Unclassified, v0.2

A Formal Model of the SpringSource OSGi R4 Resolver (v0.2)

Rob Harrop and Steve Powell

April 23, 2009

We model aspects of the OSGi R4 Resolver.

Contents

1 Introduction

This specification was begun by Rob Harrop and pursued by Steve Powell as an exercise in understanding OSGi wiring and the resolution algorithm. We intend to search for good ways to determine all possible wirings that satisfy the constraints. It may be that the problem is NP-complete and that our search for an efficient algorithm is futile. Either outcome is an advance on the present state-of-the-art.

The early work introduced `Require-Bundle` as one type of *Requirement*, but we begin by ignoring `Require-Bundle` manifest headers and concentrating upon `Import-Package` and `Export-Package`. Before re-introducing required bundle headers we intend to incorporate the constraints implied by the `uses` directives.

This work is at an intermediate stage.

References

[1] OSGi Service Platform Core Specification v4.1, April 2007 `http://www.OSGi.org`

2 Basic Types

Attributes, attribute values, identifiers, versions and names are interesting, but beside the point. Their precise structure has no bearing on the problem. We are only interested in equality on these things (even versions, it turns out) so we merely refer to them as ‘basic types’ or, in the Zednacular, ‘Given sets’.

2.1 Attributes

Attribute names (*AName*) and attribute values (*AValue*) are used to provide a flexible symbolic matching mechanism:

$$[AName, AValue]$$

The association between names and values is a map (a partial function). These maps are explicitly specified on both import and export package headers. Matching them is part of the resolution process.

2.2 Names

Packages have names and versions.

$$[PackageName, Version]$$

Apart from equality, we are not interested in what these look like. A version is (almost) a lexicographically ordered point decimal sequence, but we don’t actually care (yet).

The structure of a version is not exposed and we interpret version ranges as sets. The version range of an import package header is thus a set of *Version*s even when a single version is mandated. In that case it would be a singleton set.

There is an earliest version number which we can name here. All other versions are later than this one. We only need this as a default value; we never need to use the ordering on *Version*, but we may refer to it in the descriptive text.

$$\mid \; EARLIEST\_VERSION : Version$$

2.3 Module Identifiers

A *ModuleId* identifies a module (*aka* bundle) in a collection (the *ModuleCollection*, see later). Just like other names this is not further specified.

$$[\mathit{ModuleId}]$$

*ModuleId*s have to be distinct for distinct modules in the resolution space (the resolved module system). The OSGi specification states that the ('global') system module has an identifier of 0. We can avoid making this explicit:

$$\mathit{SYSTEM\_MODULEID} : \mathit{ModuleId}$$

Modules also have a name—the bundle symbolic name—and a version. Versions are identical in form to package versions, but bundle symbolic names are not the same as package names:

$$[\mathit{SymbolicName}]$$

These are used to constrain package wiring. This is achieved by providing attributes that refer to bundle symbolic names. This means that an attribute value must represent a *SymbolicName*, and so there is an injection of *SymbolicName* into *AValue*:

$$\mathit{BSN} : \mathit{SymbolicName} \rightarrowtail \mathit{AValue}$$

This is only used when attribute matching involving the bundle symbolic name of the module from which an exported package is obtained.

Within a system the combination of symbolic name and version must uniquely identify a module, and therefore are in 1-1 correspondence with module identifiers. These facts will be reflected in our description of a *ModuleCollection*.

3 Modularity

A *Module* is (as far as we are concerned) a collection of export and import headers (at least, it is at the moment, until we introduce `Require-Bundle` later). The exports and imports both express constraints imposed upon the 'wiring' of a module in a system.

3.1 Requirements

Modules have *Requirement*s that must be satisfied by the resolver when it is resolving a module.

Modules can have requirements for either bundles (`Require-Bundle`) or packages (`Import-Package`). At this stage we only model the `Import-Package` case, so we define an *ImportRequirement*:

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{ImportRequirement}} \\
\mathit{pName} : \mathit{PackageName} \\
\mathit{pVersionRange} : \mathbb{P}\ \mathit{Version} \\
\mathit{specAttrs} : \mathit{AName} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{AValue} \\
\hline
\end{array}$$

Each part of the *ImportRequirement* limits the exports to which this import can be wired. The *pName* is the name of the package, the *pVersionRange* is the set of permissible versions of the package, and the *specAttrs* map records those attributes that must be defined (and match in value) in a package export.

