

SAT Based Dependency Resolution

Notice

This posting evolved as a problem brainstorm, not a logical narrative. I did not expect the end result to be positive and am still continuing deliberations.

Contents

- [Contents](#)
- [Preamble](#)
- [Tree Builder](#)
- [Tree Conflict Resolver](#)
- [Resolution Contributors](#)
- [OSGi Resolution](#)
- [OSGi dependency resolution details](#)

In Mercury reincarnation of the Artifact library we are trying out the new approach to the dependency conflict resolution. Plus we are modifying the way version ranges are treated.

Let's have a look at the design.

Preamble

If we have

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
  </dependency>
</dependencies>
```

It is now interpreted as **find 3.8.1 version** and provision it.

If we have

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>[3.8.1,4.0.0)</version>
  </dependency>
</dependencies>
```

Is interpreted as **find any 3.x.x greater or equal to 3.8.1 version** and provision it.

But what happens if we have two dependencies.



Or - in plain XML

t:a:1

```
<dependencies>
  <dependency>
    <groupId>t</groupId>
    <artifactId>b</artifactId>
    <version>1</version>
  </dependency>
  <dependency>
    <groupId>t</groupId>
    <artifactId>c</artifactId>
    <version>1</version>
  </dependency>
</dependencies>
```

and then t:b:1 lists it's dependencies as

t:b:1

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
  </dependency>
</dependencies>
```

and t:c:1 lists it's dependencies as

t:b:2

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>[3.8.1,4.0.0)</version>
  </dependency>
</dependencies>
```

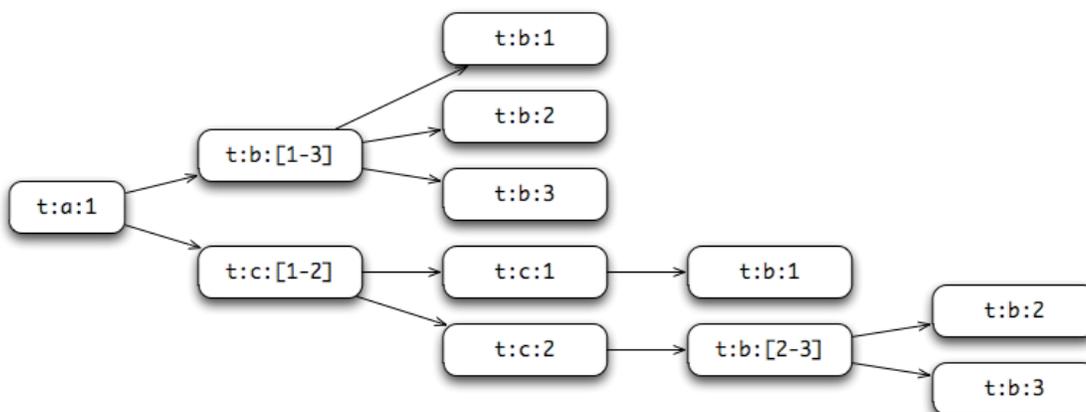
Tree builder will create a list of all available 3.x.x versions, let's say it's 3.8.1 and 3.8.2, because they are all on the same level in the tree and if our policy is "newest wins" - junit 3.8.2 will be selected. So we see, that, despite the fact that **t:b:1** declares strict dependency on junit-3.8.1, it get junit-3.8.2 when resolved. This maybe trivial for Maven users, but sometimes causes surprise among people, not familiar with dependency management basics.

And what happens if one of the ranges on the same level of the tree brings in junit-4.4, while **t:b:1** is not compatible with it? [Should Maven fail this build or still try to resolve conflicts?](#) We need some sanity rules around our resolution process. Plus - [should we try to bump up a dependency version even if there are no newer dependencies in the dependency tree?](#) Maybe some kind of **target platform** descriptor with a set of versions available in it?

