A Formal Model of the SpringSource Application Platform (v0.2)

Glyn Normington

June 18, 2008

The aim is to model the basic architecture of the SpringSource Application Platform.

Contents

1 Standard OSGi

Bundles have symbolic names (which are strings, but we ignore that here) and versions.

$$[SymbolicName, Version]$$

A bundle definition defines the bundle's name and version as well as other things we will not model here.

BundleDef

$$name : SymbolicName$$
$$version : Version$$

A bundle can be in one of several states.

$$BundleState ::= bundleInstalled \mid bundleResolved \mid bundleStarting \mid bundleActive \mid bundleUnknown \mid bundleStopping \mid bundleUninstalled$$

Note : we are modelling a bundle instance rather than a bundle definition, so we may not need the uninstalled state.

We'll need to talk about class loaders but only in terms of them providing a name space for classes.

$$[ClassName, Class]$$

We currently ignore the distinction between initiating and defining class loaders and model a class loader as a one-one map of class names to classes.

ClassLoader

$$cl : ClassName \rightarrowtail Class$$

Each class instance uniquely determines its defining class loader.

$$getClassLoader : Class \nrightarrow ClassLoader$$

$$\forall c : \mathrm{dom}\, getClassLoader \bullet c \in \mathrm{ran}(getClassLoader\ c).cl$$

A bundle has a name and a state.

Bundle

$$name : SymbolicName$$
$$version : Version$$
$$state : BundleState$$

The standard OSGi framework uniquely identifies each installed bundle by name and version. The OSGi framework also loads the classes of each bundle using

a separate class loader. Bundles which have reached the starting state have a class loader whereas those which are installed or resolved will only have a class loader if they have reached the starting state in the past.

$$
\begin{array}{l}
\textit{StandardOSGi} \\
\hline
\textit{bundles} : \textit{SymbolicName} \times \textit{Version} \rightarrowtail\!\!\!\!\!\rightarrow \textit{Bundle} \\
\textit{cl} : \textit{Bundle} \rightarrowtail\!\!\!\!\!\rightarrow \textit{ClassLoader} \\
\hline
\forall\, n : \textit{SymbolicName};\ v : \textit{Version};\ b : \textit{Bundle} \mid \\
\quad (n, v) \mapsto b \in \textit{bundles} \bullet \\
\qquad n = b.\textit{name} \land v = b.\textit{version} \\
\forall\, b : \operatorname{ran} \textit{bundles} \mid \\
\quad b.\textit{state} \notin \{\textit{bundleInstalled}, \textit{bundleResolved}\} \bullet \\
\qquad b \in \operatorname{dom} \textit{cl} \\
\hline
\end{array}
$$

A bundle may be installed in the OSGi framework by providing its bundle definition. We allow for the installation of a bundle to install other bundles (from a repository as we will see later).

$$
\begin{array}{l}
\textit{InstallBundleOk\_so} \\
\hline
\Delta \textit{StandardOSGi} \\
\textit{bd?} : \textit{BundleDef} \\
\hline
(\textit{bd?}.\textit{name}, \textit{bd?}.\textit{version}) \notin \operatorname{dom} \textit{bundles} \\
\exists\, \textit{Bundle}' \bullet \\
\quad \textit{name}' = \textit{bd?}.\textit{name} \land \textit{version}' = \textit{bd?}.\textit{version} \land \\
\quad \textit{state}' = \textit{bundleInstalled} \land \\
\quad \{(\textit{name}', \textit{version}') \mapsto \theta \textit{Bundle}'\} \cup \textit{bundles} \subseteq \textit{bundles}' \\
\hline
\end{array}
$$

2 Extended OSGi

The SpringSource Application Platform extends standard OSGi by adding a number of concepts including the concept of a *library*.

A library has a symbolic name, a version, and a list of bundles. Each bundle in the list has a unique symbolic name.

Library

$$
\begin{array}{l}
name : SymbolicName \\
version : Version \\
bundles : \operatorname{seq} Bundle \\
\hline
\forall\, i, j : \operatorname{dom} bundles \bullet \\
\quad (bundles\, i).name = (bundles\, j).name \Rightarrow i = j
\end{array}
$$

Note: check this constraint is enforced by the Platform.

The extended OSGi framework adds a collection of libraries to the standard framework. Each library is uniquely identified by name and version and its bundles are present in the framework.

