Repository matters

Steve Powell and Andy Wilkinson

April 24, 2009

We define various aspects of repository and repository chain configuration and behaviour.

This work is performed under the JIRA items `DMS-347` and `DMS-340`.

Contents

1 Basic types and functions

Repositories will be named and resources (which may be stored artefacts, or files, or remote repositories) will be identified by URI.

Artefacts stored in a repository will require a storage position—this also uses a URI—and other properties, like names, types and versions.

There is a structure on versions which is exposed, to a certain extent, in the handling of repository queries. This is encapsulated in a function which chooses the 'maximum' version from (some) sets of versions. This sort of abstraction allows us to avoid having to describe intricate details of version syntax, or indeed of version range syntax.

1.1 Basic types

These items are fundamental terms:

$[RepoName, URI]$

We will wish to identify a place in the file system where artefacts are stored. This is also a URI. There are file path *patterns* for describing what files in an external directory to read. The pattern is only accepted for an 'external' repository—not a 'managed' nor a 'watched' repository. These terms will be described later.

$[StoredArtefactPattern]$

$$\begin{aligned} &StoredArtefactArea \\ &\quad ::= saROOT\langle\langle URI\rangle\rangle \\ &\quad \mid saPATTERN\langle\langle StoredArtefactPattern\rangle\rangle \end{aligned}$$

The syntax and form of a pattern is not specified here but we will use the ANT file pattern rules.

There are also properties, but we will not need to explain them in the formal specification.

$[Properties]$

1.2 Artefact basics

There are artefact descriptors, which essentially identify the artefact bytes with a type, a name and a version, and some derived meta-data. All these

fundamental types will be used in our description of a repository.

$$[Type, Name, Version, MetaData]$$

$$TNV == Type \times Name \times Version$$

We use the abbreviation TNV in the repository descriptions below, and the individual types in defining the (definitive) descriptor of an artefact:

ArtefactDescriptor

$type : Type$
$name : Name$
$version : Version$
$uri : URI$
$metadata : MetaData$

There are no constraints that can be expressed at this level of abstraction. (In particular, we cannot refer to a file store or a remote system to determine if the *uri* really indicates a resource that matches these *type*, *name*, *version* and *metatdata* values.) The *uri* is the link from this description to the actual files on disc—here is where to go for the artefact binaries.[1]

We will want to extract parts of an artefact descriptor at various times; here are the projections to do that:

$$typeOf == (\lambda\, ArtefactDescriptor \bullet type)$$
$$nameOf == (\lambda\, ArtefactDescriptor \bullet name)$$
$$versionOf == (\lambda\, ArtefactDescriptor \bullet version)$$
$$uriOf == (\lambda\, ArtefactDescriptor \bullet uri)$$
$$tnvOf == (\lambda\, ArtefactDescriptor \bullet (type, name, version))$$
$$nonuriOf == (\lambda\, ArtefactDescriptor \bullet (type, name, version, metadata))$$

1.3 Version types

There are some manipulations we will want to perform on versions, and sets of versions. In particular, we have the notion of a version range:

$$VersionRange == \mathbb{P}\, Version$$

which may represent a (theoretically) infinite collection, and a function which picks out the highest version from a (finite) non-empty version range:

[1] Note that this *uri* is not necessarily the one that the artefact was sourced from: some repositories copy the artefact into their own storage (and so have a new *URI*) and some do not. See the *PublishArtefact* operation, later.

$$
\begin{array}{|l}
maxv : VersionRange \pfun Version \\
\hline
\dom maxv = \mathbb{F}_1\ Version \\
\forall\, vr : \dom maxv \bullet (maxv\ vr) \in vr
\end{array}
$$

Here we do not fully specify this function which depends upon a well-understood total order on *Version*s.

