dm Kernel 2.0 (v0.8)

Glyn Normington

May 11, 2009

The Jersey release of the SpringSource dm Server is based on a separate *kernel* component. This document provides a partial formal model of the kernel.

Contents

1 Overview

The dm Kernel is an embeddable OSGi component that manages various kinds of artefacts which may be obtained via a repository.

The embedding features of the kernel are discussed in the next section after which installation and uninstallation of artefacts, the runtime artefact model, and start and stop of artefacts are specified.

2 Embedding

Since the kernel may be *embedded* in a larger program, it does not have sole control of the OSGi framework on which it runs.

This is a helpful perspective even for programs like dm Server which use the kernel but which do not operate directly on the OSGi framework, since application code installed in dm Server is free to operate directly on the OSGi framework outside the control of the kernel.

To provide predictable overall behaviour, the kernel must operate on the OSGi framework in a well defined, and preferably intuitively appealing, way.

In the absence of other programs operating directly on the OSGi framework, the kernel presents a relatively clean abstraction to its users. In the presence of such programs, the abstraction breaks down and users need to understand the kernel's interaction with the OSGi framework.

So we need to take into account the *embedded* and *non-embedded* use cases in the design of the kernel. As observed above, the non-embedded use case merges into the embedded use case as soon as application code starts operating directly on the OSGi framework. Thankfully, this is explicit in the design of application code and the choice of third party libraries, most of which were developed without OSGi in mind. So the complexity of the kernel's behaviour increases in a "pay as you go" manner.

3 Artefacts and the Runtime Artefact Model

One of the primary functions of the kernel is to enable various kinds of *artefact* to be installed and started. Artefacts include OSGi bundles, PAR files, WAR files, configurations, and *plans*. A plan refers to zero or more artefacts which are to be installed when the plan is installed.

$$ATType ::= BUNDLE \mid PAR \mid WAR \mid CONFIG \mid PLAN$$

Certain types or arfefact may be installed in an OSGi framework.

$$OSGITYPES == \{BUNDLE, WAR\}$$

Artefacts have names and versions, although whether or how these and artefact types relate to the content of an artefact need not concern us here.

$$[Artefact, AName, AVersion]$$

In this specification, we deliberately ignore the distinction between artefacts, their descriptions, and transformed artefacts.

We abbreviate the artefact type, name, version triple.

$$TNV == AType \times AName \times AVersion$$

We define a relation for checking that a type, name, version triple has a supported type.

$$\begin{array}{|l}
\_conformsTo\_ : TNV \leftrightarrow (\mathbb{P}\, AType) \\
\hline
\forall\, t : AType;\ n : AName;\ v : AVersion;\ ts : \mathbb{P}\, AType \bullet \\
\quad ((t, n, v)\ conformsTo\ ts) \Leftrightarrow (t \in ts)
\end{array}$$

The kernel maintains a portion of the dependency graph known as the *runtime artefact model* and uses the OSGi framework to determine the remaining dependencies. The runtime artefact model records the *root artefacts* (the artefacts which have been explicitly installed, *roots* for short), plan resolution decisions (which artefact is chosen to satisfy each version range in a plan), and the bundles which are cloned into a plan.

The runtime artefact model does not record which plan is chosen to satisfy each `import-library` manifest header or which bundle is chosen to satisfy each `import-bundle` manifest header since these headers result in package dependencies which are managed by the OSGi framework. This split is necessary to keep the dependency graph accurate. For example, if a bundle in the graph is updated and an OSGi `refreshPackages` operation is performed, the graph may change significantly and it would be extremely difficult, if not impossible, for the kernel to maintain an accurate copy of the OSGi framework's dependencies.

