the (partially) installed components are uninstalled and the backup of the previous version is restored.

7 Grammar for Resource Definitions and Instances

This section contains a grammar for resource definitions and resource instances. All caps names (.e.g MAP_TYPE) are used for grammar rules or lexical tokens. Rules ending in _LIST represent comma-separated lists of the base grammatical form (e.g. SCALAR_LIST is a comma-separated list of SCALARs). The only meta-terms in this grammar are ::= (rule definition) and | (rule case). In addition, the grammar includes comments beginning with "//". The syntax of both resource definitions and resource instances is designed to be valid JSON [JSON]. The lexical tokens are:

- STRING_VALUE represents an arbitrary string value enclosed in double quotes
- INT_VALUE represents an integer value
- BOOL_VALUE is one of *true* or *false*

Whitespace is not significant, except when it appears within a STRING_VALUE.

```
// resource definitions are used to define how a resource may be instantiated
RESOURCE_DEFN ::= { RD_PROPERTY_LIST }

// The end result of the entire process is a collection of resource instances.
// These are used as inputs to the install/configure scripts
// Resource values are intended to be valid JSON.
RESOURCE_INSTANCE ::= { RI_PROPERTY_LIST }

RD_PROPERTY ::= "key" : { SCALAR_PROP_LIST }
              | "display_name": STRING_VALUE
              | "config_port" : { RD_PROP_DEF_LIST }
              | "input_ports": { RD_PORT_PROP_LIST }
              | "output_ports": { RD_PORT_PROP_LIST }
              | "inside" : CHOICE_CONSTRAINT
              | "environment" : COMPOUND_CONSTRAINT
              | "peers" : COMPOUND_CONSTRAINT

SCALAR_PROP ::= STRING_VALUE : SCALAR_VALUE
RD_PORT_PROP ::= STRING_VALUE : { RD_PROP_DEF_LIST }
RD_PROP_DEF ::= STRING_VALUE : SCALAR_TYPE
                  | STRING_VALUE : { RD_TYPE_PROP_LIST }

RD_TYPE_PROP ::= "type": RD_TYPE
                    | "default": JSON_VALUE // can only be used in config ports
                    | "fixed-value": JSON_VALUE // used in output ports to refer to constants
                                                // or input/config properties.
                    | "source": STRING_VALUE // reference to another property. Only used in
                                             // output ports to refer to input/config properties.
                    | "includes": [ JSON_VALUE_LIST ] // used only in input ports

RD_TYPE ::= SCALAR_TYPE
              | { RD_MAP_TYPE_PROP_LIST }
              | [ RD_TYPE_LIST ]

RD_MAP_TYPE_PROP ::= STRING_VALUE: RD_TYPE

SCALAR_TYPE ::= "int"
                    | "string"
                    | “path”  // for filesystem paths
                    | “password”
                    | “hostname”
                    | “tcp_port” // valid tcp/ip port number
                    | “bool”


SCALAR_VALUE ::= STRING_VALUE
               | INT_VALUE
               | BOOL_VAUE

JSON_VALUE ::= SCALAR_VALUE
               | [ JSON_VALUE_LIST ]
               | { JSON_PROP_LIST }

JSON_PROP ::= STRING_VALUE: JSON_VALUE

CHOICE_CONSTRAINT ::= ATOMIC_CONSTRAINT
                    | { “one_of”: [ ATOMIC_CONSTRAINT_LIST ] }

COMPOUND_CONSTRAINT ::= CHOICE_CONSTRAINT
                      | { “all_of”: [ CHOICE_CONSTRAINT_LIST ] }

ATOMIC_CONSTRAINT ::= { “key”: { KEY_CONSTRAINT_LIST },
                        “port_mapping”: { STRING_PROP_LIST }}

KEY_CONSTRAINT ::= STRING_VALUE: SCALAR_VALUE
                 | STRING_VALUE: { KEY_PROP_CONSTRAINT_LIST }

KEY_PROP_CONSTRAINT ::= “greater-than”: STR_INT_VALUE
                      | “greater-than-or-equal”: STR_INT_VALUE
                      | “less-than”: STR_INT_VALUE
                      | “less-than-or-equal”: STR_INT_VALUE

STR_INT_VALUE ::= STRING_VALUE
                | INT_VALUE

STRING_PROP ::= STRING_VALUE: STRING_VALUE

//
// Grammar rules for resource instances
//

RI_PROPERTY ::= "id" : STRING_VALUE
          | "key" : { SCALAR_PROP_LIST }
          | "config_port": { JSON_PROP_LIST }
          | "input_ports": { JSON_PORT_VALUE_MAP_LIST }
          | "output_ports": { JSON_PORT_VALUE_MAP_LIST }
          | "inside": RESOLVED_CONSTRAINT
          | "peers": [ RESOLVED_CONSTRAINT_LIST ]
          | "environment": [ RESOLVED_CONSTRAINT_LIST ]

JSON_PORT_VALUE_MAP ::= STRING_VALUE: { JSON_PROP_LIST }

RESOLVED_CONSTRAINT ::= { "id": STRING_VALUE,
                          "key": { SCALAR_PROP_LIST },
                          "port_mapping": { STRING_PROP_LIST } }
```