2 Repositories

2.1 Repository state

A *Repository* is essentially a map from artefact types, names and versions to full artefact descriptions:

$$
\begin{array}{|l}
\textit{Repository} \\
\hline
artems : TNV \pinj ArtefactDescriptor \\
arteuri : URI \pinj TNV \\
\hline
tnvOf \circ artems = \mathrm{id}(\dom artems) \\
arteuri = (artems \fatsemi uriOf)^\sim \\
\hline
\end{array}
$$

The first constraint, on *artems*, says that the type, name and version used to index a descriptor are actually embedded in the descriptor indexed (no mismatches occur).

We have added a derived component of the state (*arteuri*) which, by declaring it also to be an injection, imposes the rule that the *uri* component of an artefact descriptor must uniquely identify a descriptor in the repository. The second constraint shows that the *arteuri* map is fully derived from *artems*. The derivation shows that the *URI* used to index the artefacts in *arteuri* is that stored in the artefact descriptor so indexed.

These constraints are *state invariants*, to be preserved by all operations on the state.

2.2 Repository initialisation

Initialising a repository is to populate the map *artems* consistently. This is done differently for the different implementations of *Repository*, so we will defer this operation until the implementations are discussed.

2.3 Repository operations

The operations on a (general) repository are quite simple, and involve modifying the *artems* map. Because the *arteuri* map is derived from *artems* we need not specify changes to it; however, the fact that *arteuri* is an injection does impose some not-so-obvious pre-conditions on the operations. We will mention these in each case.

2.3.1 Get artefact

To get an artefact we need to ask the repository about it, which means asking the repository for the artefact descriptor. This is a simple look-up, except that we might give a range of acceptable versions we would like. The version we get is the highest (*maxv*) that there is in the repository.

GetArtefactOK

$$
\begin{array}{l}
\Xi Repository \\
t? : Type \\
n? : Name \\
vr? : VersionRange \\
ad! : ArtefactDescriptor \\
\hline
ad! = \\
\quad (\textbf{let}\ mv == maxv(versionOf(\!|\ artems(\!|\ \{\ t?\ \} \times \{\ n?\ \} \times vr?\ |\!)\ |\!)) \\
\quad \bullet\ artems(t?, n?, mv))
\end{array}
$$

This rather imposing constraint simply chooses the highest version artefact with that type and name that exists in the repository and has a version in the range supplied. *ad*! is the resulting artefact descriptor.

We have called this *GetArtefactOK* because it describes the result only if the pre-condition is satisfied—that there is an artefact that fits the bill therein. If not, this operation specification says nothing. We will augment these operations with failure cases later.

Observe that the repository does not change.

2.3.2 Query artefacts

Considerably more complex retrievals from the repository are required, but this interface will not support a complex query language. Instead we will use a query object, defined elsewhere, which will identify a set of artefact descriptors that fit the bill. When that query object is applied to a repository a set of available artefacts that satisfy the query is returned. The query

object is represented here as a set—a subset of *ArtefactDescriptor*—which may very well be (theoretically) infinite. Application to the repository guarantees a finite result (since *artems* is a finite injection) and this is returned. The latter can therefore be represented as a simple enumeration, whereas the query itself is probably a structural representation of a boolean function on artefact descriptors. We will defer deciding the actual format by requiring the repository itself to construct the query objects: this gives the repository the opportunity to tailor the query representation to the repository implementation, if we wish.

The key observation here is what the query object is a subset *of*. Only information in the artefact descriptor is to be relevant in the query. In fact, we might have been even more proscriptive[2]: only the type, name, version and meta-data information ought to be used. The *URI*, and in particular the form of the *URI*, should not have any influence on the query function applied. This observation, though expressible here, is rather awkward to tease out (without reconstructing the specification state somewhat) so we leave this as understood.

A repository query is a subset of the set of all possible *ArtefactDescriptor*s:

$$RepositoryQuery == \mathbb{P}\, ArtefactDescriptor$$

and the operation to 'run' the query against the repository is *QueryArtefactsOK*:

QueryArtefactsOK

$$\Xi Repository$$
$$q? : RepositoryQuery$$
$$ads! : \mathbb{F}\, ArtefactDescriptor$$
$$ads! = q? \cap \operatorname{ran} artems$$

We have chosen here to return a simple set of artefact descriptors.