The runtime artefact model also records the lifecycle state of each artefact in the model.

$$\begin{aligned} \mathit{AState} ::= {} & \mathit{INSTALLED} \mid \mathit{RESOLVED} \mid \mathit{STARTING} \mid \mathit{ACTIVE} \mid \\ & \mathit{STOPPING} \mid \mathit{UNINSTALLING} \mid \mathit{UNINSTALLED} \end{aligned}$$

An artefact which is in the *RESOLVED*, *STARTING*, *ACTIVE*, or *STOPPING* state is resolved. In other words *STARTING*, *ACTIVE*, and *STOPPING* can be regarded as sub-states of *RESOLVED*.

$$\begin{aligned} \mathit{RESOLVED\_STATES} == \{ & \mathit{RESOLVED}, \mathit{STARTING}, \mathit{ACTIVE}, \\ & \mathit{STOPPING}\} \end{aligned}$$

An artefact which is in the *STARTING*, *ACTIVE*, or *STOPPING* state is active. In other words *STARTING* and *STOPPING* can be regarded in some circumstances as equivalent to *ACTIVE*.

$$\mathit{ACTIVE\_STATES} == \{\mathit{STARTING}, \mathit{ACTIVE}, \mathit{STOPPING}\}$$

The runtime artefact model has a set of supported artefact types, a collection of artefacts, each uniquely identified by type, name, and version, a set of root artefacts, a dependency relation, a unique state for each artefact, and set of resolved and actcive artefacts.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAM}} \\
\hline
types : \mathbb{P}\, AType \\
art : TNV \rightarrowtail\!\!\!\!\!+\!\!\!\!\rightarrow Artefact \\
roots : \mathbb{P}\, TNV \\
dependsOn : TNV \leftrightarrow TNV \\
st : TNV \nrightarrow AState \\
resolved : \mathbb{P}\, TNV \\
active : \mathbb{P}\, TNV \\
\hline
\forall\, tnv : \mathrm{dom}\, art \bullet tnv\ conformsTo\ types \\
roots \subseteq \mathrm{dom}\, art \\
\mathrm{dom}\, st = \mathrm{dom}\, art \\
dependsOn \subseteq \mathrm{dom}\, art \times \mathrm{dom}\, art \\
resolved = st^{\sim}(\!|\ RESOLVED\_STATES\ |\!) \\
active = st^{\sim}(\!|\ ACTIVE\_STATES\ |\!) \\
dependsOn(\!|\ resolved\ |\!) \subseteq resolved \\
dependsOn(\!|\ active\ |\!) \subseteq active \\
\hline
\end{array}
$$

Each artefact in the model has a supported type. Each root artefact is known in the model. Some artefacts in the model depend on other artefacts in the model. Each artefact has a state. Resolved artefacts are those with a *RESOLVED*, *STARTING*, *ACTIVE*, or *STOPPING* state. Active artefacts are those with a *STARTING*, *ACTIVE*, or *STOPPING* state. Active artefacts may only depend on other active artefacts.

Each kernel instance has a runtime artefact model under its sole control. Kernel instances running on an OSGi framework share the content of the framework. These relationships are depicted in Figure 1.