8 Examples

Here, we show example resource definitions and resource instances for OpenMRS.

8.1 Resource Definitions for OpenMRS

```
{ "key": {"name":"mac-osx", "version":"10.5.6"},
  "display_name": "Machine (mac-osx)",
  "config_port": {
    "hostname" : {"type":"hostname", "default":"localhost"},
    "os_user_name" : "string",
    "cpu_arch" : {"type":"string", "default":"x86"}
   },
  "output_ports": {
    "host": {
      "hostname": {"type":"hostname", "source":"config_port.hostname"},
      "os_type" : {"type":"string", "fixed-value":"mac-osx"},
      "os_user_name" : {"type": "string",
                        "source":"config_port.os_user_name"},
      "cpu_arch" : {"type":"string", "source":"config_port.cpu_arch"},
      "genforma_home" : {"type":"path",
                         "fixed-value":"/Users/${config_port.os_user_name}/apps"}
    }
  }
},
{ "key": { "name": "ubuntu-linux", "version": "9.0.4"},
  "display_name": "Ubuntu Linux 9.0.4",
  "config_port": {
    "hostname": {"type":"hostname", "default":"localhost"},
    "os_user_name" : "string",
    "cpu_arch" : {"type":"string", "default":"x86"}
  },
  "output_ports": {
    "host": {
      "hostname": {"type":"hostname", "source":"config_port.hostname"},
      "os_type" : {"type":"string", "fixed-value":"windows-xp"},
      "os_user_name" : {"type": "string",
                        "source":"config_port.os_user_name"},
      "cpu_arch" : {"type":"string", "source":"config_port.cpu_arch"},
      "genforma_home" : {"type":"path",
                         "fixed-value":"/home/${config_port.os_user_name}/genforma"}
    }
  }
},
{ "key": {"name":"mysql", "version":"5.1"},
  "display_name": "MySQL 5.1",
  "config_port": {
    "install_dir" : {"type":"path",
                     "display_name": "Install directory",
                     "default":"${input_ports.host.genforma_home}/mysql-5.1"},
    "port": {"display_name": "TCP/IP Port",
             "type":"tcp_port", "default":3306},
    "admin_password": {"type":"password",
                       "display_name":"Adminstrator's password"}
  },
  "input_ports": {
    "host" : {
      "hostname": "hostname",
      "os_type": "string",
      "os_user_name": "string",
      "cpu_arch": "string",
      "genforma_home": "path"
    }
  },
  "output_ports": {
    "mysql" : {
      "host": {"type":"hostname", "source":"input_ports.host.hostname"},
      "port": {"type":"tcp_port", "source":"config_port.port"}
    },
    "mysql_admin" : {
      "root_password": {"type":"password",
                        "source":"config_port.admin_password"},
      "install_dir": {"type":"string",
                      "source":"config_port.install_dir"}
    }
  },
  "inside": {
    "one-of": [
      { "key": {"name":"mac-osx", "version":{"greater-than":"10.5.0",
                                             "less-than":"10.6"}},
        "port_mapping": {"host":"host"}},
      { "key": {"name": "ubuntu-linux", "version":"9.0.4"},
        "port_mapping": {"host":"host"}}
    ]
  }
},
{ "key": {"name":"java-developer-kit", "version":"1.5.0_16-133"},
  "display_name":"JDK 1.5",
  "config_port": {
    "JAVA_HOME": "string"
  },
  "input_ports": {
    "host" : {
      "hostname": "hostname",
      "os_type": "string",
      "os_user_name": "string",
      "cpu_arch": "string"
    }
  },
  "output_ports": {
    "jdk": {
      "type": {"type":"string", "fixed-value":"jdk"},
      "home": {"type":"string", "source":"config_port.JAVA_HOME"}
    }
  },
  "inside": {
    "one-of": [
      { "key": {"name":"mac-osx", "version":{"greater-than":"10.5.0",
                                             "less-than":"10.6"}},
        "port_mapping": {"host":"host"}},
      { "key": {"name": "ubuntu-linux", "version":"9.0.4"},
        "port_mapping": {"host":"host"}}
    ]
  }
},
{ "key": {"name": "apache-tomcat", "version": "6.0.18"},
  "display_name": "Apache Tomcat 6.0.18",
  "config_port": {
    "admin_user": {"type":"username", "default":"admin",
                   "display_name":"Adminstration user"},
    "admin_password": {"type":"password",
                       "display_name":"Adminstrator's password"},
    "manager_port": {"type":"tcp_port", "default":8080,
                     "display_name":"TCP/IP Port"},
    "home" : {"type":"path",
