Maven 2.1 Artifact Resolution Specification

Notes to work out in later sections:

- Graph-based artifact resolution
- Decouple from Maven's core
- Binary graph that is pre-resolved for a POM
- Artifacts should never be automatically updated, the update policy should be never by default
- Repository defintions forbidden in POMs. Move toward having teams setup with the repositories they need. So that all artifact resolution processes start with a known set of repositories. This radically simplifies artifact resolution and avoids all chicken-egg problems. Teams in OSS or corporate must move toward using a proxy and/or have their settings.xml file synced up for the team. This also avoids all the crap of putting repositories in for snapshots and then taking them out. This will make the process deterministic.
- The central repository definition is also removed from the SuperPOM and the repository definition moves to a standard settings.xml file. This way all repository definition are moved to a place where they can be easily altered. No more "mirrors" and replace versus cascading. It's all upfront and the set of repositories are fully defined before resolution. Yes this requires some stringent upfront checking to make sure things are intact but removes all question as to how resolution works.
- api/impl specifications - what about conflicting impls (instead of versions)
- exclusions!!! how far to they reach? diff between dep and depMgt?

Dependency Conflict Resolution

Introduction

The version, scope, classifier, type to use for a particular dependency depends on a number of choices. For each possible location where dependency information can be specified, several choices have to be made.

For the sake of simplicity, we'll focus on the version property of a dependency only.

Simple case: direct dependency.

As a first example, an indication of the number of versions to consider is given for this simple graph;

```
P
 ^
 A -> B
```

The information for the dependency B can come from several places:

- a dependency declaration in A
- a dependencyManagement declaration in A
- a dependency declaration in any profile in A
- a dependencyManagement declaration in any profile in A
- a dependency declaration in P

- a dependencyManagement declaration in P
- a dependency declaration in any profile in P
- a dependencyManagement declaration in any profile in P

The number of possible versions for B is quite big, even for this simple one-project-one-parent-one-dependency graph.

If A only declared a direct dependency on B, and P contains a depMgt section with B declared in there,
we'd have 2 versions to consider. If both A and P contain both the dependency and the dependencyManagement for B, we'd have 4 versions to consider. If there were profiles, each profile could contribute up to 2 versions (from the dependency and the dependencyManagement).
For N profiles in A, and M profiles in B, the maximum number of versions to consider is:
(N + 1 + M + 1) * 2 = 4 + 2N + 2M.
(for 3 profiles in A, and 3 in P, both having a dep and a depMgt entry for B, we'd have 16 versions already!)

Formalization

We'll now formalize the maximum number of versions to consider for a dependency in any graph.

- The simplest case is a direct dependency, pom A and pom B, no parents, no dependencyManagement sections.
  f(s, d) = 1 (where a is the starting point of resolution, and b is the dependency)
- When dependencyManagement comes into play, the function becomes:
  f(s, d) = 2
- When profiles come into play, this becomes:
  f(s, d) = (1 + np(s)) * 2 (np(s) = number of profiles for a; the 1 indicates the project a itself, the np(s) the additional profiles for s)
- When parent poms come into play, the maximum number becomes:
  f(s, d) = ne(s) * (1 + np(s)) * 2 (ne(a) is the number of 'extends' (parents) of a)
- When transitive dependencies come into play, say A -> B -> C, the function becomes:
  f(s, d) = SUM(a : dt(s, d)) (ne(a) * 2 * (1 + np(a))) (dt(a, b) gives a collection of all the nodes, from a to b (excluding b); in our case A and B)

Since this'll get complex very soon, we aren't going to focus on the exact number of versions
to consider, as this is higly dependent upon what is specified where.
Instead, we'll focus on the maximum number of versions to consider.
These formalizations are done to give an idea of the complexity of version resolution.

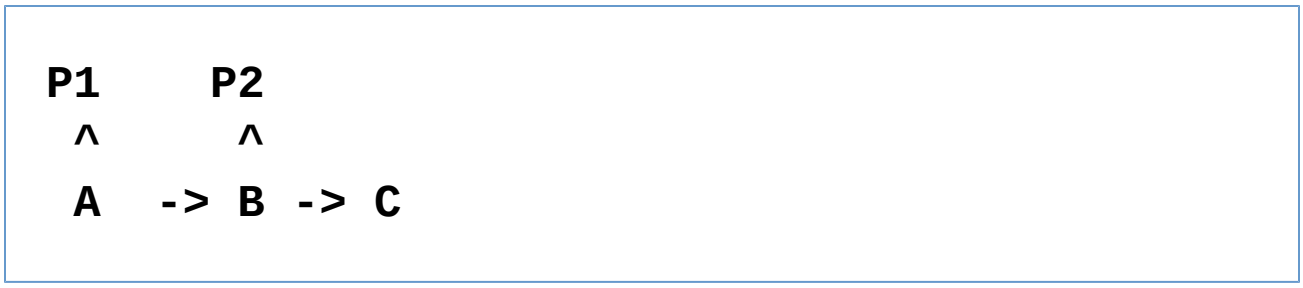
All-covering case.