An *ImportRequirement* may also be *optional*, modelled in *Module* below.

3.2 Providers

Modules provide potential wiring points to satisfy *Requirement*s. These are called *Provider*s. The `Export-Package` headers expose one sort of *Provider*.

An *ExportProvider* has a package name and an explicit version, a collection of named attributes (some of which are mandatory) and a set of (package) names that this export *uses*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ExportProvider}} \\ \hline
\mathit{pName} : \mathit{PackageName} \\
\mathit{pVersion} : \mathit{Version} \\
\mathit{attrs} : \mathit{AName} \mathrel{+\!\!\!\rightarrow} \mathit{AValue} \\
\mathit{mandatory} : \mathbb{P}\, \mathit{AName} \\
\mathit{uses} : \mathbb{P}\, \mathit{PackageName} \\ \hline
\mathit{mandatory} \subseteq \operatorname{dom} \mathit{attrs} \\
\mathit{pName} \notin \mathit{uses} \\ \hline
\end{array}
$$

The *pName* is the name of the exported package, and *pVersion* is its (explicit) version. (These are always present – the default *pVersion* of *EARLIEST_VERSION* is assumed if the `Export-Package` header omits it.) The mandatory set of attribute names must all come from the attributes of the export. The package name must not appear in its own *uses* set.

The other sort of *Provider* is a module itself, these satisfy the `Require-Bundle` or `Import-Bundle` headers. We omit these for now.

3.3 Modules

A module both exports and imports packages. Some of the imports are optional.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Module}} \\ \hline
\mathit{bName} : \mathit{SymbolicName} \\
\mathit{bVersion} : \mathit{Version} \\
\mathit{imports} : \mathit{PackageName} \mathrel{+\!\!\!\rightarrow} \mathit{ImportRequirement} \\
\mathit{mandatoryImports} : \mathbb{P}\, \mathit{PackageName} \\
\mathit{exports} : \mathbb{P}\, \mathit{ExportProvider} \\ \hline
\mathit{mandatoryImports} \subseteq \operatorname{dom} \mathit{imports} \\
\forall\, n : \operatorname{dom} \mathit{imports} \bullet (\mathit{imports}\ n).\mathit{pName} = n \\ \hline
\end{array}
$$

The *bName* denotes the bundle-symbolic-name, *bVersion* is the bundle-version, and the import and export bundle headers are summarized in the *imports* and *exports* sets.

The mandatory imports are a specified subset of the import packages, the others, in the complementary set ($\operatorname{dom} \mathit{imports} \setminus \mathit{mandatoryImports}$), are the optional imports. Packages in an `Import-Package` header are mandatory by default, and optional when the directive `resolution:=optional` is specified.

It must be the case that distinct imports refer to distinct package names—which is why we explicitly index the *ImportRequirements* by the package names that occur there—though many exports can refer to the same package name, provided some attribute (including version) differs. This latter rule is sufficiently enforced by the use of a *set* of *ExportProviders*.

It is worth noting that exports referring to the same package name need only differ in one other attribute—for example the *uses* set, or the set of *mandatory* attribute names.

We proceed by describing a *collection* of modules, without regard to any resolution. These form the collection of 'installed' modules.

3.4 Module Collection

In a collection a module is identified by a single identifier and also by a combination of its symbolic name and version. The natural ways to represent this are as a(n injective) map from identifiers to modules, or from symbolic names and versions to modules. We define both and keep them consistent by constraint.

$$\begin{array}{|l}
\textit{ModuleCollection} \\
\hline
\textit{moduleById} : \textit{ModuleId} \mathrel{\rightarrowtail\!\!\!\!\!\!+} \textit{Module} \\
\textit{moduleByName} : \textit{SymbolicName} \times \textit{Version} \mathrel{\rightarrowtail\!\!\!\!\!\!+} \textit{Module} \\
\hline
\textit{SYSTEM\_MODULEID} \in \operatorname{dom} \textit{moduleById} \\
\operatorname{ran} \textit{moduleById} = \operatorname{ran} \textit{moduleByName} \\
\forall\, \textit{mid} : \operatorname{dom} \textit{moduleById};\; m : \textit{Module} \mid m = \textit{moduleById}\ \textit{mid} \\
\qquad \bullet\ \textit{moduleByName}(m.\textit{bName}, m.\textit{bVersion}) = m \\
\hline
\end{array}$$

moduleById is the (1-1) indexed collection of modules of the system, which are also exactly those indexed by the combinations of *SymbolicName* and *Version* in *moduleByName*. The name and the version indexing a module is that which is recorded in the module itself.