ExtendedOSGi

$$
\begin{array}{l}
StandardOSGi \\
libraries : SymbolicName \times Version \pfun Library \\
\hline
\forall\, n : SymbolicName;\ v : Version;\ l : Library \mid \\
\quad (n, v) \mapsto l \in libraries \bullet \\
\qquad n = l.name \land v = l.version \\
\forall\, l : \operatorname{ran} libraries \bullet \\
\quad \operatorname{ran} l.bundles \subseteq \operatorname{ran} bundles
\end{array}
$$

3 Subsystems

Subsystem names are strings, but we ignore that here.

$$[SubsystemName]$$

A subsystem can be in one of several states.

$$SubsystemState ::= subsystemInstalling \mid subsystemInstalled \mid subsystemStarting \mid subsystemStarted \mid subsystemUnknown \mid subsystemStopping \mid subsystemStopped \mid subsystemUninstalling$$

The Platform is composed of *subsystems* each of which has a name, a status, and a set of bundles.

$$\begin{array}{|l} \textit{Subsystem} \\ \hline name : SubsystemName \\ state : SubsystemState \\ bundles : \mathbb{P}\, Bundle \\ \hline \end{array}$$

Note: there relationship between the state of a subsystem and the states of its bundles is not modelled here. It is not an invariant except in steady states.

4 Subsystem Registry

The subsystem registry maintains a collection of subsystems indexed by name and navigational information to determine the bundle for a given class loader and the subsystem for a given bundle.

SubsystemRegistry

$$
\begin{array}{l}
ExtendedOSGi \\
ss : SubsystemName \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow Subsystem \\
b : ClassLoader \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow Bundle \\
bs : Bundle \twoheadrightarrow\!\!\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow Subsystem \\
\hline
b = cl^{\sim} \\
\forall\, n : \mathrm{dom}\, ss \bullet (ss\ n).name = n \\
\mathrm{ran}\, b \subseteq \mathrm{dom}\, bs \\
\mathrm{ran}\, bs \subseteq \mathrm{ran}\, ss \\
\forall\, s : \mathrm{ran}\, ss \mid s.state \neq subsystemInstalling \bullet \\
\qquad s.bundles \subseteq bs^{\sim}(\!|\, \{s\} \,|\!)
\end{array}
$$

getSubsystemByClassOk

$$
\begin{array}{l}
\Xi SubsystemRegistry \\
c? : Class \\
s! : Subsystem \\
\hline
\exists\, cl : ClassLoader \mid cl = getClassLoader\ c? \bullet \\
\qquad cl \in \mathrm{dom}\, b \land \\
\qquad s! = bs(b\ cl)
\end{array}
$$

5 Repository

A repository holds bundle and library definitions.

A library definition defines the library's name and version and a specification of a collection of bundles. Each bundle in the specification has a name and a version range (modelled here as a set of versions).

LibraryDef

$$
\begin{array}{l}
name : SymbolicName \\
version : Version \\
bundleSpecs : SymbolicName \pfun \mathbb{P}\ Version
\end{array}
$$

The repository uniquely identifies each bundle and library by name and version.

Repository

$$
\begin{array}{l}
bundleDefs : SymbolicName \times Version \pfun BundleDef \\
libraryDefs : SymbolicName \times Version \pfun LibraryDef \\
\hline
\forall\, n : SymbolicName;\ v : Version;\ bd : BundleDef \mid \\
\quad (n, v) \mapsto bd \in bundleDefs \bullet \\
\quad\quad n = bd.name \land v = bd.version \\
\forall\, n : SymbolicName;\ v : Version;\ ld : LibraryDef \mid \\
\quad (n, v) \mapsto ld \in libraryDefs \bullet \\
\quad\quad n = ld.name \land v = ld.version
\end{array}
$$

The internal representation of a repository may be quite different from the model presented here. The aim here is to provide a specification that defines how a repository behaves rather than a detailed description of a particular repository implementation.

The repository provides an operation for adding a bundle.

AddBundleOk

$$
\begin{array}{l}
\Delta Repository \\
bd? : BundleDef \\
\hline
(bd?.name, bd?.version) \notin \mathrm{dom}\ bundleDefs \\
bundleDefs = bundleDefs \cup \{(bd?.name, bd?.version) \mapsto bd?\}
\end{array}
$$

The implementation allows any bundle definition to be added to the repository, even if it is already present.

The repository also provides an operation for adding a library.