              "display_name": "Install directory",
              "default":"${input_ports.host.genforma_home}/tomcat-6.0.18"}
  },
  "input_ports": {
    "host" : {
      "hostname": "hostname",
      "os_type": "string",
      "os_user_name": "string",
      "cpu_arch": "string",
      "genforma_home": "path"
    },
    "java": {
      "type": "string",
      "home": "string"
    }
  },
  "output_ports": {
    "tomcat": {
      "admin_user": {"type":"username", "source":"config_port.admin_user"},
      "admin_password": {"type":"password",
                         "source":"config_port.admin_password"},
      "hostname": {"type":"hostname", "source":"input_ports.host.hostname"},
      "manager_port": {"type":"tcp_port",
                       "source":"config_port.manager_port"},
      "home": {"type":"path", "source":"config_port.home"},
      "environment_vars": {"type":[{"name":"string", "value":"string"}]},
      "genforma_home": {"type":"path", "source":"input_ports.host.genforma_home"}
```

```
          }
        },
        "inside": {
          "one-of": [
            { "key": {"name":"mac-osx", "version":{"greater-than":"10.5.0",
                                             "less-than":"10.6"}},
              "port_mapping": {"host":"host"}},
            { "key": {"name": "ubuntu-linux", "version":"9.0.4"},
              "port_mapping": {"host":"host"}}
          ]
        },
        "environment": {
          "one-of": [
            { "key": {"name":"java-developer-kit",
                      "version":{"greater-than-or-equal":"1.5"}},
              "port_mapping": {"java":"jdk"}},
            { "key": {"name":"java-runtime-environment",
                      "version":{"greater-than-or-equal":"1.5"}},
              "port_mapping": {"java":"jre"}}
          ]
        }
      },
      { "key": {"name": "OpenMRS", "version": "1.3.4"},
        "display_name": "OpenMRS 1.3.4",
        "config_port": {
          "database_user": {"type":"username", "default":"openmrs",
                            "display_name":"Database user"},
          "database_password": {"type":"password",
                                "display_name":"Database user's password"},
          "home" : {"type":"path",
                    "display_name":"Install directory",
                    "default":"${input_ports.tomcat.genforma_home}/openmrs-1.3.4"}
        },
        "input_ports": {
          "java": {
            "type": "string",
            "home": "string"
          },
          "tomcat": {
            "admin_user": "username",
            "admin_password": {"type":"password"},
            "hostname": "hostname",
            "manager_port": {"type":"tcp_port"},
            "home": {"type":"path"},
            "genforma_home": "path",
            "environment_vars": {
              "type":[{"name":"string", "value":"string"}],
              "includes":[{"name":"OPENMRS_RUNTIME_PROPERTIES_FILE",
                        "value":"${config_port.home}/runtime.properties"}]
            }
          },
          "mysql": {
            "host": "hostname",
            "port": "tcp_port"
          },
          "mysql_admin": {
           "root_password": "password",
           "install_dir": "path"
          }
        },
        "output_ports": {
          "url" : {
            "application_url": {"type":"string",
                                "fixed-value":
             "http://${input_ports.tomcat.hostname}:${input_ports.tomcat.manager_port}/openmrs/login.htm"}
          }
        },
        "inside": {
          "key": {"name": "apache-tomcat", "version": "6.0.18"},
          "port_mapping": {"tomcat": "tomcat"}
        },
        "environment": {
          "one-of": [
            { "key": {"name":"java-developer-kit",
                      "version":{"greater-than-or-equal":"1.5"}},
              "port_mapping": {"java":"jdk"}},
            { "key": {"name":"java-runtime-environment",
                      "version":{"greater-than-or-equal":"1.5"}},
          "port_mapping": {"java":"jre"}}
        ]
      },
      "peers":
        { "key": {"name":"mysql",
                  "version":{"greater-than-or-equal":"5.1",
                             "less-than":"6.0"}},
          "port_mapping": {"mysql":"mysql", "mysql_admin":"mysql_admin"}
        }
  }
```