It should be obvious that *moduleByName* must be a partial injection because the symbolic name and version are embedded in the module; if we had defined it only as a partial function it would follow easily that it is 1-1.

4 Resolution - Part One

To introduce resolved systems of modules we have to discuss wires and wirings. In this first look at resolution we will consider simple wiring rules and constraints for correct resolution. Uses constraints are considered in Part Two.

4.1 Wires

A *wire* is the smallest unit of resolution – it connects a requirement in a module to a provider in another module. We consider only imports connected to exports. When a particular import requirement (in a particular module) is satisfied by a particular export from a particular module, this association (or connection) is a *ConsistentWire*. A collection of consistent wires is called a 'wiring'.

A wire connects an import requirement (`Import-Package`) in a module to a consistent export provider (`Export-Package`) in another module; by consistent we mean that the requirement and the provider match attributes, package name and version in a particular way:

$$
\begin{array}{|l}
\textit{ConsistentWire} \\
\hline
\textit{ImportRequirement} \\
\textit{ExportProvider} \\
\textit{imod}, \textit{emod} : \textit{Module} \\
\hline
\textit{pName} \in \operatorname{dom} \textit{imod.imports} \\
\theta \textit{ImportRequirement} = \textit{imod.imports}\ \textit{pName} \\
\theta \textit{ExportProvider} \in \textit{emod.exports} \\
\textit{pVersion} \in \textit{pVersionRange} \\
\textit{specAttrs} \subseteq \textit{attrs} \\
\textit{mandatory} \subseteq \operatorname{dom} \textit{specAttrs} \\
\hline
\end{array}
$$

The *pName* is shared between the import and the export, which means they must match, and identifies the *ImportRequirement* in the corresponding module; the *ExportProvider* is one of those in the export headers of the *emod* module; the *pVersion* must appear in the requirement *pVersionRange*; all the attributes specified must be set (and agree in value) in the export; and the mandatory attributes must be specified.

This form of consistency is called 'local' because only the properties of the immediately connected modules are needed to determine if it is valid. When we consider an entire wiring, there are other consistencies to consider which are ‘non-local’ (or ‘global’): it is necessary to consider other wires in the wiring before we can determine their validity.

4.2 Wire space

A *wiring* accompanies a module collection with a set of consistent wires that can be applied to the collection. These wires explicitly connect an import of a module to a single export in some other module, so we could make *wiring* a partial function. It is, however, convenient to consider all the *possible* wires from an import simultaneously, so for the moment we will make no such restriction and instead consider the *WireSpace* which identifies all the locally consistent wires in a collection.

First, some abbreviations for the wires themselves:

$$WIRE == (ModuleId \times PackageName) \times (ModuleId \times ExportProvider)$$

$$WIRES == ModuleId \times PackageName \leftrightarrow ModuleId \times ExportProvider$$

and another to refer to a single consistent wire in the form we prefer:

$$\begin{array}{|l}
\hline
\textit{OneWire} \\
\hline
\textit{ConsistentWire} \\
\textit{imid}, \textit{emid} : \textit{ModuleId} \\
w : \textit{WIRE} \\
\hline
w = (\textit{imid}, \textit{pName}) \mapsto (\textit{emid}, \theta \textit{ExportProvider}) \\
\hline
\end{array}$$

The full wire space (all locally consistent possibilities) is thus:

$$\begin{array}{|l}
\hline
\textit{WireSpace} \\
\hline
\textit{ModuleCollection} \\
\textit{wirespace} : \textit{WIRES} \\
\hline
\textit{wirespace} = \\
\qquad \{\, \textit{OneWire} \mid \{\textit{imid} \mapsto \textit{imod}, \textit{emid} \mapsto \textit{emod}\} \subseteq \textit{moduleById} \bullet w \,\} \\
\hline
\end{array}$$

where all the (locally) consistent wires which are possible in the *ModuleCollection* are put into *wirespace*. Each wire association is therefore represented by an element of the relation.

