Unclassified, Draft

A Formal Model of OSGi R4 Modularity (v0.83)

Glyn Normington

November 6, 2008

The aim is to model a subset of the features of the modularity layer of OSGi Release 4.

1 Introduction

Probably omit this from the paper, although there needs to be 'warning' of the formal stuff near the start of the paper and in the abstract.

This specification only models a subset of the modularity function of OSGi R4. It does not model:

- 'uses'
- optional resolution
- dynamic imports
- export filters
- require-module

2 UML Overview

Probably omit this from the paper.

Figure 1 shows the main artefacts of OSGi modularity, some of which will be covered by the formal model.

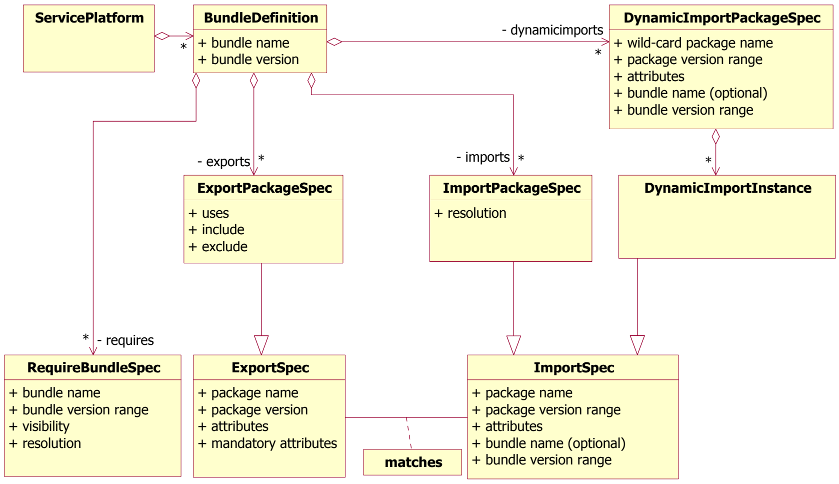

Figure 1: UML Overview

There are three new concepts in the diagram which are purely to explain the other "classes" which relate directly to RFC 79 manifest headers.

ExportSpec is a piece of the "export signature" of a module which is a generalisation of `Export-Package`. What the diagram doesn't show is that `Require-Bundle` with `visibility:=reexport` also produces these abstract exports.

ImportSpec is a kind of generalisation of `Import-Package` and `DynamicImport-Package`. AbstractImport really does generalise `Import-Package`. But `DynamicImport-Package` can, because of its wild-carded package name, be thought of as a kind of macro or template which generates (at class load time) "dynamic import instances", described next.

DynamicImportInstance is effectively an import which is derived, at class load time, from a dynamic import. AbstractImport generalises DynamicImportInstance.

The crux of the diagram is that there is a “matches” relationship between AbstractImport and AbstractExport. This relationship describes potential package wirings - RFC 79 has to cover the cases when an import matches more than one export.

3 Basic Types

Insert this section in section 3 of the paper.

Some basic types need defining.

Module Names

Modules are identified within a module system by module name and module version number.

$$[MName, MVer]$$

Module version, consisting of the name and version number of a module, needs an abbreviation.

$$MV == MName \times MVer$$

Packages

We need the notion of packages (or, more precisely, package names) and package version numbers.

$$[Package, PVer]$$

Package version, consisting of the name and version number of a package, needs an abbreviation.

$$PV == Package \times PVer$$

We need the notion of (fully qualified) class names and classes.

$$[ClassName, Class]$$

Each class name belongs a unique package.

$$\mid\ package : ClassName \rightarrow Package$$

Certain classes belong to packages beginning with “java.” (and “org.omg.” etc.).

$$\mid\ javaClasses : \mathbb{P}\ ClassName$$

Class Spaces and Consistency

A class space is a collection of classes indexed by class name.

$$ClassSpace == ClassName \rightarrowtail\!\!\!\!\!\!\rightarrow Class$$

We say that two class spaces s and t are *consistent* if and only if

$s \cup t \in ClassSpace$

, i.e. s and t agree on the class names they have in common.

Attributes

Arbitrary attribute names and values will be used to provide a flexible import/export matching mechanism.

$$[AName, AValue]$$

We will use mappings of attribute names to values.

$$AMap == AName \pfun AValue$$

Module names are sometimes used as attribute names.

$$\mid \; MNameAttr : AName$$

4 Simple Class Loaders

Insert this section in section 3 of the paper.