An alternative, which might be more suitable, is to return a partial map of the results:

AltQueryArtefacts

$$\Xi Repository$$
$$q? : RepositoryQuery$$
$$subarte! : TNV \pfun ArtefactDescriptor$$
$$subarte! = artems \rhd q?$$

[2]proscribe—*v. tr.* forbid, esp. by law: 'strikes remained proscribed in the armed forces'; denounce or condemn: 'certain practices which the Catholic Church proscribed, such as polygyny'; (*hist.*) outlaw (someone).

Probably a much more useful result. Of course these are logically indistinguishable—the type, name and version information can be unambiguously recovered from the artefact descriptor, as the *Repository* state ensures.

2.3.3 Publish artefact

Having described innocuous operations that leave the repository unchanged, we wish to describe the modifying operations: publish and retract.

Not every repository type supports publish and retract. Those that do should adhere to this behaviour.

Publishing an artefact to the repository requires that a reference to the artefact be supplied. This takes the form of a *URI*. The information gleaned from the *URI* will be used to populate the repository map, although it has to be realised that the URI itself need not be referenced in the repository. Thus although a *URI* is passed in, there is no reason to suppose that the same *URI* is passed out.

The definition of *PublishArtefact* is therefore rather abstract and non-deterministic:

PublishArtefactOK

$$\Delta Repository$$
$$uri? : URI$$
$$ad! : ArtefactDescriptor$$

$$artems' = artems \oplus \{ (typeOf\ ad!, nameOf\ ad!, versionOf\ ad!) \mapsto ad! \}$$

Some artefact descriptor is returned. Although it has to be a consistent addition to *artems* to qualify for return, there is nothing to enforce a relationship with the value of *uri*? passed in. In particular, there is no necessary relationship between *uri*? and *uriOf ad*!. In fact, they will be equal for some types of repository, though for others they will not.

When we come to describe possible failures for this operation, we will realise that the result is non-deterministic in this respect also. In fact, this is to be expected—we cannot guarantee what may be found when a *URI* is 'followed up' at run-time—*there's mony a slip 'tween cup 'n' lip.*

Despite appearances this operation doesn't say nothing at all—it crucially explains in what way the repository may change as a result of a publish—nothing is lost from the repository; something may be added; that something need not have the same *URI* as supplied (though it has to be distinct from the other *URI*s already in the repository, as the *Repository* state dictates).

Notice that *overwrite* is used in this definition. It is entirely possible that a publish operation replaces (or hides) an existing artefact. In general, this cannot be prevented.

2.3.4 Retract artefact

Not every repository type supports retract and publish. Those that do should adhere to this behaviour.

There are, potentially, two ways to identify an artefact to remove from the repository. The obvious way is to use the (main) index—TNV. However, not all clients of the repository need know the canonical identification of the artefact they wish to remove. It is possible that the *URI* is known instead.

We describe two retract operations, one which removes an artefact given the *Type*, *Name* and *Version* of it, and another which removes an artefact identified by its *URI*. Note that the *URI* operation requires the *URI* of the artefact *in the repository* and not the original *URI* as supplied on the *Publish* operation.

Retract an artefact by name by supplying the TNV for the artefact:

RetractArtefactByNameOK

$$\Delta Repository$$
$$t? : Type$$
$$n? : Name$$
$$v? : Version$$
$$ad! : ArtefactDescriptor$$

$$(t?, n?, v?) \in \mathrm{dom}\ artems$$
$$\{ (t?, n?, v?) \} \lhd artems' = \{ (t?, n?, v?) \} \lhd artems$$
$$nonuriOf\ ad! = nonuriOf\ (artems\ (t?, n?, v?))$$

The precondition is that the artefact identification actually identifies a known artefact. The repository is not affected except (possibly) on that artefact instance. An artefact descriptor is returned, though the *uri* in it means nothing (we probably do not return one).