Notice carefully that here we have *not* imposed the constraint that *the import and export modules are distinct in every wire* (no wiring to self); nor the constraint that *all the imports should be wired in any module connected* *to.* These constraints are non-local and will be introduced in the next section; besides the maximally wired module collection is an interesting object in its own right: all correctly resolved combinations of wires are *subgraphs* of this directed graph.

Importing and exporting the same package

It is permitted to specify an import and export(s) of the same package (name) in a single module. This is, however, modified as part of the resolution process (see section *3.7 Resolving Process* in [1]). At resolution time, a choice is made between the import and the exports and one is "discarded" (*sic*). Where there are multiple exports for the same package name in a module they are kept (or discarded) together.

The OSGi specification places a priority upon import definitions—if an import can be resolved (wired) to (an export in) another module without reference to the exports (of the same name) then the import is kept and the exports discarded. Otherwise, the import is discarded and the exports are taken, *provided an export matches the import*, in the sense of local consistency. If no such match occurs then the import is deemed 'un-resolved' (that is, it is not wired). In this case, due to 'non-local' consistency rules (specified later), the entire module cannot be resolved: nothing can be wired to exports or imports of that module.

These rules appear to be circular: the rule for determining which imports and exports to keep and wire (and which to discard) depends upon the wiring which, due to 'non-local' consistency rules, depends upon which imports and exports are wired. This gets worse when we consider uses constraints (next section).

We do not model this process, yet, neither do we prohibit self-wiring yet. If necessary we should impose this rule independently as the other (non-local) constraints are imposed.

4.3 Simple non-local constraints

The *ModuleCollection* maintains a set of all known modules (those that have been installed) and a *wirespace* associates imports with exports. There are further non-local constraints that must be satisfied by a wiring before we can say it has properly resolved the modules in it. These constraints are called non-local because they rely upon the arrangement of many wires, and cannot be determined by consideration of each wire alone.

Each non-local constraint will be defined separately to produce a *schema* defining wirings that satisfy the constraint. The join of the schemas specifies wirings that are ultimately permitted.

We first describe any consistent wiring:

$$
\begin{array}{|l}
\hline
\textit{ConsistentWiring} \\
\textit{WireSpace} \\
\textit{consistent} : \textit{WIRES} \\
\hline
\textit{consistent} \subseteq \textit{wirespace} \\
\hline
\end{array}
$$

Every sub-relation of the (maximal) relation *wirespace* is a (locally) consistent wiring.

Single import source

The first simple non-local constraint imposes the rule that an import package requirement can be wired to at most one export package provider.

$$
\begin{array}{|l}
\hline
\textit{SingleImports} \\
\textit{ConsistentWiring} \\
\hline
\textit{consistent} \in \textit{ModuleId} \times \textit{PackageName} \pfun \textit{ModuleId} \times \textit{ExportProvider} \\
\hline
\end{array}
$$

That is, the wiring relation must be a partial function. This is exactly what we want to say: each import requirement must be related to at most one export provider.

All or nothing imports

The next simple non-local constraint is that, in a particular module, either all (mandatory) imports are wired or none of them are, and if an export is wired all its module's (mandatory) imports must be wired as well.

It is easier to understand this constraint if we first define (and name) the set of ids of those modules that have some wire to or from them—*modids*:

$$
\begin{array}{|l}
\hline
\textit{AllImportsExposed} \\
\textit{ConsistentWiring} \\
\textit{modids} : \mathbb{P}\,\textit{ModuleId} \\
\hline
\textit{modids} = \mathrm{dom}(\mathrm{ran}\,\textit{consistent}) \cup \mathrm{dom}(\mathrm{dom}\,\textit{consistent}) \\
\forall\, m : \textit{modids} \bullet \\
\qquad \{m\} \times (\textit{moduleById}\ m).\textit{mandatoryImports} \subseteq \mathrm{dom}\,\textit{consistent} \\
\hline
\end{array}
$$

the set of all module identifiers that appear at either end of a wire, *modids*, can be gleaned from the consistent wiring relation, and we can then go further and stipulate that all (mandatory) import packages of these modules must be wired.

After we have imposed this, there is no further need to expose the set *modids*, and the constraint we use later is named *AllImports*.

$$\mathit{AllImports} \mathrel{\widehat{=}} \mathit{AllImportsExposed} \setminus (\mathit{modids})$$

5 Resolution - Part Two

In this section we consider the more complex non-local rules imposed by 'uses' constraints.