Before we continue to describe OSGi modularity, we model a simple class loader which delegates to a parent class loader. This will help to familiarise the reader with the notation and how class loading is modelled.

A simple class loader has a local collection of class definitions, typically implemented as a sequence of jar files on a file system, but modelled here as a class space. It also has a class space of classes which have been defined (i.e. loaded locally) by the class loader and which are derived from the loader's local collection of class definitions. Finally it has a class space of all classes loaded by the class loader. We defer tying this to the the parent class loader.

$$\begin{array}{|l}
\textit{SimpleLoader} \\
\hline
\textit{definitions} : \textit{ClassSpace} \\
\textit{defined} : \textit{ClassSpace} \\
\textit{loaded} : \textit{ClassSpace} \\
\hline
\textit{defined} = \textit{definitions} \cap \textit{loaded} \\
\hline
\end{array}$$

The defined classes are those which have definitions and which are also loaded. Note that this only makes sense because the child and parent definitions turn out to be disjoint. If a class is loaded from identical class files by two distinct class loaders, the resultant class objects are distinct.

A simple loader initially has no loaded classes.

$$\begin{array}{|l}
\textit{SimpleLoaderInit} \\
\hline
\textit{SimpleLoader} \\
\hline
\textit{loaded} = \emptyset \\
\hline
\end{array}$$

We now model a successful class load.

$$\begin{array}{|l}
\textit{SuccessfulLoad} \\
\hline
\Delta \textit{SimpleLoader} \\
\textit{cn}? : \textit{ClassName} \\
\hline
\textit{definitions}' = \textit{definitions} \\
\textit{defined} \subseteq \textit{defined}' \\
\textit{loaded} \subseteq \textit{loaded}' \\
\textit{cn}? \in \operatorname{dom} \textit{loaded}' \\
\hline
\end{array}$$

The definitions are preserved. The collections of defined and loaded classes may not decrease. The specified class is loaded. Note that more than one class may be loaded, which may be necessary if the specified class refers to other classes which have not yet been loaded.

A *bootstrap* class loader is at the top of the parent-child hierarchy and therefore only loads classes it also defines. We refer to such a class loader as a *top loader*.

$$
\begin{array}{|l}
\textit{TopLoader} \\
\hline
\textit{SimpleLoader} \\
\hline
\textit{defined} = \textit{loaded} \\
\hline
\end{array}
$$

We shall see later that a simple loader together with its parentage up to and including the bootstrap loader may be modelled as a top loader.

With this in mind, we can express the relationship between a child class loader and its parentage.

$$
\begin{array}{|l}
\textit{ParentalConstraint} \\
\hline
\textit{child} : \textit{SimpleLoader} \\
\textit{parentage} : \textit{TopLoader} \\
\hline
\textit{child.loaded} \setminus \textit{child.defined} \subseteq \textit{parentage.loaded} \\
(\operatorname{dom} \textit{child.defined}) \cap (\operatorname{dom} \textit{parentage.definitions}) = \emptyset \\
\textit{child.definitions} \cap \textit{parentage.definitions} = \emptyset \\
\hline
\end{array}
$$

Classes which the child has loaded but not defined must have been loaded by the parentage. The child has not defined a class for which the parentage has a definition. The child and parentage definitions are disjoint which models the fact that distinct class loaders produce distinct class instances when they load from the same class bytecode.

The first two properties are ensured by the child class loader first of all delegating each class load to its parent and then only attempting to define the class if its parent failed to load the class.

It follows that *child.loaded* and *parentage.loaded* are consistent class spaces. The proof is omitted because of lack of space.

We now show how a child together with a top loader satisfying the above constraint can be combined to produce a new top loader.

$$
\begin{array}{|l}
\textit{Combine} \\
\hline
\textit{ParentalConstraint} \\
\textit{loader} : \textit{SimpleLoader} \\
\hline
\textit{loader.definitions} = \textit{child.definitions} \oplus \textit{parentage.definitions} \\
\textit{loader.defined} = \textit{child.defined} \cup \textit{parentage.defined} \\
\textit{loader.loaded} = \textit{child.loaded} \cup \textit{parentage.loaded} \\
\hline
\end{array}
$$

The proof that the result is really a top loader is omitted for lack of space.

$$\textit{Combine} \vdash \textit{loader} \in \textit{TopLoader}$$

The reverse process of factoring a child class loader from a top loader is also sometimes possible, but is not explored further here.