AddLibraryOk

$$
\begin{array}{l}
\Delta Repository \\
ld? : LibraryDef \\
\hline
(ld?.name, ld?.version) \notin \mathrm{dom}\ libraryDefs \\
libraryDefs = libraryDefs \cup \{(ld?.name, ld?.version) \mapsto ld?\}
\end{array}
$$

6 Installation

Bundles are installed in OSGi and then started in order to initialise the Platform and to deploy applications on to the Platform.

Installation takes some bundles and attempts to satisfy their dependencies by implicitly installing other bundles from the repository. Installation automatically starts bundles implicitly installed from the repository, but does not start the bundles that were input to installation.

InstallBundle

$$\Xi Repository$$
$$\Delta ExtendedOSGi$$
$$InstallBundleOk\_so$$

$$\exists\, i : SymbolicName \times Version \rightarrowtail BundleDef \bullet$$
$$i \subseteq bundleDefs \land$$
$$\operatorname{dom} bundles' = \operatorname{dom} bundles \cup \{(bd?.name, bd?.version)\} \cup \operatorname{dom} i$$

Note: the above schema does not model the states of the implicitly installed bundles. Each such bundle is installed from the corresponding bundle definition in the repository and is started after being installed.

7 Deployment

Quite separately from the installation of bundles during deployment, we need to model the way application deployment is exposed to the user.

Users deal with deployable artefacts such as Platform archive files (PARs), web archive files (WARs), and single OSGi bundles (JARs). We abstract away from the differences and model a deployable artefact as having just a *deployment identity* consisting of a symbolic name and a version.

DeploymentIdentity

$$name : SymbolicName$$
$$version : Version$$

We don't bother modelling the contents of a deployment artefact other than its deployment identifier. of deployable artefacts.

DeployableArtefact

$$DeploymentIdentity$$

A deployable artefact may be deployed either by calling a Platform API passing the location of the artefact in the file system or by copying the artefact into the Platform's hot deployment directory.

$$[Location]$$

$$Boolean ::= yes \mid no$$

FileSystem

$$fs : Location \pfun DeployableArtefact$$

Artefacts deployed in the Platform are identified by name and version. Artefacts which were hot deployed are stored in a portion of the file system and each such artefact has a unique location within that portion of the file system. Some deployed artefacts are recoverable including all those stored in the deployer's portion of the file system.

Deployed

$$dep : DeploymentIdentity \pinj DeployableArtefact$$
$$rec : \mathbb{P}\, DeployableArtefact$$
$$FileSystem_0$$

$$\forall\, DeploymentIdentity;\ da : DeployableArtefact \mid$$
$$\quad \theta DeploymentIdentity \mapsto da \in dep \bullet$$
$$\quad\quad name = da.name \land version = da.version$$
$$\operatorname{ran} fs_0 \subseteq \operatorname{ran} dep$$
$$fs_0 \in Location \pinj DeployableArtefact$$
$$rec \subseteq \operatorname{ran} dep$$
$$\operatorname{ran} fs_0 \subseteq rec$$

Deploying from a location stores the deployable artefact but does not remember the location.

DeployLocationOk

$$
\begin{array}{l}
\Xi FileSystem \\
\Delta Deployed \\
loc? : Location \\
recoverable? : Boolean \\
\hline
\mathrm{dom}\, fs_0 \cap \mathrm{dom}\, fs = \emptyset \\
loc? \in \mathrm{dom}\, fs \\
fs_0' = fs_0 \\
\exists\, d : DeployableArtefact;\ DeploymentIdentity \mid \\
\quad d = fs\, loc? \land d.name = name \land d.version = version \bullet \\
\quad \theta DeploymentIdentity \notin \mathrm{dom}\, dep \land \\
\quad dep' = dep \cup \{\theta DeploymentIdentity \mapsto d\} \land \\
\quad recoverable? = yes \Rightarrow rec' = rec \cup \{d\} \\
recoverable? = no \Rightarrow rec' = rec
\end{array}
$$

Hot deploying a deployable artefact stores the artefact and gives it a location in the deployer's portion of the file system.