So, the problem is rather complex.
We have several aspects that contribute to the complexity:

- the inheritance hierarchy
- the dependency trail
- depMgt vs dependencies
- profiles
- multiple trails

We have 5 orthogonal axes to apply to a single node to find the right version.
As I said before, each location that declares dependency poses a question: which one to take?

Here's a graph that illustrates te first 4 aspects:
The following



where all of P1, A, P2, B have:

- a dependency declaration on C
- a dependencyManagement declaration on C
- two profiles, both having a dependency declaration on C and a dependencyManagement declaration on C.

The 5th aspect isn't demonstrated by this example as there's only 1 trail. The section covering multiple trails will give a specific example of the multiple trails problem.

We'll now work out all of the 5 aspects.

Inheritance Hierarchy

This represents the trail from a pom A to the root pom, using the 'extends' relation.
In our example, the inheritance hierarchies are [A, P1] and [B, P2].

Inheritance hierarchy consolidation is done by Maven in the very earliest stages after reading the pom.
To resolve dependency C from A, after the inheritance hierarchy is consolidated, only A and B are considered,
The information from P1 is merged with A c.q. P2 is merged with B, so all information from parent poms are somehow injected into A.
What results, is a single POM containing all the information, where conflicts between the POM and it's ancestors have already been resolved.

When merging parent poms, there can be conflicts, if the same kind of information is present in both the parent and the child.
The question to be answered here is: which one overrides, child overrides parent or vice versa?
As it turns out, this is not the same for all POM elements. The elements that affect dependency resolution are:

- /dependencies
- /dependencyManagement
- /profiles/profile/dependencies
- /profiles/profile/dependencyManagement
- /profiles/build/plugins/plugin/dependencies
- /build/plugins/plugin/dependencies
- /extensions
  The precedence of either parent or child on any of these elements will affect dependency resolution.

TODO: create a table listing these elements and stating wheter 'parent wins', or 'child wins'.

Note that inheritance warrants a separate chapter, and sub-specifications should direct how to merge all the elements.

This section does NOT specify which of the 6 locations for a dependency 'wins'. This is done in subsequent chapters.

The relationship between all the elements will be described in a later section.

Dependency trail

The dependency trail in our example is A->B->C, where only A and B are considered since C is the dependency itself.

Resolution starts with POM A. At that point, there may or may not be information present about C. For the purposes of fleshing out this specification, we'll assume that *all* POMs and *all* elements that could possibly contain information about the dependency, do in fact contain that information.
So A contains information about C. And also about B. When B is resolved, the information present in B about C becomes
available to the dependency resolver.

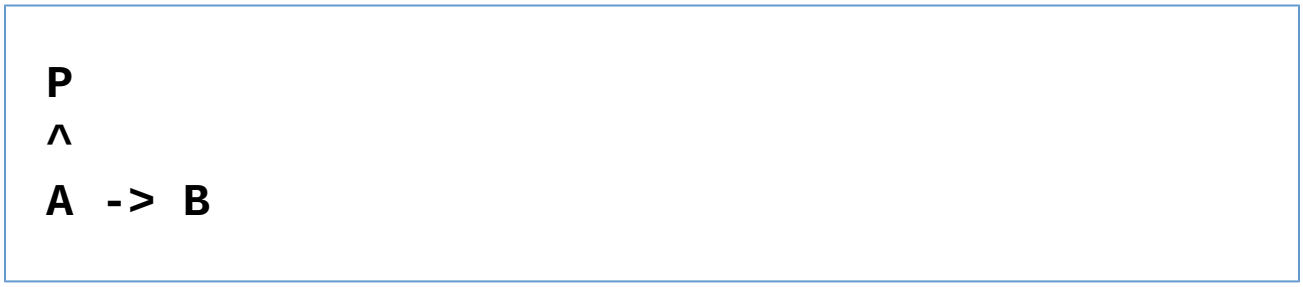
The decision to be made here is: which takes precedence, the information in A, or the information in B?
Or, more generally: which one overrides, the one closer to the start of the resolution process (A), or
the one closer to the final dependency (B).

DependencyManagement vs. Dependencies.

What if a POM, after inheritance, or before, contains conflicting information about a dependency
in the dependencies section and the dependencyManagement section?

Should dependencyManagement only provide defaults, or should it force overrides?
Should the distance between the dependency and the dependencyManagement have any influence on which one takes precedence?
Should parent depMgt override childDepmgt or vice versa?