5 Delegating Class Loaders

Possibly omit this from the paper.

As a further step towards OSGi R4 modularity, we extend the notion of a simple loader to model imports and exports.

An importer is a simple loader which defines the class names it is willing to import from elsewhere. In OSGi R4 as we shall see later, imports are defined in terms of packages, but we will overlook that level of detail for the moment. For simplicity, we also ignore the parent class loader.

$$
\begin{array}{|l}
\hline
\textit{Importer} \\
\hline
\textit{SimpleLoader} \\
\textit{imports} : \mathbb{P}\,\textit{ClassName} \\
\textit{imported} : \textit{ClassSpace} \\
\hline
\textit{imported} = \textit{loaded} \setminus \textit{defined} \\
\operatorname{dom} \textit{imported} \subseteq \textit{imports} \\
\hline
\end{array}
$$

Classes which are loaded, but not defined, are imported. Imported classes must be named as imports.

An exporter is a simple loader which defines the class names it is willing to export to an importer.

$$
\begin{array}{|l}
\hline
\textit{Exporter} \\
\hline
\textit{SimpleLoader} \\
\textit{exports} : \mathbb{P}\,\textit{ClassName} \\
\textit{exported} : \textit{ClassSpace} \\
\hline
\textit{exported} \subseteq \textit{loaded} \\
\operatorname{dom} \textit{exported} \subseteq \textit{exports} \\
\hline
\end{array}
$$

Exported classes must be loaded and named as exports.

We now define a constraint between a 'matching' importer and exporter.

$$
\begin{array}{|l}
\hline
\textit{DelegationConsistency} \\
\hline
i : \textit{Importer} \\
e : \textit{Exporter} \\
\hline
i.\textit{definitions} \cap e.\textit{definitions} = \emptyset \\
\textbf{let}\ \textit{shared} == i.\textit{imports} \cap e.\textit{exports} \bullet \\
\qquad \textit{shared} \lhd i.\textit{imported} \subseteq \textit{shared} \lhd e.\textit{exported} \\
\hline
\end{array}
$$

The importer and exporter definitions are disjoint which models the fact that distinct class loaders produce distinct class instances when they load from the same class bytecode. If shared classes denote those which may be both imported by the importer and exported by the exporter, then the shared classes which have been imported have also been exported.

It follows that $\textit{shared} \lhd i.\textit{loaded}$ and $\textit{shared} \lhd e.\textit{loaded}$ are consistent class spaces.

One of the main problems solved by OSGi R4 is how to match imports to exports in order to maintain proper consistency of class spaces and avoid certain kinds of type mismatches resulting in class loading failures, class cast exceptions, etc.

The solution starts with a 'module definition'.

6 Module Definition

Insert the start of section in section 4.2 (Module Content) of the paper and intersperse the import/export related schemas into sections 4.3 (Imported Packages) and 4.4 (Exported Packages). Insert the 'matches' relation in section 4.8 (Module Resolution) of the paper.

A module system contains a collection of modules which share packages in various ways.

Before we model the module system, we focus first on the definition of an individual module and secondly, in the next section, on the runtime state of an individual module. A module's definition identifies the module by module name and module version. It also contains class definitions which are contained in the module and may be loaded by the module.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ModuleId}} \\
\hline
\textit{mname} : \textit{MName} \\
\textit{mver} : \textit{MVer} \\
\textit{mv} : \textit{MV} \\
\hline
\textit{mv} = (\textit{mname}, \textit{mver}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ModuleDef}_0} \\
\hline
\textit{ModuleId} \\
\textit{definitions} : \textit{ClassSpace} \\
\hline
\textit{mv} = (\textit{mname}, \textit{mver}) \\
\hline
\end{array}
$$

A complete list of imported and exported classes would be hard to maintain. Also a given class module does not typically want an arbitrary mixture of classes from various sources. The OSGi design point is to make the Java package the minimum unit of resolution.