5.1 Uses constraints - concepts

In order to explain (and explore) the notion of 'uses' more easily we introduce a diagrammatic notation. Figure 1 shows the conventions we shall use.

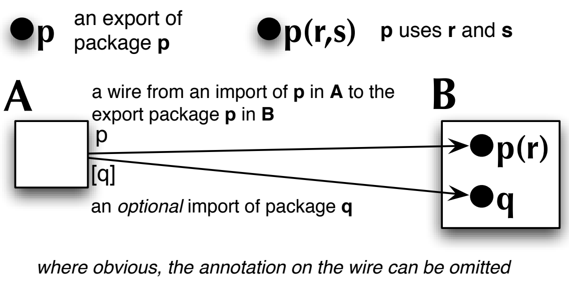
●p
an export of package **p**
●p(r,s)
p uses **r** and **s**
A
a wire from an import of **p** in **A** to the export package **p** in **B**
B
p
[q]
●p(r)
●q
an *optional* import of package **q**
where obvious, the annotation on the wire can be omitted

Figure 1: Key to diagrammatic notation for uses constraints.

Essentially, the uses constraint says that if a module imports a package `p` which 'uses' a package `q`, then if it has access to `q` (for example, if it imports it) then it must access the same one that the module that exported `p` does.

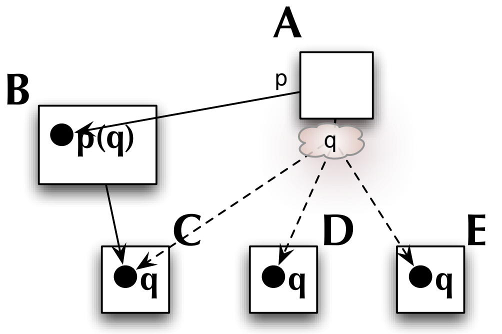

Figure 2: Illustration of the uses constraint in the simple case.

In figure 2, for example, is a module (bundle) `A` which imports packages `p` and `q`. The module `B` exports package `p` and imports package `q`, and gets `q` from `C` (the import `q` is wired to an export `q` in `C`).

The module `A` could apparently source package `q` from either `C`, `D` or `E`, but only one is consistent with the wiring from `B`.

Without the 'uses' constraint on module `B` any one of the dotted lines could be used to satisfy the wiring constraint (the import of `q` on `A`). The uses constraint on `B` says that `A` must get the same exported package `q` that `B` gets. Thus, `A` must wire to `C`'s exported `q`. In fact, must wire to the *same export of* `q` in `C` as `B` does.

'Uses' constraints are transitive. Consider the arrangement of modules and wires in figure 3.

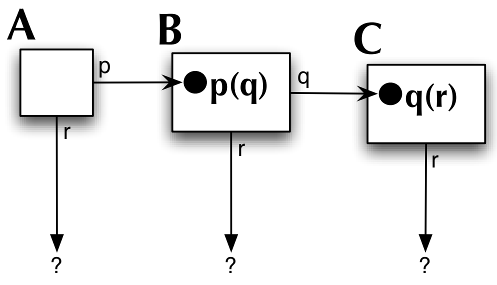

Figure 3: The uses relation carries through wires transitively.

The wiring of package `r` in the modules `A` and `B` depend on where `C` sources the package named `r`.

Assuming that `C` wires (imports) `r`, then both `A` and `B` must, independently, wire any import of `r` to the same place as `C` does.

If `C` did not import `r` then it could instead *export* it. This would mean that `C` 'gets' its package `r` from itself (its own class-loader, in fact) and so `A` and/or `B` would need to wire to `C`'s export of `r`.

There is one remaining possibility: that `C` neither imports nor exports a package named `r`. In this case, it would be presumed that `r` is an internal, private, package used in `q` (in `C`) but not available elsewhere, so it is not possible to wire any imports in `A` or `B` to it. Thus any such imports could not be resolved.

Notice that even if `B` does not import `r`, the constraint on where `A` gets `r` from still applies. The constraint on importing `r` gets associated with `q` in `B` and passes to `A` along the uses constraint of the export of `p`.

5.2 Uses constraints - formulation

In order to precisely describe uses constraints we want to impose a non-local condition, just as we did for *AllImports* above.

All previous constraints have been independently imposed, so we can apply them in combination as we think fit, but the uses constraint appears to be hard to apply by this means. The problem resides in the fact that the constraint on wiring `r`, where `q` uses `r`, is associated with `q` and ‘travels through’ the wires on which `q` is exported.