8.2 Resource instances for OpenMRS

These (partial) resource instances form an "install specification" for OpenMRS. These are combined with the resource definitions to build constraints for an install solution.

```
{
  "id": "machine-1",
  "key": { "name": "mac-osx", "version": "10.5.6"},
  "properties": {
    "installed":true
  },
  "config_port": {
    "hostname": "jfischer.local",
    "os_user_name" : "jfischer",
    "cpu_arch": "x86"
  }
},
{
  "id": "mysql-1",
  "key": {"name":"mysql", "version":"5.1"},
  "inside": {
    "id": "machine-1",
    "key": { "name": "mac-osx", "version": "10.5.6"},
    "port_mapping": {"host":"host"}
  }
},
{
  "id": "jdk-1",
  "key": {"name":"java-developer-kit", "version":"1.5.0_16-133"},
  "properties": {
    "installed":true
  },
  "config_port": {
    "JAVA_HOME": "/System/Library/Frameworks/JavaVM.framework/Versions/CurrentJDK/Home"
  },
  "inside": {
    "id": "machine-1",
    "key": {"name":"mac-osx", "version":"10.5.6"},
    "port_mapping": {"host": "host"}
  }
},
{
  "id": "tomcat-1",
  "key": {"name": "apache-tomcat", "version": "6.0.18"},
  "inside": {
     "id": "machine-1",
     "key": {"name":"mac-osx", "version":"10.5.6"},
     "port_mapping": {"host": "host"}
   },
  "environment": [
    { "id": "jdk-1",
      "key": {"name":"java-developer-kit", "version":"1.5.0_16-133"},
      "port_mapping": {"java":"jdk"}
    }
  ]
},
{ "id": "openmrs-1",
  "key": {"name": "OpenMRS", "version": "1.3.4"},
  "inside": {
     "id": "tomcat-1",
     "key": {"name": "apache-tomcat", "version": "6.0.18"},
     "port_mapping": {"tomcat": "tomcat"}
   },
  "peers": [
    { "id": "mysql-1",
      "key": {"name":"mysql", "version":"5.1"},
      "port_mapping": {"mysql":"mysql", "mysql_admin":"mysql_admin"}
    }
  ]
}
```

9 References

[DebVer] Version Numbers, Debian Policy Manual. http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version

[JSON] Crockford, D. Java Script Object Notation. json.org.

[Kahn62] Kahn, A. B. (1962), "Topological sorting of large networks", *Communications of the ACM* **5** (11): 558–562.

[Mancinelli06] 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.

[OpenMRS] www.openmrs.org.

[Tucker07] 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.