Each package which a module exports has an export specification which specifies the module name, module version, package version, and a set of matching attributes some of which must be supplied by an importer which wishes to use the exported package. The export specification may also require a matching importer to specify module name or module version (or both).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ExportSpec}} \\
\hline
\textit{ModuleId} \\
\textit{pver} : \textit{PVer} \\
\textit{attr} : \textit{AMap} \\
\textit{mandatory} : \mathbb{P}\, \textit{AName} \\
\hline
\textit{MNameAttr} \notin \mathrm{dom}\, \textit{attr} \\
\textit{mandatory} \subseteq (\mathrm{dom}\, \textit{attr} \cup \{\textit{MNameAttr}\}) \\
\hline
\end{array}
$$

Each package which a module imports has an import specification.

A module name may be specified (using a singleton set) or not (using an empty set). Module versions may only be constrained if a module name has been specified. A range of acceptable module versions or package versions is represented as a set containing all the elements of the range.

$$
\begin{array}{|l}
\textit{ImportSpec} \\
\hline
\textit{mname} : \mathbb{P}\,\textit{MName} \\
\textit{mver} : \mathbb{P}\,\textit{MVer} \\
\textit{pver} : \mathbb{P}\,\textit{PVer} \\
\textit{attr} : \textit{AName} \mathrel{+\!\!\!\rightarrow} \textit{AValue} \\
\hline
\#\textit{mname} \leq 1 \\
\textit{mname} == \Rightarrow \textit{mver} = \textit{MVer} \\
\textit{MNameAttr} \notin \operatorname{dom} \textit{attr} \\
\hline
\end{array}
$$

An export specification matches an import specification if and only if:

- any module name, module versions, and package versions in the import specification are satisfied by exported module name, module version, and package version,
- any arbitrary attributes specified in the import specification match the values specified in the export specification,
- any arbitrary attributes specified as mandatory in the export specification are specified in the import specification,
- if the export specification mandated the module name, the import specification specifies the module name.

$$
\begin{array}{|l}
\textit{matches} : \textit{ImportSpec} \leftrightarrow \textit{ExportSpec} \\
\hline
\forall\, \textit{is} : \textit{ImportSpec};\ \textit{es} : \textit{ExportSpec} \bullet \\
\quad \textit{is} \mapsto \textit{es} \in \textit{matches} \Leftrightarrow \\
\qquad \textit{is.mname} = \vee\ \textit{is.mname} = \{\textit{es.mname}\} \wedge \\
\qquad \textit{es.mver} \in \textit{is.mver} \wedge \\
\qquad \textit{es.pver} \in \textit{is.pver} \wedge \\
\qquad \textit{is.attr} \subseteq \textit{es.attr} \wedge \\
\qquad \textit{es.mandatory} \subseteq (\operatorname{dom} \textit{is.attr} \cup \{\textit{MNameAttr}\}) \wedge \\
\qquad \textit{MNameAttr} \in \textit{es.mandatory} \Rightarrow \textit{is.mname} \neq
\end{array}
$$

A module's definition identifies the packages the module imports and exports by providing corresponding import and export specifications. A module may provide at most one import specification for a given package name but it may have multiple export specifications for a given package name.

A module may not name itself in an import specification.

However a module can import and export the same package, although as we shall see later, such a package is defined by one and only one module. If a module imports and exports the same package, either the import or the export is honoured when the module is resolved and the other statement is disregarded. We add abbreviations for the sets of classes which may be exported and imported.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{ModuleDef} \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\mathit{ModuleDef}_0 \\
\mathit{exports} : \mathit{Package} \leftrightarrow \mathit{ExportSpec} \\
\mathit{imports} : \mathit{Package} \nrightarrow \mathit{ImportSpec} \\
\mathit{classExports}, \mathit{classImports} : \mathbb{P}\, \mathit{ClassName} \\
\hline
\forall\, \mathit{es} : \operatorname{ran} \mathit{exports} \bullet \mathit{es}.\mathit{mv} = \mathit{mv} \\
\forall\, \mathit{is} : \operatorname{ran} \mathit{imports} \bullet \mathit{mv} \notin \mathit{is}.\mathit{mname} \times \mathit{is}.\mathit{mver} \\
\mathit{classExports} = \mathit{package}^{\sim} (\!|\, \operatorname{dom} \mathit{exports} \,|\!) \\
\mathit{classImports} = \mathit{package}^{\sim} (\!|\, \operatorname{dom} \mathit{imports} \,|\!) \\
\hline
\end{array}
$$

7 Module

Insert this section in section 4.8 (Module Resolution) of the paper.

Based on its definition, a module loads classes either from its contents or by importing from another module[1]. These loaded classes are modelled using various class spaces in addition to the module definition:

defined classes loaded by the module[2],

imported classes imported from other modules.