We first attempt to understand when an exported package ‘uses’ another. This is a non-trivial relation which depends upon the wiring and the `uses` directives in the exported package. In general, it is a relation on package *names* and modules, which can be built upon any consistent wiring.

After developing the relation we can use it to precisely describe the constraints it implies. We will be describing these constraints upon a general ‘consistent’ wiring—we will not assume, for example, that any of the other non-local constraints are satisfied. This is a deliberate attempt to allow us to impose the non-local constraints in any order.

Usage across a single wire

The first step is to define the non-transitive ‘base’ of the relation. We introduce a shorthand for the relation carrier:

$$\mathit{UsesRelations} == \mathit{ModuleId} \times \mathit{PackageName} \leftrightarrow \mathit{ModuleId} \times \mathit{PackageName}$$

and use it to define the ‘base’ relation:

$$
\begin{array}{|l}
\hline
\mathit{UsesBase} \\
\hline
\mathit{ConsistentWiring} \\
\mathit{usesBase} : \mathit{UsesRelations} \\
\hline
\mathit{usesBase} = \\
\quad \{\ \mathit{OneWire};\ \mathit{pUsed} : \mathit{PackageName} \\
\quad \mid w \in \mathit{consistent} \land \mathit{pUsed} \in \mathit{uses} \\
\quad \bullet\ (\mathit{imid}, \mathit{pName}) \mapsto (\mathit{emid}, \mathit{pUsed})\ \} \\
\hline
\end{array}
$$

which records the relation of *immediate* usage. Each relationship $(m_1, p_1) \mapsto (m_2, p_2)$ in *usesBase* arises from an import of p_1 in m_1 wired to an export of p_1 in m_2, where the export declares p_1 `uses` p_2.

The definition generates these pairs from the wires in *consistent*. Each wire w in *consistent* establishes a possible (one-step) use of all the package names in the `uses` directive on the exported side. (Since the reference is quite remote, we remind the reader that *uses* is a set of package names exposed by *OneWire*, and is a component of the *ExportProvider* object in the target of w, a consistent wire of the *ModuleCollection*.)

Usage extended transitively

The transitive rule arises from cases like the transitive diagram in figure 3 on page 13. In that diagram we see two wires ‘chained’. In *UsesBase* we would see two relationships:

$$(\mathtt{A}, \mathtt{p}) \mapsto (\mathtt{B}, \mathtt{q})$$
$$(\mathtt{B}, \mathtt{q}) \mapsto (\mathtt{C}, \mathtt{r})$$

and we want to deduce that:

$$(\mathtt{A}, \mathtt{p}) \mapsto (\mathtt{C}, \mathtt{r})$$

which is to say, the package named `p` in module `A` uses (albeit indirectly) package `r` in module `C`.

The point is that from *UsesBase* we can deduce that imports of `q` in `A` must wire to the same place that `B` gets `q` from (either another wire from `B` or an export from `B`); and that imports of `r` in `B` must wire to the same place that `C` gets `r` from. We *cannot* infer that imports of `r` in `A` are constrained at all—that is why we need to ‘transitively close’ our relation, for that is what will properly describe the relationship.

The definition is actually rather simple:

$$\begin{array}{|l}
\hline
\textit{GeneralUsesExposed} \\
\textit{UsesBase} \\
\textit{usesGen} : \textit{UsesRelations} \\
\hline
\textit{usesGen} = \textit{usesBase}^{+} \\
\hline
\end{array}$$

and we can now dispense with the base relation:

$$\textit{GeneralUses} \mathrel{\widehat{=}} \textit{GeneralUsesExposed} \setminus (\textit{usesBase})$$

Applying the uses constraint

Having determined the general uses condition, we can now define how that constrains our wirings. Essentially we need to put into formal terms the rule we expressed above (“imports of `q` in `A` must wire to the same place that `B` gets `q` from”), bearing in mind that the “place that `B` gets `q` from” isn’t necessarily on a wire from `B` but might be an export from `B` (or might be *hidden* in `B`).

So, what is the constraint? We have a constraint for every relationship in *usesGen*. Let’s try to formulate one by itself:

```
┌─ UsesConstraintOne ─────────────────────────────
│ GeneralUses
│
└─────────────────────────────────────────────────
```