HotDeployOk

$$
\begin{array}{l}
\Xi FileSystem \\
\Delta Deployed \\
da? : DeployableArtefact \\
\hline
\exists\, DeploymentIdentity \mid name = da?.name \land version = da?.version \bullet \\
\quad \theta DeploymentIdentity \notin \mathrm{dom}\, dep \land \\
\quad dep' = dep \cup \{\theta DeploymentIdentity \mapsto da?\} \\
\exists\, l' : Location \mid l' \notin \mathrm{dom}\, fs \cup \mathrm{dom}\, fs_0 \bullet \\
\quad fs_0' = fs_0 \cup \{l' \mapsto da?\} \\
rec' = rec \cup \{da?\}
\end{array}
$$

A artefact that has already been deployed can be undeployed in a number of ways. The most general way is to pass in deployment identity (equivalent to the name and version).

UndeployNameVersionOk

$$
\begin{array}{l}
\Delta Deployed \\
di? : DeploymentIdentity \\
\hline
di? \in \mathrm{dom}\, dep \\
dep' = \{di?\} \lhd dep \\
fs_0' = fs_0 \rhd \{dep\ di?\} \\
rec' = rec \setminus \{dep\ di?\}
\end{array}
$$

The other way to undeploy an artefact is to delete an artefact that was hot deployed which we model as passing in the location of the hot deployed artefact.

HotUndeployOk

$\Delta Deployed$
$loc? : Location$

$loc? \in \operatorname{dom} fs_0$
$fs_0' = \{loc?\} \lhd fs_0$
$dep' = dep \rhd \{fs_0\ loc?\}$
$rec' = rec \setminus \{fs_0\ loc?\}$

Recovery deletes non-recoverable artefacts.

Recover

$\Delta Deployed$

$dep' = dep \rhd rec$
$rec' = rec$
$fs_0' = fs_0$

8 Z Notation

Numbers:

$\mathbb{N}$ `Natural numbers` {0,1,...}

Propositional logic and the schema calculus:

$\ldots \wedge \ldots$	And	$\langle\!\langle \ldots \rangle\!\rangle$	Free type injection
$\ldots \vee \ldots$	Or	$[\ldots]$	Given sets
$\ldots \Rightarrow \ldots$	Implies	$', ?, !, {}_0 \ldots {}_9$	Schema decorations
$\forall .. \mid .. \bullet ..$	For all	$\ldots \vdash \ldots$	theorem
$\exists .. \mid .. \bullet ..$	There exists	$\theta \ldots$	Binding formation
$\ldots \setminus \ldots$	Hiding	$\lambda \ldots$	Function definition
$\ldots \mathrel{\widehat=} \ldots$	Schema definition	$\mu \ldots$	Mu-expression
$\ldots == \ldots$	Abbreviation	$\Delta \ldots$	State change
$\ldots ::= \ldots \mid \ldots$	Free type definition	$\Xi \ldots$	Invariant state change

Sets and sequences:

$\{\ldots\}$	Set	$\ldots \setminus \ldots$	Set difference
$\{.. \mid .. \bullet ..\}$	Set comprehension	$\bigcup \ldots$	Distributed union
$\mathbb{P} \ldots$	Set of subsets of	$\# \ldots$	Cardinality
$\emptyset$	Empty set	$\ldots \subseteq \ldots$	Subset
$\ldots \times \ldots$	Cartesian product	$\ldots \subset \ldots$	Proper subset
$\ldots \in \ldots$	Set membership	$\ldots$ partition $\ldots$	Set partition
$\ldots \notin \ldots$	Set non-membership	seq	Sequences
$\ldots \cup \ldots$	Union	$\langle \ldots \rangle$	Sequence
$\ldots \cap \ldots$	Intersection	disjoint $\ldots$	Disjoint sequence of sets

Functions and relations:

$\ldots \leftrightarrow \ldots$	Relation	$\ldots \mapsto \ldots$	maplet
$\ldots \nrightarrow \ldots$	Partial function	$\ldots^\sim$	Relational inverse
$\ldots \rightarrow \ldots$	Total function	$\ldots^*$	Reflexive-transitive closure
$\ldots \rightarrowtail \ldots$	Partial injection	$\ldots (\!\mid \ldots \mid\!)$	Relational image
$\ldots \rightarrowtail \ldots$	Injection	$\ldots \oplus \ldots$	Functional overriding
dom $\ldots$	Domain	$\ldots \lhd \ldots$	Domain restriction
ran $\ldots$	Range	$\ldots \lhd \ldots$	Domain subtraction

Axiomatic descriptions:

Declarations

Predicates

Schema definitions:

SchemaName

Declaration

Predicates