Care needs to be taken here—just because the artefact is successfully removed, doesn't mean it isn't still there. This paradoxical result is due to the fact that the artefact removed may have 'hidden' another (with identical TNV) when it was published (this tends to happen with chains). The removal will not then have any perceptible effect—although the artefact URI

and binaries might indeed be different when we look it up. (Of course, this can be different each time we look anyway.) We have documented these subtleties by being careful to say nothing about the contents of the resulting repository *under the TNV key retracted.*

It is also possible that removal fails. We deal with that later.

Notice there is no indication that the artefact binaries (located by *uriOf ad*!) are touched in any way. For some repositories they are not, for others they may be (or have been) deleted.

Retract an artefact by *URI* by supplying the *repository's uri*? for the artefact:

RetractArtefactByUriOK

$$\Delta Repository$$
$$uri? : URI$$
$$ad! : ArtefactDescriptor$$

$$uri? \in \mathrm{dom}\ arteuri$$
$$\{ uri? \} \lhd arteuri' = \{ uri? \} \lhd arteuri$$
$$nonuriOf\ ad! = nonuriOf(artems\ (arteuri\ uri?))$$

The *uri*? must identify one of the artefacts in the repository. The repository is not affected except (possibly) on that artefact instance. An artefact descriptor is returned, though the *uri* in it means nothing (we probably do not return one).

The same subtleties are observed here as for the *TNV* case: there is no guarantee that the artefact is now absent from the repository, though we do guarantee that no other artefact is affected. The artefact descriptor returned has no relevant *URI*.

2.4 Complete operations

To describe the failure conditions, we can either leave them implicit (and choose to throw an exception, say—though this leaves the state invariants open to doubt) or we can make them explicit as far as is possible. We choose the latter, being aware that a 'return code' need not be a return value in actual code. The point here is to document what happens if preconditions fail (or non-deterministically), what states result, and what information is available in those cases.

We need to introduce some indicator of failure and success:

$$
\begin{aligned}
&\mathit{ReturnCode} ::= \\
&\quad \mathit{OK} \mid \\
&\quad \mathit{UnknownUri}\langle\!\langle \mathit{URI} \rangle\!\rangle \mid \\
&\quad \mathit{UnknownId}\langle\!\langle \mathit{TNV} \rangle\!\rangle \mid \\
&\quad \mathit{UnknownIds}\langle\!\langle \mathit{Type} \times \mathit{Name} \times \mathit{VersionRange} \rangle\!\rangle \mid \\
&\quad \mathit{CannotPublishUri}\langle\!\langle \mathit{URI} \rangle\!\rangle \mid \\
&\quad \mathit{IOReadFailureUri}\langle\!\langle \mathit{URI} \rangle\!\rangle \mid \\
&\quad \mathit{IOReadFailureId}\langle\!\langle \mathit{TNV} \rangle\!\rangle \mid \\
&\quad \mathit{IOWriteFailureUri}\langle\!\langle \mathit{URI} \rangle\!\rangle \mid \\
&\quad \mathit{IOWriteFailureId}\langle\!\langle \mathit{TNV} \rangle\!\rangle
\end{aligned}
$$

$$\mathit{Rc} \mathrel{\widehat=} [\, \mathit{rc}! : \mathit{ReturnCode} \,]$$

$$\mathit{RcOK} \mathrel{\widehat=} [\, \mathit{Rc} \mid \mathit{rc}! = \mathit{OK} \,]$$

The return code spectrum indicates what the error information contains; *RcOK* being short for everything worked (the precondition of the operation core being satisfied).

We then use some abbreviations for signatures and generate state change promises:

$$\mathit{OpRc} \mathrel{\widehat=} \Delta \mathit{Repository} \land \mathit{Rc}$$
$$\mathit{StetRc} \mathrel{\widehat=} \Xi \mathit{Repository} \land \mathit{Rc}$$

$$\mathit{SigUri} \mathrel{\widehat=} [\, \mathit{uri}? : \mathit{URI} \,]$$
$$\mathit{SigId} \mathrel{\widehat=} [\, t? : \mathit{Type};\ n? : \mathit{Name};\ v? : \mathit{Version} \,]$$
$$\mathit{SigIdRan} \mathrel{\widehat=} [\, t? : \mathit{Type};\ n? : \mathit{Name};\ vr? : \mathit{VersionRange} \,]$$