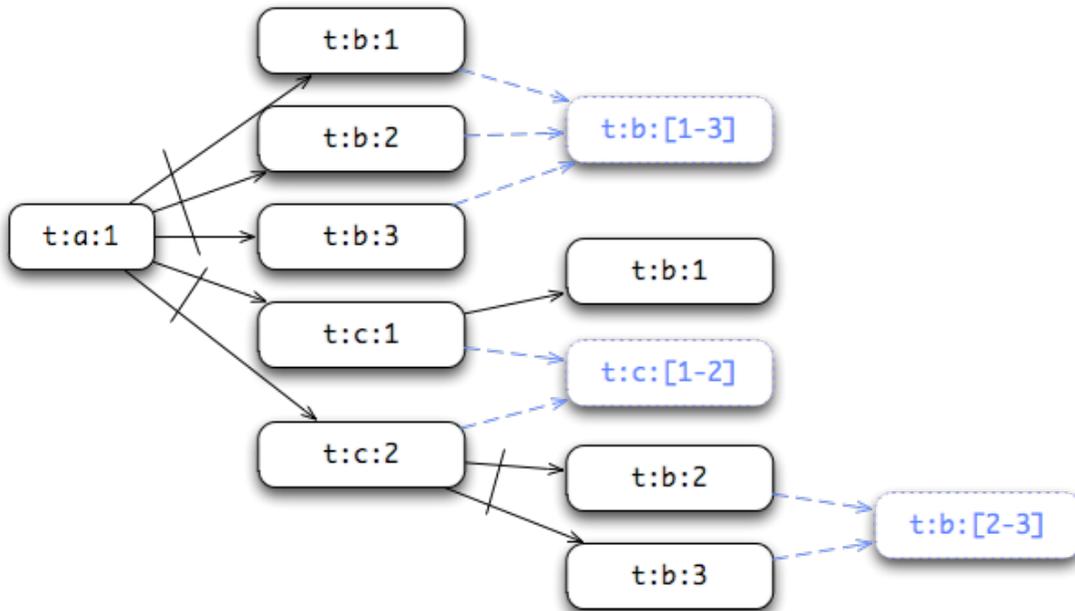
BTW - in OSGi world the case above may be resolved differently - because neither **t:b:1** nor **t:c:1** re-export those dependencies and could safely use different versions in their respective classloading fortresses.

Tree Builder

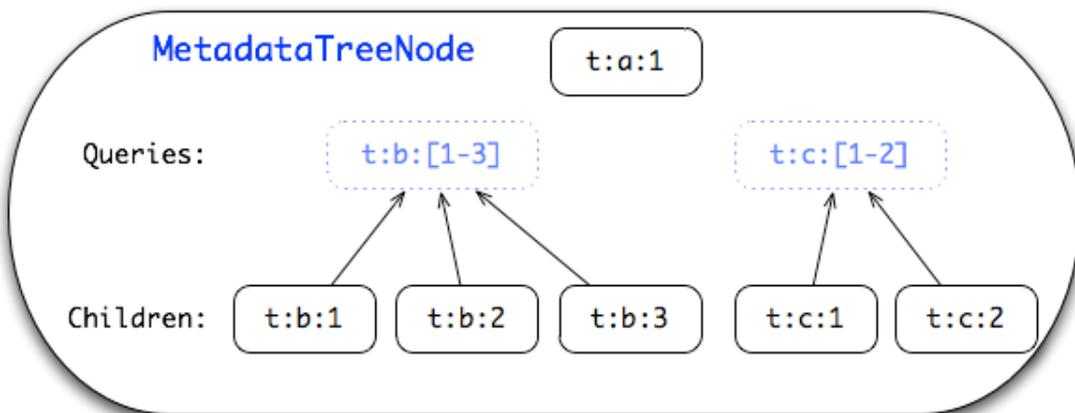
Let's switch from the benign examples above to more interesting cases.



Here we have the same parent with two dependencies, but each dependency is now a range. This leads to a simple **AND-OR** tree (graph?), where each node also has a reference to a query it resulted from and **OR** links are crossed by a line as below:



And this structure is kept in the tree by the means of MetadataTreeNode object



Where **Queries** is nothing else but the <dependencies> element contents from Arifact's POM. Each **dependency** i.e . **query** is then expanded into children list via calling **MetadataSource**. Various implementations of the latter are supposed to return back all available artifacts, matching the query criteria.

Tree Conflict Resolver

Now, as the tree is stuffed with all possible artifacts, it's time to start removing duplicates. This process is also known **conflict resolution**

Previous attempts to use standard graph-based algorithms proved to have limited success due to the fact that there is no polynomial time solution to this problem - it is not an NPC (NP Complete) problem. I tried to use dynamic programming based heuristics, but that immediately became too complex.

Since mid 90-ties first academia, then commercial entities, started applying elements of AI research to these problems. In particular - Satisfiability Solvers (SAT Solvers). Basic problem is - given a boolean function

$f(x_1, \dots, x_n)$ find boolean values x_1, \dots, x_n such as $f(x_1, \dots, x_n)$ turns true. With n growing, this problem turns NP.

Our problem is described in this terms as:

- each artifact version is mapped into a variable
- the tree structure is converted into a set of pseudo boolean **clauses**
- all version ranges are converted into pseudo boolean **atmost(1)**
- conflict resolution policy (newest/shallowest and the like) is represented as **objective function** to be minimized by the SAT Solver
- Solver gives back a list of **true** variables, that are mapped back to Artifacts
- if there is no solution, solver throws back **contradiction exception**

This presents some challenges

- we don't know which exactly artifacts was selected
- explanation function exists in the solver but yields nothing so far
- contradiction exception does not say what exactly contradicts what, we need more research here

Resolution Contributors

SAT solver calls the so called **objective function** selecting this or that solution based on the number, returned, trying to minimize it.