Note that *defined* and *imported*, together with the module's definition, determine the other class spaces:

loaded all classes available to a module,

exported classes exported to other modules,

private classes loaded by the module which are not exported.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Module}} \\
\hline
\textit{ModuleDef} \\
\textit{defined} : \textit{ClassSpace} \\
\textit{imported} : \textit{ClassSpace} \\
\textit{loaded} : \textit{ClassSpace} \\
\textit{exported} : \textit{ClassSpace} \\
\textit{private} : \textit{ClassSpace} \\
\hline
\textit{defined} \subseteq \textit{definitions} \\
\textit{defined} = \textit{classImports} \ntriangleleft \textit{loaded} \\
\textit{imported} = \textit{classImports} \triangleleft \textit{loaded} \\
\textit{exported} = \textit{classExports} \triangleleft \textit{loaded} \\
\textit{private} = \textit{classExports} \ntriangleleft \textit{loaded} \\
\hline
\end{array}
$$

Some interesting properties follow.

$$
\begin{aligned}
&\textit{Module} \vdash \\
&\qquad \langle \textit{defined}, \textit{imported} \rangle \ \mathsf{partition}\ \textit{loaded} \wedge \\
&\qquad \langle \textit{private}, \textit{exported} \rangle \ \mathsf{partition}\ \textit{loaded} \wedge \\
&\qquad \textit{package}(\!|\ \mathrm{dom}\,\textit{defined}\ |\!) \cap \mathrm{dom}\,\textit{imports} = \emptyset \wedge \\
&\qquad \textit{defined} = \textit{definitions} \cap (\textit{loaded} \setminus \textit{imported})
\end{aligned}
$$

The defined and imported class spaces partition the loaded class space. The private and exported class spaces also partition the loaded class space. No classes are loaded locally which belong to imported packages. Locally loaded

[1]For the purposes of this model, we ignore the parent class loader

[2]Strictly speaking, *defined* models classes for which the module's class loader is the *defining* loader.

classes are precisely those which have local definitions and which have been loaded but not imported.

The proofs of these properties is left as an exercise for the reader.

8 Module System

Insert this section in section 4.8 (Module Resolution) of the paper.

To give an indication of how the pieces of the specification fit together, we model a service platform of installed modules (m) some of which are resolved and with a wiring function ($wire$) wiring together resolved modules for packages.

$$
\begin{array}{|l}
\hline
\textit{ModuleSystemBase} \\
m : MV \rightarrowtail\!\!\!\!\!\!\!\to Module \\
resolved : \mathbb{P}\, MV \\
wire : MV \times Package \nrightarrow MV \\
\hline
\operatorname{ran} wire \subseteq resolved \\
resolved \subseteq \operatorname{dom} m \\
first \,(\!|\, \operatorname{dom} wire \,|\!) \subseteq resolved \\
(\forall\, mv : resolved \mid (m\ mv).imports \neq \emptyset \bullet mv \in first \,(\!|\, \operatorname{dom} wire \,|\!)) \\
\hline
\end{array}
$$

Note that the package wiring for a given package p is described by the function $\lambda\, mv : MV \bullet wire\,(mv, p)$.

Two constraints must be satisfied. The first is that imports must be matched by exports.

$$
\begin{array}{|l}
\hline
\textit{ImportExportMatching} \\
ModuleSystemBase \\
\hline
(\forall\, m1 : MV;\ p : Package \mid (m1, p) \in \operatorname{dom} wire \bullet \\
\quad p \in \operatorname{dom}(m\ m1).imports \land \\
\quad p \in \operatorname{dom}((m \circ wire)(m1, p)).exports \land \\
\quad (\mathbf{let}\ is == (m\ m1).imports\ p \bullet \\
\qquad (\exists\, es : (((m \circ wire)(m1, p)).exports)(\!|\, \{p\} \,|\!) \bullet \\
\qquad\quad (is, es) \in matches))) \\
\hline
\end{array}
$$

The second is that matched importers and exporters are consistent.

$$
\begin{array}{|l}
\hline
\textit{WiringConsistency} \\
ModuleSystemBase \\
\hline
\forall\, i, e : MV;\ p : Package \mid (i, p) \mapsto e \in wire \bullet \\
\quad \mathbf{let}\ mi == m\ i;\ me == m\ e;\ c == package^{\sim}(\!|\, \{p\} \,|\!) \bullet \\
\qquad c \lhd mi.imported \subseteq c \lhd me.exported \\
\hline
\end{array}
$$