and we can succinctly define the error cases. First the 'not known' (deterministic) cases:

$$
\begin{aligned}
\mathit{RcUnknownIds} \mathrel{\widehat=} [\, & \mathit{StetRc};\ \mathit{SigIdRan} \\
& \mid (\{t?\} \times \{n?\} \times vr?) \cap \operatorname{dom} \mathit{artems} = \emptyset \\
& \land\ \mathit{rc}! = \mathit{UnknownIds}(t?, n?, vr?) \,]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{RcUnknownId} \mathrel{\widehat=} [\, & \mathit{StetRc};\ \mathit{SigId} \\
& \mid (t?, n?, v?) \notin \operatorname{dom} \mathit{artems} \\
& \land\ \mathit{rc}! = \mathit{UnknownId}(t?, n?, v?) \,]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{RcUnknownUri} \mathrel{\widehat=} [\, & \mathit{StetRc};\ \mathit{SigUri} \\
& \mid \mathit{uri}? \notin \operatorname{dom} \mathit{arteuri} \\
& \land\ \mathit{rc}! = \mathit{UnknownUri}\ \mathit{uri}? \,]
\end{aligned}
$$

and now the non-deterministic cases:

$$RcIOStetIdRange \mathrel{\widehat=} [\, StetRc;\ SigIdRan \mid \exists\, v : vr? \bullet rc! = IOReadFailureId(t?, n?, v) \,]$$
$$RcCannotPublishUri \mathrel{\widehat=} [\, StetRc;\ SigUri \mid rc! = CannotPublishUri\ uri? \,]$$

$$RcIOStetUri \mathrel{\widehat=} [\, StetRc;\ SigUri \mid rc! = IOReadFailureUri\ uri? \,]$$
$$RcIOFailUri \mathrel{\widehat=} [\, OpRc;\ SigUri \mid rc! = IOWriteFailureUri\ uri? \,]$$
$$RcIOUri \mathrel{\widehat=} RcIOStetUri \lor RcIOFailUri$$

$$RcIOStetId \mathrel{\widehat=} [\, StetRc;\ SigId \mid rc! = IOReadFailureId(t?, n?, v?) \,]$$
$$RcIOFailId \mathrel{\widehat=} [\, OpRc;\ SigId \mid rc! = IOWriteFailureId(t?, n?, v?) \,]$$
$$RcIOId \mathrel{\widehat=} RcIOStetId \lor RcIOFailId$$

Notice the complete state flapping in the *RcIOFail...* cases.

We proceed to apply them to define the complete operation specs as follows:

Get only fails deterministically if the artefact is unknown, or non-deterministically if there is some (non-destructive) access or read failure:

$$GetArtefact \mathrel{\widehat=} (GetArtefactOK \land RcOK) \lor RcUnknownIds \lor RcIOStetId$$

Publish only fails for its own reasons (this is where an unrecognised artefact would surface):

$$PublishArtefact \mathrel{\widehat=} (PublishArtefactOK \land RcOK) \lor RcCannotPublishUri \lor RcIOUri$$

If there is a read error then the *uri*? is returned and the state does not change; if there is a write error then the resulting repository state is undetermined.

The return information from *PublishArtefact* is a little sparse—we would have liked to return information about duplication, if detected; but there is nothing to say at this level of abstraction, although we at least have described the sort of errors one might get.

Retract by name fails if the artefact is unknown (by that type, name and version), or if some IO failure occurs:

$$RetractArtefactByName \mathrel{\widehat=} (RetractArtefactByNameOK \land RcOK) \lor RcUnknownId \lor RcIOId$$

Retract by *URI* fails if the artefact is unknown (by that *uri*?) or if some IO failure occurs:

$$RetractArtefactByUri \mathrel{\widehat=} (RetractArtefactByUriOK \wedge RcOK) \vee RcUnknownUri \vee RcIOUri$$

Query artefacts will not fail:

$$QueryArtefacts \mathrel{\widehat=} QueryArtefactsOK \wedge RcOK$$

We have not augmented the operation *QueryArtefacts*, since this can always return an empty set of results—we anticipate no errors from this operation.