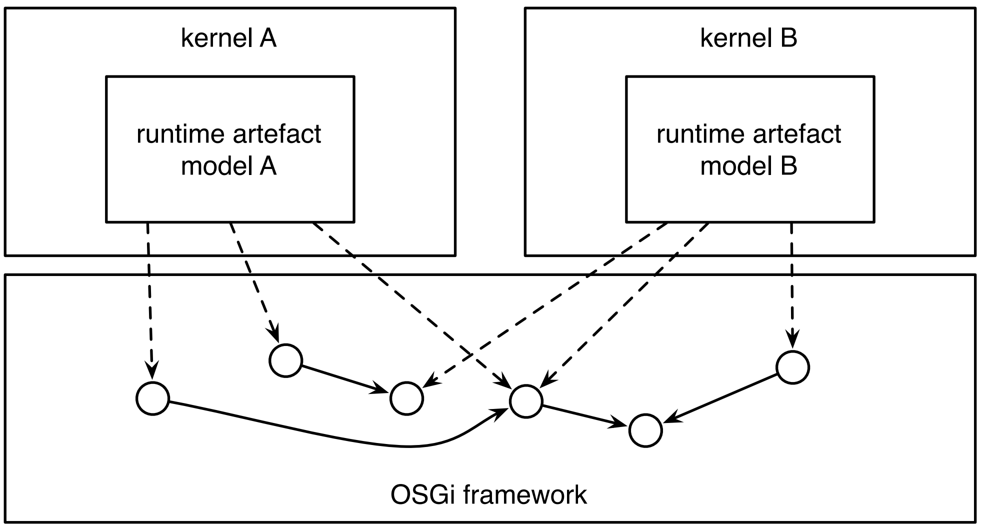

Figure 1: Multiple kernels and runtime artefact models

4 Runtime Artefact Model Operations

Initially a runtime artefact model has a set of supported artefact types and no contents.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMInit}} \\
\hline
\mathit{RAM}' \\
\mathit{types}? : \mathbb{P}\,\mathit{AType} \\
\hline
\mathit{types}' = \mathit{types}? \\
\mathit{art}' = \emptyset \\
\hline
\end{array}$$

The set of root artefacts may be inquired.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMgetRoots}} \\
\hline
\Xi \mathit{RAM} \\
\mathit{roots}! : \mathbb{P}\,\mathit{TNV} \\
\hline
\mathit{roots}! = \mathit{roots} \\
\hline
\end{array}$$

The set of supported artefact types may be inquired.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMgetArtefactTypes}} \\
\hline
\Xi \mathit{RAM} \\
\mathit{types}! : \mathbb{P}\,\mathit{AType} \\
\hline
\mathit{types}! = \mathit{types} \\
\hline
\end{array}$$

An artefact may be inquired by type, name, and version.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMgetArtefact}} \\
\hline
\Xi \mathit{RAM} \\
t? : \mathit{AType} \\
n? : \mathit{AName} \\
v? : \mathit{AVersion} \\
a! : \mathit{Artefact} \\
\hline
(t?, n?, v?) \in \operatorname{dom} \mathit{art} \\
a! = \mathit{art}(t?, n?, v?) \\
\hline
\end{array}$$

A set of artefacts may be inquired by type.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMgetArtefactsByType}} \\
\Xi RAM \\
t? : AType \\
as! : \mathbb{P}\, Artefact \\
\hline
as! = \{ tnv : \mathrm{dom}\, art \mid tnv\ conformsTo\ \{t?\} \bullet art\ tnv \} \\
\hline
\end{array}$$

A set of artefacts may be inquired by type and name.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMgetArtefactsByTypeAndName}} \\
\Xi RAM \\
t? : AType \\
n? : AName \\
as! : \mathbb{P}\, Artefact \\
\hline
as! = \{ tnv : \mathrm{dom}\, art;\ v : AVersion \mid tnv = (t?, n?, v) \bullet art\ tnv \} \\
\hline
\end{array}$$

A set of artefacts may be inquired by type, name, and version range.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMgetArtefactsByTypeNameAndVersionRange}} \\
\Xi RAM \\
t? : AType \\
n? : AName \\
vr? : \mathbb{P}\, AVersion \\
as! : \mathbb{P}\, Artefact \\
\hline
as! = \{ tnv : \mathrm{dom}\, art;\ v : vr? \mid tnv = (t?, n?, v) \bullet art\ tnv \} \\
\hline
\end{array}$$

Operations that modify the runtime artefact model do not change the set of supported types.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMOp}} \\
\Delta RAM \\
\hline
types' = types \\
\hline
\end{array}$$

A root may be installed into the model.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMInstallRootOk}} \\
\hline
\textit{RAMOp} \\
\textit{tnv}? : \textit{TNV} \\
a? : \textit{Artefact} \\
\hline
\textit{tnv}? \notin \operatorname{dom} \textit{art} \\
\textit{art}' = \textit{art} \cup \{\textit{tnv}? \mapsto a?\} \\
\textit{roots}' = \textit{roots} \cup \{\textit{tnv}?\} \\
\textit{dependsOn}' = \textit{dependsOn} \\
\textit{st}' = \textit{st} \cup \{\textit{tnv}? \mapsto \textit{INSTALLED}\} \\
\hline
\end{array}$$

Resolving a root involves a number of steps each of which operates on the model.