Then a module system must satisfy both constraints.

$$ModuleSystem \mathrel{\widehat=} ImportExportMatching \land WiringConsistency$$

$$
\begin{array}{|l}
\hline
\textit{ResolveOk} \\
\hline
\Delta \textit{ModuleSystem} \\
m? : \textit{MV} \\
\textit{mdef}? : \textit{ModuleDef} \\
\hline
m? \notin \operatorname{dom} m \\
m? \in \textit{resolved}' \\
\hline
\end{array}
$$

9 Class Search Order

Insert this section in section 4.9 (Run-time Class Search Order) of the paper.

Although we have avoided modelling the parent class loader and Require-Bundle so far, it is essential to see how these features combine into the final class search order. We add the parent class loader in, modelled as a top loader, and the contribution of all required modules modelled as a class space. We also model the search order explicitly as a class space.

$$
\begin{array}{|l}
\hline
\textit{CompleteModule} \\
\textit{parent} : \textit{TopLoader} \\
\textit{required} : \textit{ClassSpace} \\
\textit{search} : \textit{ClassSpace} \\
\textit{ModuleDef} \\
\textit{defined} : \textit{ClassSpace} \\
\textit{imported} : \textit{ClassSpace} \\
\textit{loaded} : \textit{ClassSpace} \\
\textit{exported} : \textit{ClassSpace} \\
\textit{private} : \textit{ClassSpace} \\
\hline
(\textit{classImports} \cup \textit{classExports}) \cap \textit{javaClasses} = \emptyset \\
\textit{classImports} \cap \operatorname{dom} \textit{required} = \emptyset \\
\textit{loaded} = (\textit{defined} \cup \textit{required} \cup \textit{imported}) \\
\qquad \oplus (\textit{javaClasses} \lhd \textit{parent.loaded}) \\
\textit{search} = ((\textit{classImports} \ntriangleleft (\textit{definitions} \oplus \textit{required})) \cup \textit{imported}) \\
\qquad \oplus (\textit{javaClasses} \lhd \textit{parent.definitions}) \\
\textit{defined} \subseteq \textit{definitions} \\
\textit{defined} = ((\textit{classImports} \cup \textit{javaClasses}) \ntriangleleft \textit{loaded}) \setminus \textit{required} \\
\textit{imported} = \textit{classImports} \lhd \textit{loaded} \\
\textit{exported} = \textit{classExports} \lhd \textit{loaded} \\
\textit{private} = \textit{classExports} \ntriangleleft \textit{loaded} \\
\hline
\end{array}
$$

Neither imports not exports may specify packages beginning with "java.". Required classes do not include any classes which are specified as imports.

Loaded classes consist of those defined by the module, those required from other modules, those imported from other modules, but all overridden with the parent class loader's packages which begin with "java.".

The search order reflects the parent class loader being searched for packages beginning with "java.", and then imported packages, and then required and defined classes which do not belong to impored packages.

The defined classes are those which are loaded but not inherited from the parent, imported, or required. Imported classes are those which are loaded and which belong to imported packages. Exported classes are those which are loaded and which belong to exported packages. Private classes are those which are loaded but which do not belong to exported packages.

10 Z Notation

Numbers:

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

Propositional logic and the schema calculus:

$\ldots \land \ldots$	And	$\langle\!\langle \ldots \rangle\!\rangle$	Free type injection
$\ldots \lor \ldots$	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
$\ldots \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} \ldots$	Partial function	$\ldots^{\sim}$	Relational inverse
$\ldots \rightarrow \ldots$	Total function	$\ldots^{*}$	Reflexive-transitive closure
$\ldots \rightarrowtail\!\!\!\!\!+\; \ldots$	Partial injection	$\ldots (\!\vert \ldots \vert\!)$	Relational image
$\ldots \rightarrowtail \ldots$	Injection	$\ldots \oplus \ldots$	Functional overriding
$\mathrm{dom} \ldots$	Domain	$\ldots \lhd \ldots$	Domain restriction
$\mathrm{ran} \ldots$	Range	$\ldots \mathrel{\lhd\!\!\!-} \ldots$	Domain subtraction

Axiomatic descriptions:

Declarations

Predicates

Schema definitions:

SchemaName

Declaration

Predicates