For classic Maven conflict resolution policy, this function should operate on two dimensions:

- depth
- version

On the same depth version decides, otherwise depths decides the output.

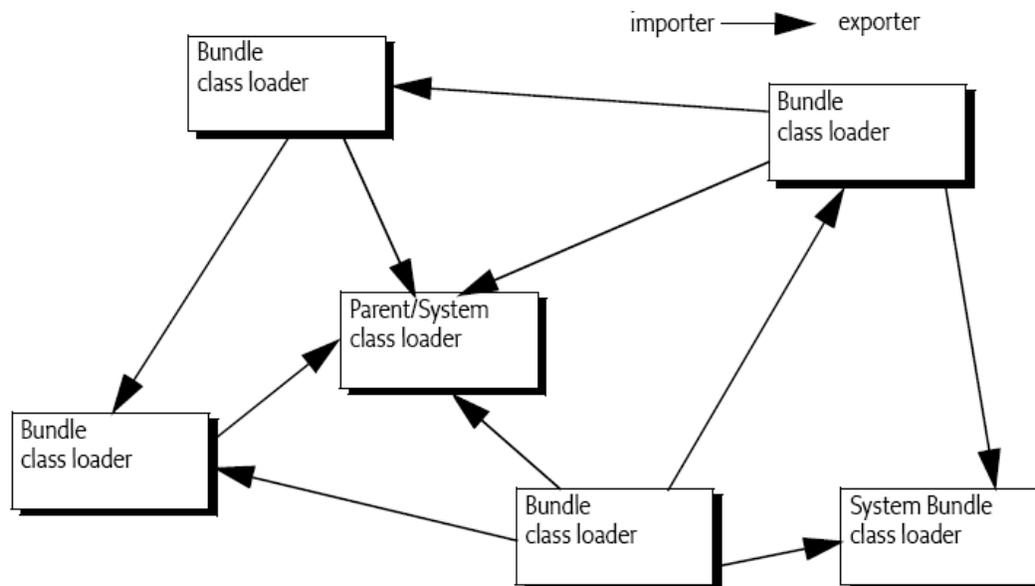
To make this process more flexible - we will introduce two types of contributors to the resolution process:

- **ArtifactFilter*s* - the ones that are called during tree building. They will be able to veto any artifacts from being added to the tree
- **ArtifactSorter*s* - contributors, that define preferences during conflict resolution. They just favor this or that artifact, but cannot completely ban it. So if no better solution exists - the bad one will be selected.

The process is also enhanced by the ability to chain **MetadataSource** implementations, so when m2eclipse needs to make our new resolver aware of the workspace projects - it simply inserts the specific **MetadataSource** at the head of the chain

OSGi Resolution

From OSGi 4.1 spec



As we see here - OSGi model is a true graph of versioned binaries a.k.a. **bundles**. Major deviation from Java hierarchy of classloaders is - well - the graph of classloaders. It is possible, although very unlikely in reality, to have different versions of the same bundle in various parts of the graph. We can have the same picture in, say, webapp variation of plain java classloading - when classloader checks it's own cache before asking the parent classloader.

In OSGi - this happens differently - classloaders are **wired** together via package export/import declarations - **wires**. When looking for a class - classloader just follows the wire - graph edge - for that package.

This is still a friendly turf for Maven dependency management, we can calculate all the wires for any given bundle relatively easy. Trouble comes if we are given a problem of calculating entire OSGi runtime environment - full graph. In this case we can calculate non-conflicting in Maven sense (no duplicates) set for each bundle, but wiring them all together requires a different skill, not our tree-based resolver.

So looks like we can use our resolver in the bundle build process - to provide all the binaries to compile a bundle. But we cannot (yet?) address wiring of the OSGi runtime.

OSGi dependency resolution details

There are 5 distinct ways of specifying an OSGi dependency:

- require bundle
- require bundle where resolution:=optional
- import package
- import package where resolution:=optional
- DynamicImport package

The first two are almost the same as classical Maven model. Difference is that they can contribute their packages to some other wires.

The last three request a single versioned package. Dynamic import is supposed to work during runtime classloading and **is not clear to me now**. This is addressed in Maven tree resolver via an additional set of clauses, that describe relationships between bundles and packages.

The OSGi resolver shapes up as the following:

- each package version is mapped into a variable
- all packages are grouped into versioned bundles. [This piece is the only difference from classical Maven](#)
- the package tree structure is converted into a set of pseudo boolean **clauses**
- all version ranges are converted into pseudo boolean **atmost(1)**
- conflict resolution policy (newest/shallowest and the like) is represented as **objective function** to be minimized by the SAT Solver
- Solver gives back a list of **true** variables, that are mapped back to Artifacts
- if there is no solution, solver throws back **contradiction exception**

The OSGi resolution problem is somewhat more complex compared to Maven (classical) and SAT suits the purpose perfectly.