Artefacts may be installed to satisfy the transitive dependencies of roots.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMInstallOk}} \\
\hline
\textit{RAMOp} \\
\textit{tnv}? : \textit{TNV} \\
a? : \textit{Artefact} \\
\hline
\textit{tnv}? \notin \operatorname{dom} \textit{art} \\
\textit{art}' = \textit{art} \cup \{\textit{tnv}? \mapsto a?\} \\
\textit{roots}' = \textit{roots} \\
\textit{dependsOn}' = \textit{dependsOn} \\
\textit{st}' = \textit{st} \cup \{\textit{tnv}? \mapsto \textit{INSTALLED}\} \\
\hline
\end{array}$$

An individual dependency from one artefact in the model to another in the model may be recorded.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMRecordDependency}} \\
\hline
\textit{RAMOp} \\
\textit{from}? : \textit{TNV} \\
\textit{to}? : \textit{TNV} \\
\hline
\textit{from}? \in \operatorname{dom} \textit{art} \\
\textit{to}? \in \operatorname{dom} \textit{art} \\
\textit{art}' = \textit{art} \\
\textit{roots}' = \textit{roots} \\
\textit{dependsOn}' = \textit{dependsOn} \cup \{\textit{from}? \mapsto \textit{to}?\} \\
\hline
\end{array}$$

A set of artefacts may have their state set to a given value. This is an operation on a set of artefacts since, for example, marking an individual artefact's state as resolved is insufficient for handling circular dependencies.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMSetStateOk}} \\
\hline
\textit{RAMOp} \\
\textit{res}? : \mathbb{P}\, \textit{TNV} \\
s? : \textit{AState} \\
\hline
\textit{res}? \subseteq \operatorname{dom} \textit{art} \\
\textit{art}' = \textit{art} \\
\textit{roots}' = \textit{roots} \\
\textit{dependsOn}' = \textit{dependsOn} \\
\textit{st}' = \textit{st} \oplus (\textit{res}? \times \{s?\}) \\
\hline
\end{array}
$$

The following operation specifies the overall operation of installing and resolving a root using an input collection of dependencies.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMResolveRootOk}} \\
\hline
\textit{RAMOp} \\
\textit{tnv}? : \textit{TNV} \\
a? : \textit{Artefact} \\
\textit{deps}? : \textit{TNV} \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\; \textit{Artefact} \\
\hline
\textit{tnv}? \notin \operatorname{dom} \textit{art} \\
\\
\textit{art}' = \textit{art} \cup \{\textit{tnv}? \mapsto a?\} \cup \textit{deps}? \\
\textit{roots}' = \textit{roots} \cup \{\textit{tnv}?\} \\
\textit{st}' = \textit{st} \cup ((\{\textit{tnv}?\} \cup \operatorname{dom} \textit{deps}?) \times \{\textit{RESOLVED}\}) \\
\\
\textit{dependsOn} \subseteq \textit{dependsOn}' \\
\operatorname{dom} \textit{deps}? \subseteq \operatorname{ran} \textit{dependsOn}' \\
\hline
\end{array}
$$

The root must not already be in the model. The root and new dependencies are added to the model in *RESOLVED* state. The new dependencies are depended upon in the updated model.

5 Installing and Uninstalling Artefacts

A *repository* is a collection of artefacts each uniquely identified by type, name, and version.

$$Repository == TNV \rightarrowtail\!\!\!\!\!\!\!\shortmid\;\; Artefact$$

A kernel may be configured with a repository from which it can install artefacts. A kernel which is not configured with a repository is equivalent, at least from an install perspective, to a kernel configured with an empty repository.

So we model a kernel as having a runtime artefact model and a repository.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Kernel}} \\
\hline
RAM \\
repo : Repository \\
\hline
\end{array}$$

Kernel instances may share a repository or not, as shown in Figure 2.