The return information from *PublishArtefact* is a little sparse—we would have liked to return information about duplication, if detected; but there is nothing to say at this level of abstraction, although we at least have described the sort of errors one might get.

3 Repository types

We have already mentioned that repositories come in several types. These are:

external this repository reads artefact files which are generated by other programs, and does not modify or move them in any way—they are treated as read-only;

managed this repository creates and manages all its own artefact files;

watched this repository adds (publishes) or removes (retracts) files automatically based upon a 'watched' directory—the file operations trigger addition or removal of the stored artefacts from the repository index;

remote this is a 'proxy' for a remote (hosted) repository—the artefacts and indexes are managed by a separate system;

chained this is a 'compound' repository, which manages an ordered sequence of other repositories in such a way as to support the repository interfaces.

4 Repository configuration

In a *dm Server* there are *local* repositories, which are defined locally and accessed directly by the current server, and *remote* repositories, defined elsewhere and accessed by communication with a remote (dm Repository) server. Each repository has a named definition, which identifies its artefact store (a file path), its name, and various properties useful for management of the artefacts.

We identify five types of repository: a 'managed' repository—the artefacts are created, stored and held by the repository; an 'external' repository—the artefacts are created and destroyed by another program; a 'watched' repository—the artefacts are created and destroyed by file system operations in a (flat) directory structure; a 'remote' repository—the artefacts aren't stored here, but are managed by another server; and a 'chain' repository—a compound repository made up from a sequence of other (non-chain) repositories.

A repository definition is named, and has properties. Other than that there is little in common between the types. Therefore we define a base, which all the other types 'inherit':

$$
\begin{array}{|l}
\hline
\mathit{RepoDefnBase} \\
\mathit{rName} : \mathit{RepoName} \\
\mathit{rProps} : \mathit{Properties} \\
\hline
\end{array}
$$

For each of the types of repository we define an extension of these data:

$$
\begin{aligned}
&\mathit{ManagedRepoDefn} \mathrel{\widehat=} [\, \mathit{root} : \mathit{URI} \,] \\
&\mathit{ExternalRepoDefn} \mathrel{\widehat=} [\, \mathit{pattern} : \mathit{StoredArtefactPattern} \,] \\
&\mathit{WatchedRepoDefn} \mathrel{\widehat=} [\, \mathit{dir} : \mathit{URI} \,] \\
&\mathit{RemoteRepoDefn} \mathrel{\widehat=} [\, \mathit{remoteUri} : \mathit{URI} \,] \\
&\mathit{ChainRepoDefn} \mathrel{\widehat=} [\, \mathit{chain} : \mathrm{iseq}\, \mathit{RepoName} \,]
\end{aligned}
$$

We can now define what a repository definition logically looks like, starting with the basic data:

$$
\begin{aligned}
&\mathit{RepoDefnData} ::= \\
&\quad \mathit{MANAGED}\langle\!\langle \mathit{ManagedRepoDefn} \rangle\!\rangle \mid \\
&\quad \mathit{EXTERNAL}\langle\!\langle \mathit{ExternalRepoDefn} \rangle\!\rangle \mid \\
&\quad \mathit{WATCHED}\langle\!\langle \mathit{WatchedRepoDefn} \rangle\!\rangle \mid \\
&\quad \mathit{REMOTE}\langle\!\langle \mathit{RemoteRepoDefn} \rangle\!\rangle \mid \\
&\quad \mathit{CHAINED}\langle\!\langle \mathit{ChainRepoDefn} \rangle\!\rangle
\end{aligned}
$$

and then defining the definition form:

$$RepositoryDefn \mathrel{\widehat=} [\, RepoDefnBase;\ defnData : RepoDefnData \,]$$