Consider this:



The following combination of cases are possible:

- P declares dependency on B
  - P declares depMgt on B
    - A declares dep on B
      - [1] A declares depMgt on B
      - [2] A does NOT declare depMgt on B
    - A does NOT declare dep on B
      - [3] A declares depMgt on B
      - [4] A does NOT declare depMgt on B
  - P does NOT declare depMgt on B
    - A declares dep on B
      - [5] A declares depMgt on B
      - [6] A does NOT declare depMgt on B
    - A does NOT declare dep on B

        - [7] A declares depMgt on B
        - [8] A does NOT declare depMgt on B
- P does NOT declare dependency on B
    - P declares depMgt on B
        - A declares dep on B
            - [9] A declares depMgt on B
            - [10] A does NOT declare depMgt on B
        - A does NOT declare dep on B
            - [11] A declares depMgt on B
            - [12] A does NOT declare depMgt on B
    - P does NOT declare depMgt on B
        - A declares dep on B
            - [13] A declares depMgt on B
            - [14] A does NOT declare depMgt on B
        - A does NOT declare dep on B
            - [15] A declares depMgt on B
            - [16] A does NOT declare depMgt on B

So we have 16 cases, all could have a maximum of 4 possible versions for B, which gives us 64 possible algorithms, even in this simple example.

Most cases do not have 4 possibilities though. There are:

- 8 functions with only 3 (a),
- 4 functions with only 2 (b),
- 2 functions with only 1 (c)
- 1 function with 4 possibilities (d).
- the rest is invalid.
  That leaves us with 38 possible functions (implementations).

(note: the tree above does not match this table)

winner	**P depMgt**	**P dep**	**A depMgt**	**A dep**	
					0 x
Ad				X	1 (c)
			X		2 x
Am,Ad			X	X	3 (b)
etc etc		X			4 (c)
		X		X	5 (b)
		X	X		6 (b)
		X	X	X	7 (a)
	X				8 x

	X			X	9 (b)
	X		X		a x
	X		X	X	b (a)
	X	X			c (b)
	X	X		X	d (a)
	X	X	X		e (a)
	X	X	X	X	f (d)

The algorithm would have effect on several dimensions/axes/aspects of the problem, that's why there are so many possible implementations.

I strongly advice NOT to write algorithms that span different aspects. In this case, this means an algorithm that solves a single-aspect problem (depMgt vs. dependencies), by taking into account information from other aspects (being the inheritance hierarchy aspect).
As you can see, this complicates things too much. Also, in other situations, for instance with profiles, we'd also have a specific 'cross cutting' algorithm, taking into account more than 1 aspect.
It's easy to have 2 algorithms (one that solves the depMgt vs. dep problem, and one that solves the profiles vs. model problem), to conflict with each other, or at least provide very inconsistent behaviour. I'm afraid that Maven currently suffers from this problem.

So, aside from my suggestion, let's continue on the algorithm on an aspect by aspect case.
In this case that means we let the inheritance hierarchy algorithm remove the complexity of having a parent pom. We then only have 2 locations to consider: depMgt and dependencies, both in A.
This decreased the complexity of this aspect alone with a factor of 16; imagine what it does to other aspects, and the combined complexity! Divide and conquer.

So, we've reduced the complex problem to a simple question: should depMgt win, or dependencies, no matter what the circumstances?

We're losing some edge cases here with this simplification. The two possible algorithms are:

- depMgt used for 'defaults'
- depMgt used as overrides.

We're losing some abilities by this simplification. Here's one possible algorithm we won't be able to support:

- say we have a pom A extending from P, and A is declaring a dependency. The algorithm enforces all these rules:
    - If P declares a depMgt (and A does not), this depMgt is used as a default, and doesn't force transitive deps. Only applied to inheritance.
    - If A declares depMgt (and P does not), this depMgt is used to force the version, even of transitive deps.
    - If both A and P declare depMgt, the depMgt from P is used as a default for direct dependencies, and the depMgt from A is used only for transitive
      dependencies and will not override the depMgt from P.

Do we want to support something this complex?

Profiles.

Multiple trails

This is an odd one. Consider 5 POMs, A, B, C, D, and E.

where:

- A -> B -> C -> E
- A -> D -> E

When all the rules are applied, we can end up with 2 versions of dependency D - the one from C and the one from D.
The previous rules only solved conflicts for a single trail, not for 2 or more trails joining.

In this case, we can provide 2 strategies: 'nearest-winst', or 'newest-wins', where 'nearest-wins' falls back to 'newest-winst' in case the 2 trails are equally long (A->B->D and A->C->D).