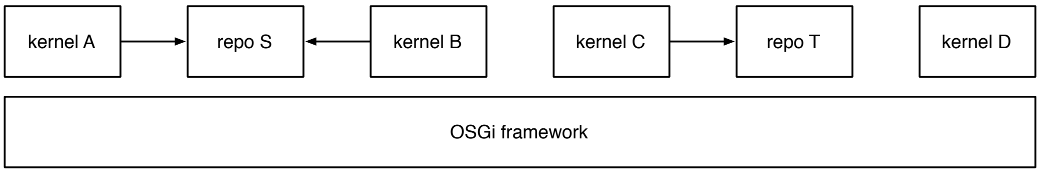

Figure 2: Kernels and repositories

We model kernel kernel instance creation as taking a repository parameter.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{KernelInit}} \\
\hline
Kernel' \\
RAMInit \\
repo? : Repository \\
\hline
repo' = repo? \\
\hline
\end{array}$$

Thereafter, the repository associated with a kernel can change in arbitrary ways. We summarise this by stating that the kernel has a notional operation which can update the kernel's repository arbitrarily. This operation is not implemented.

$$
\begin{array}{|l}
\hline
\textit{UpdateRepository} \\
\Delta Kernel \\
\Xi RAM \\
repo? : Repository \\
\hline
repo' = repo? \\
\hline
\end{array}
$$

Kernel operations do not modify the repository.

$$
\begin{array}{|l}
\hline
\textit{KernelOp} \\
\Delta Kernel \\
\hline
repo' = repo \\
\hline
\end{array}
$$

Certain operations on the runtime artefact model are promoted into corresponding kernel operations (although these will probably be grouped in a `RuntimeArtifactModel`[1] interface).

$$getRoots \mathrel{\widehat=} RAMgetRoots \land KernelOp$$

$$getArtefactTypes \mathrel{\widehat=} RAMgetArtefactTypes \land KernelOp$$

$$getArtefact \mathrel{\widehat=} RAMgetArtefact \land KernelOp$$

$$getArtefactsByType \mathrel{\widehat=} RAMgetArtefactsByType \land KernelOp$$

$$getArtefactsByTypeAndName \mathrel{\widehat=} \\ \quad RAMgetArtefactsByTypeAndName \land KernelOp$$

$$getArtefactsByTypeNameAndVersionRange \mathrel{\widehat=} \\ \quad RAMgetArtefactsByTypeNameAndVersionRange \land KernelOp$$

Since the contents of the repository may change during the lifetime of the kernel, we can think of a repository as being an input parameter to the install operation. Invariant relationships between the contents of the repository and the contents of the OSGi framework can only be maintained by a system which controls all the kernel instances sharing the OSGi framework and all the repositories[2] associated with those kernel instances. Any such invariants will also depend on the behaviour of any bundles which operate directly on the OSGi framework.

[1]American spelling is used in Java interfaces and javadoc.

[2]Retracting an artefact from a repository may cause the artefact's URI, that may have been previously obtained by a kernel, to become invalid and produce runtime errors in the kernel if the URI is used.

An install operation on the kernel takes a single root artefact. This may be supplied directly or as a reference to an artefact in the kernel's repository. The latter form is modelled below.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{InstallOkRaw}} \\
\mathit{KernelOp} \\
\mathit{RAMResolveRootOk} \\
\hline
\mathit{tnv}? \in \operatorname{dom} \mathit{repo} \\
\mathit{deps}? \subseteq \mathit{repo} \\
a? = \mathit{repo}\ \mathit{tnv}? \\
\mathit{tnv}? \in \mathit{resolved}' \\
\hline
\end{array}$$

The supplied artefact is installed *and* resolved.

$$\mathit{InstallOk} \mathrel{\widehat{=}} \mathit{InstallOkRaw} \setminus (a?, \mathit{deps}?)$$

Artefacts may have dependencies on other artefacts. For example, a plan may refer to a configuration file and a bundle may import a package which is exported by another bundle. Dependencies may be *optional* or *mandatory.* For example, an import of a package which specifies the directive `resolution:=optional` expresses an optional dependency on some bundle which exports that package.

When the kernel installs a root, it attempts to satisfy the dependencies of the root. Some of these dependencies may be satisfied by artefacts that were previously installed. The kernel attempts to satisfy any remaining dependencies by installing artefacts from the kernel's repository. These artefacts may also have dependencies which the kernel will attempt to satisfy, and so on. A root's dependencies, together with their dependencies, and so on are referred to collectively as the *transitive dependencies* of the root. If any *chain* of mandatory dependencies cannot be satisfied, then the installation of the root fails.