Before defining a configuration we explain some basic functions on definition data, some projections:

$$rNameOf == (\lambda\, RepositoryDefn \bullet rName)$$
$$rPropsOf == (\lambda\, RepositoryDefn \bullet rProps)$$

and a partial function for extracting chains:

$$rChainOf : RepositoryDefn \pfun \operatorname{iseq} RepoName$$
$$rChainOf = (\lambda\, d : RepositoryDefn \mid d.defnData \in \operatorname{ran} CHAINED \bullet (CHAINED^{\sim}\ d.defnData$$

and then extending to a complete configuration, which is a (consistent) collection of definitions with constraints:

RepositoryConfig

$$defns : \mathbb{F}\, RepositoryDefn$$

5 Chain configuration reified

We give examples of these definitions as `JSON` files.

In a repository configuration file we might see definitions like this:

```
"repositories": {
  "bundles-ext": {
    "type" : "external",
    "searchPattern": "repository/bundles/ext/*.jar"
  },
  "bundles-usr": {
    "type" : "external",
    "searchPattern": "repository/bundles/usr/*.jar"
  },
  "libraries-ext": {
    "type" : "external",
    "searchPattern": "repository/libraries/ext/*.jar"
  },
  "libraries-usr": {
```

```
    "type" : "external",
    "searchPattern": "repository/libraries/usr/*.jar"
  },
  "managed-repo" : {
    "type" : "managed",
    "storageDirectory" : "repository/managed"
  },
  "watched-repo" : {
    "type" : "watched",
    "watchDirectory" : "repository/watched",
    "watchInterval" : 5
  },
  "remote-repo" : {
    "type" : "remote",
    "uri" : "http://localhost:8080/com.springsource.repository/foo",
    "indexRefreshInterval" : 5
  }
}
```

where we emphasise that the order of the definitions is not relevant and that the repository names are distinct within the configuration file. We envisage one such definition file per dm Server.

The repository chain can then be a sequence, for example:

```
"repositoryChain": [
  "bundles-ext",
  "bundles-usr",
  "watched-repo",
  "remote-repo"
]
```

in this case of two local repositories followed by a watched (local) repository and a remote repository. Notice that not all the defined repositories need be used.

Actual chain construction is carried out programmatically—the list of configurations *names* is obtained (from the configuration file?), and the configurations are looked-up (by name) and passed (as a list) to construct the repository chain itself.

6 Repositories and artefact stores

Every repository has an artefact store from which it obtains its artefacts. The store contains files which separately or in combination constitute the binaries from which an artefact is deployed. Normally an artefact is stored at a *File* URI, though it can be any URI and we exploit this to allow *hosting*really.

A file URI can be owned by the repository or not. If owned by the repository then the actions of publish and retract become significant. We design a repository (and an interface) that can be placed in a chain and can support publish and retract operations (either implicit or explicit) into the artefact store.

6.1 Artefact stores

Artefacts are stored. These are normally files or collections of files, each of which may be identified by a (single) URI. We do not need to expose the files in this specification, the artefact descriptor (above) is sufficient.

An artefact store can be regarded simply as a map from stored artefact URIs to the stored artefacts themselves. With this we can specify an artefact store (which is essentially a piece of a file system):

ArtefactStore

$$aStore : URI \pfun ArtefactDescriptor$$

$$aStore^{\sim} \subseteq uriOf$$

The map *aStore* is injective, which implies that distinct stored artefacts have distinct URIs, and the URI for any stored artefact is entirely dictated by the file system. This means that the artefact store is essentially just a set of stored artefacts, indexed by URI.

There are three types of artefact store, corresponding to the three ways in which the *aStore* is initialised and managed.

6.1.1 External artefact store

The external artefact store is part of a file system containing stored artefacts. The files are not managed by the artefact store, the store merely keeps information about the stored artefacts in it. It is initialised by supplying a file store pattern which is used to populate the map from a given file system and it supports an operation *GetURIs* that returns all the URIs.

The state of the external artefact store includes the pattern.