Influence of different locations declaring dependency information

As stated earlier, the locations in a POM that can contain dependency information are:

- /dependencies
- /dependencyManagement
- /profiles/profile/dependencies
- /profiles/profile/dependencyManagement
- /profiles/build/plugins/plugin/dependencies
- /build/plugins/plugin/dependencies
- /extensions
  In fact, there are several locations above that can't contradict eachother, since their information is used in parallell with the other information.
- plugin dependencies contribute to a plugin's runtime environment and have no relation to the project dependencies.
- extensions contribute to the environment of the currently building project, and have nothing to do with project dependencies.
- profiles are merged with the POM itself, as is described in a later chapter.
- dependencies and dependencyManagement are used to resolve project dependencies.

So we have 3 resolution processes: extensions, plugin dependencies, and project dependencies. The first two contribute to the maven environment, the latter contributes to the project's environment.

The relationships between the locations are as follows:

- dependencyManagement influences dependencies
- profiles/dependencies influence dependencies (if profiles take precedence over the model dependencies)
- profiles/dependencyManagement influences dependencyManagement (if profiles take precedence over the model depMgt)
- profiles/build/plugins/plugin/dependencies influence build/plugins/plugin/dependencies (if it takes precedence).
- dependencyManagement influences extensions
- dependencyManagement influences plugin dependencies

The influence of profiles on the model has been described, as has the influence of depMgt on dependencies.

There's only 1 type of dependencyManagement, so if we want that kind of functionality available for extensions and plugin dependencies, we'll have to use the dependencyManagement section for that.

Combining all of them.

Making only 1 decision in any of the above sections will NOT yield the final information about C.
All of the decisions above have to be made, and what's more important, they have to be combined.
So, added to the complexity of each individual decision, we have to specify the order the decisions are made.
Each decision eliminates one dimension/axis/aspect, so only the order of these decisions will be sufficient to get a final result.

The proposed (and probably currently implemented sequence) is this:

- Inheritance consolidation - this leaves us with only direct POMS; it removes a dimension from the graph.
- Profiles - this leaves us with only plugin deps, extensions, dependencies and depMgt
- depMgt vs dependencies - this leaves us with only plugin deps, extensions, and dependencies
- the dependency trail - this leaves us with one version per trail
- multiple trails - this decides on one version out of all the trails.

Snapshots and plugins - Other sources of information

So far, the only locations where version information about artifacts/dependencies could be stored, was the POM (or profiles in profiles.xml/settings.xml - anyway, information that ultimately ends up in the POM itself).

Snapshots however, add a new location for version information: remote repositories.
As do plugins, as the version isn't required in their declaration. To make matters worse, plugins can also define profiles.

Both snapshots and plugin declarations don't have a final, concrete version: plugin versions are optional, and the latest version will be retrieved from a repository. Snapshot versions containing the 'SNAPSHOT' keyword aren't final either - their version will also be fixed using information in a repository (local or remote).

Snapshots and plugin declarations have no influence on the described aspects themselves, but they do add another aspect: another source of information.

Artifact Resolution and Repositories

We have several things to resolve:

- project dependencies
- plugin dependencies
- extensions

They should not conflict with eachother - a dependency from the project cannot affect a dependency in a plugin and vice versa.
This also affects repository settings.
While resolving any of the abovementioned artifacts, the only common context can be the POM, which must not be changed. This will
make sure nothing contaminates the other resolution processes.

In Maven 2.1, all repositories will be known beforehand. Repository declarations in POM's will no longer

be used; they serve the same purpose as optional dependencies.

Resolving snapshots

Project dependency resolution has been discussed. We'll now add snapshot resolution to the mix.

When the normal project resolution has done it's job, and it results in a SNAPSHOT version,
repositories will be consulted and the actual version will be filled in for that artifact alone.
NOTE that $

Unknown macro: {project.version}

will keep evaluating to the SNAPSHOT version, not the timestamped version,
so that each possible dependency can be resolved to the proper version.
So this step will come after the normal project dependency resolution, as the last step
after the steps described in 'Combining all of them' (TODO: anchor link).

Resolving extensions

Extensions are resolved just as project dependencies. When decisions are made
while traversing dependency trails etc. about certain dependencies, these decisions
will only last as long as is needed to resolve extensions. When project dependencies
or plugin dependencies are resolved, all this information will be forgotten. Beware of
caching!
If a project has a dep on B 1.0, and an extension specifies B 1.1, then they should have
no influence on eachother and both dependencies should be used - one for maven, one for the
project. They pose no conflict. Transitive dependencies can also pose no conflicts.

Resolving plugins

The same as stated under 'Resolving extensions' applies here, with the following addition.

First, the project resolution is applied to plugins.
If no version can be found, instead of failing (like is done for normal dependencies),
plugin versions are resolved from plugin repositories.
Do we want to keep supporting this, or do we want to fail, i.e. require plugin
versions to be set?

What do we want to do with groupId - that's another source of information.