If the kernel does not have a repository, only (directly supplied) roots may be installed in the kernel and, before a root may be installed successfully, the root's transitive dependencies must already be installed.

If the installation of a root fails, any artefacts which were installed to satisfy the root's transitive dependencies must remain installed in case they are now depended upon by some other artefact that was installed concurrently and outside the control of the kernel.

To provide some level of clean-up after install failure, the transitive dependencies may be calculated and a check made that the OSGi bundles to be installed can resolve *before* the root or any of its transitive dependencies are installed in the OSGi framework. Since the OSGi framework may change concurrently with this calculation, the root installation may still fail after (correct) validation beforehand. This clean-up need not be fool-proof.

6 Starting and Stopping Artefacts

Installed artefacts may be started and later stopped any number of times and then, ultimately, uninstalled. Certain installed artefacts may be refreshed to pick up changes.

The main complexity surrounds when to start and stop the transitive dependencies of a root. Dependencies are those defined in the runtime artefact model and package dependencies defined by the OSGi framework. Service dependencies are too dynamic and unpredictable to be useful in this context.

Start

In general, artefacts in a dependency graph should be started in *reverse dependency order*. This means that artefacts are started, whenever possible, after the artefacts on which they depend. If there are cyclic dependencies, the artefacts in each cycle could be started in an arbitrary order, although it may be better to choose a deterministic order such as the order of installation.

Dependencies of plans are started in an order defined by the plan.

When a root is installed, any implicit transitive dependency (that is, a dependency that need not be defined in the runtime artefact model) causes the artefact satisfying the dependency to be started *before* the installation of the root completes.

Before a root is started, the kernel starts the transitive dependencies, in reverse dependency order.

Starting an artefact which is in `ACTIVE` state has no effect.

Starting a root starts all its dependencies.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAMstartRootOk}} \\
\hline
\textit{RAMOp} \\
\textit{tnv}? : \textit{TNV} \\
\hline
\textit{tnv}? \in \textit{resolved} \\
\textit{art}' = \textit{art} \\
\textit{roots}' = \textit{roots} \\
\textit{dependsOn}' = \textit{dependsOn} \\
\textit{st}' = \textit{st} \oplus ((\textit{dependsOn}^{*}(\!|\ \{\textit{tnv}?\}\ |\!)) \times \{\textit{ACTIVE}\}) \\
\hline
\end{array}
$$

Stop

Before a root is stopped, the kernel stops certain transitive dependencies, in dependency order.

The kernel traverses the root's dependencies in the runtime artefact model and ignores dependencies in the OSGi framework. Also, the kernel does not stop transitive dependencies which are depended on, in the runtime artefact model, by other active artefacts.

Stopping an artefact which is in `INSTALLED` or `RESOLVED` state has no effect.

Stopping a root stops all its unique dependencies.

$$
\begin{array}{|l}
\textit{RAMstopRootOk} \\
\hline
\textit{RAMOp} \\
\textit{tnv}? : \textit{TNV} \\
\hline
\textit{tnv}? \in \operatorname{dom} \textit{art} \\
\textit{st}\ \textit{tnv}? = \textit{ACTIVE} \\
\textit{art}' = \textit{art} \\
\textit{roots}' = \textit{roots} \\
\textit{dependsOn}' = \textit{dependsOn} \\
(\textit{dependsOn}^* (\!| \{\textit{tnv}?\} |\!)) \lhd \textit{st} = (\textit{dependsOn}^* (\!| \{\textit{tnv}?\} |\!)) \lhd \textit{st}' \\
\textit{st}'\ \textit{tnv}? = \textit{RESOLVED} \\
\hline
\end{array}
$$

Kernel Operations

These operations on the runtime artefact model are promoted into corresponding kernel operations (although these will probably be grouped in a `Artifact` interface).

$$\textit{startRootOk} \mathrel{\widehat{=}} \textit{RAMstartRootOk} \land \textit{KernelOp}$$

$$\textit{stopRootOk} \mathrel{\widehat{=}} \textit{RAMstopRootOk} \land \textit{KernelOp}$$