ExternalArtefactStore

$$ArtefactStore$$
$$pattern : StoredArtefactPattern$$

Initialising the *ExternalArtefactStore* involves setting the pattern, and populating the map using the stored artefacts found there. We do not attempt to show the dependency upon a file system that is evident—we hide this in a function that maps a pattern to a collection of stored artefacts that match it:

$$filesOf : StoredArtefactPattern \pfun \mathbb{P}\, ArtefactDescriptor$$

Initialising the store gets the matching files and populates the *aStore* by reference to the file system:

InitExternalArtefactStore

$$ExternalArtefactStore'$$
$$pattern? : StoredArtefactPattern$$

$$pattern' = pattern?$$
$$aStore'^{\sim} = (filesOf\ pattern?) \rhd uriOf$$

Operations on the ExternalArtefactStore do not ever modify the pattern:

ExternalArtefactOp

$$\Delta ExternalArtefactStore$$

$$pattern' = pattern$$

and we can document the (trivial) query operation, which doesn't modify anything:

GetURIs

$$\Xi ExternalArtefactStore$$
$$storedArtefactURIs! : \mathbb{P}\, URI$$

$$storedArtefactURIs! = \mathrm{dom}\ aStore$$

6.1.2 Managed artefact store

The managed artefact store manages the stored artefacts itself, and therefore is not initialised with a pattern. It supports *store* and *remove* operations as well as query.

6.1.3 Watched artefact store

7 Conversational notes

(*This section is for future expansion and is 'in transit'.*)

A repository is given a (single) file store place for artefacts.

The repositories have three types: external, managed and watched.

An *external* repository is read/scanned and artefacts can be installed from it, but publish and retract is *not* supported. This means that the only way to modify this is by changing the file store directly. One application of this is to have a pre-populated repository for first use. Another use is to point to a Maven directory so that artefacts can be built and tested in development easily.

A *managed* repository is wholly managed by the server, and the server allows modification of the artefacts of the repository by publish and retract. The internal structure and content of a managed repository is entirely the responsibility of the server.

A *watched* repository cannot be modified by publish and retract, but by file creation (move/copy) and file deletion (move/delete). This is precisely the pickup directory semantics. File store watching is then a matter for the repository implementation which raises repository events to the rest of the system (kernel/deployer).

Repositories can be labelled 'autodeploy', which, for a watched repository means that the repository event triggers full deploy, that is, install and start. For an external repository this would imply that all artefacts in the repository are installed/started when the kernel is started. Autodeploy might also work for a mutable (managed) repository, which would imply deploy on publish. The autodeploy option is only relevant for a repository in a chain—should we put the autodeploy option on a (local) repository?

'Autodeploy' is not recorded here, it appears to be a feature of the deployer and not the repository, nor the chain.

Repositories raise events when things happen to them. The deployer (in the kernel) can listen for these. When an artefact is published in a repository, for example, the repository can raise the 'publish' event, identifying the artefact (by type, name and version) and the repository in which it is published (by name in the repository definition). This only makes sense if the local deployer/server is listening to the events. At this point the deployer has enough information to execute an auto-deploy—if this is configured. There remains the thorny problem of a pickup directory being lower in the chain

than a repository that has the artefact already in it. With these data on the event it is possible for the deployer (the user of the chain) to determine if the artefact in question would be installed. This allows it to correctly diagnose (and report) 'shadowed' artefacts, and not attempt to deploy them.

This reasoning also applies to retracts of artefacts. The repository and the artefact identification is reported on the event, and the deployer looks to see if this is deployed or not. The retract can then trigger an automatic stop and uninstall of the artefact if necessary, or ignore it if not.

The pickup directory (watched) gives strange problems, and in general publish of an artefact to a watched repository has issues. For example, if a bundle is published to a watched repository but that bundle is already installed from another repository in the chain, what happens? If the bundle is visible (from the chain) then can we install it (after un-installing the old one?) if the bundle is not visible, perhaps we should issue a warning, or error – we certainly cannot 'install' it, because then when the item is deleted we might actually un-install the one that is earlier in the chain! This is counter-intuitive and needs careful thought.