Dependency Mediation and Conflict Resolution

This document describes the rest of the requirements for dependency management that have not yet been implemented for Maven 2.0, especially with regards to transitive dependencies.

Design

Version comparison requirement

The effectiveness of the above ranges reqires that we are able to compare two versions and determine which is newer. This should be designed such that it is pluggable, but initially we should only provide one scheme and attempt to use it uniformly.

Default Version comparison definition

The default specification should be composed as follows:

```
<major>.<minor>.<revision>([ -<qualififer> ] | [ -<build> ])
```

where:

- the qualifier section is optional (and is SNAPSHOT, alpha-1, alpha-2)
- the build section is optional (and increments starting at 1 if specified)
- any '0' build or revision elements can be omitted.
- only one of build and qualifier can be given (note that the timestamped qualifier includes a build number, but this is not the same)
- the build number is for those that repackage the original artifact (eg, as is often done with rpms)

For ordering, the following is done in order until an element is found that are not equal:

- numerical comparison of major version
- numerical comparison of minor version
- if revision does not exist, add ".0" for comparison purposes
- numerical comparison of revision
- if qualifier does not exist, it is newer than if it does
- case-insensitive string comparison of qualifier
    - this ensures timestamps are correctly ordered, and SNAPSHOT is newer than an equivalent timestamp
    - this also ensures that beta comes after alpha, as does rc
- if no qualifier, and build does not exist, add "-0" for comparison purposes
- numerical comparison of build

Note also the proposed extension from a user in an rpm environment: Extending Maven 2.0 Dependencies

Dependency Version Ranges

Need to be able to declare minimum, maximum allowed versions of a dependency (both min and max may be optional), and allow "holes" for known incompatible versions.

Proposed syntax:

Range	Meaning
`(,1.0]`	x <= 1.0
`1.0`	"Soft" requirement on 1.0 (just a recommendation - helps select the correct version if it matches all ranges)
`[1.0]`	Hard requirement on 1.0
`[1.2,1.3]`	1.2 <= x <= 1.3
`[1.0,2.0)`	1.0 <= x < 2.0
`[1.5,)`	x >= 1.5
`(,1.0],[1.2,)`	x <= 1.0 or x >= 1.2. Multiple sets are comma-separated
`(,1.1),(1.1,)`	This excludes 1.1 if it is known not to work in combination with this library

Mathematical syntax chosen to avoid the use of - as it would conflict with what is used in many version number, and because < and >= doesn't look nice in XML. (,1.0] is used because infinity is not really helpful here.

Default strategy: Of the overlapping ranges, the highest soft requirement is the version to be used. If there are no soft requirements inside the prescribed ranges, the most recent version is used. If that does not fit the described ranges, then the most recent version number in the prescribed ranges is used. If the ranges exclude all versions, an error occurs.

Addition of ranges leads to additional necessary specifications on the dependency element.

Incorporating `SNAPSHOT` versions into the specification

Resolution of dependency ranges should not resolve to a snapshot (development version) unless it is included as an explicit boundary. There is no need to compile against development code unless you are explicitly using a new feature, under which the snapshot will become the lower bound of your version specification. As releases are considered newer than the snapshot they belong to, they will be chosen over an old snapshot if found.

It is possible that applications such as Continuum may have a mode that enables always resolving to the snapshot version, but this is external to the POM itself.

Conflict Resolution

Strategies for conflict resolution need to be pluggable and stackable.

In some cases, the resolution of a version may be workable, but not ideal. For example instead of erroring out, we may just drop to a different method of conflict resolution if there is no match in the first method (ending up using the current nearest wins algorithm).

Some suggested techniques would be:

- use version specification (as defined above, default)
- use nearest (always get closest transitive dep regardless of version specifications)
- fail if no match found

I'm inclined to say the strategy should be per project instead of per dependency. This is to avoid a large amount of duplication if you want to use it across the board.

There is also the question of whether you can apply it globally, or whether you have to honour the setting of the original project as you traverse the transitive dependencies. At this point this isn't deemed necessary.

Forcing a version

A version will always be honoured if it is declared in the current POM with a particular version - however, it should be noted that this will also affect other poms downstream if it is itself depended on using transitive dependencies.

Improved Diagnostics

It is important that we are able to easily and clearly represent the state of a projects dependency tree both from m2 and graphical tools given the added complexity of transitive dependencies and version management. The diagnostics should as much as possible explain why a version was chosen.

Implementation Details

The actual dependency calculation logic needs to be made into a separate plexus component so that other visualisation tools can use the same code.

The Dependency Management element

This is inherited, and gives a full profile over your dependency set. While it will not add new dependencies itself, it will enforce the following constraints:

- your policy on the version allowed for a dependency
- affects both your declared dependencies and transitive dependencies
- apply a specific scope to a dependency

Resolve Dependencies once only

This is a relatively straightforward task in the resolver:

- eg if maven-artifact is in the top level pom, go through the tree and find all the references to it, decide the version, and then use that version's dependencies, not the others
- this would probably be done with a DAG using just group/artifact ID with counts on the edges. When a version is chosen all the other versions are removed, and when a count drops to 0 it is removed from the graph. Proceed until all versions are determined.
- this ensures the pool of dependencies is smaller when trying to determine the version to use, and much more accurate.

Preventing `RELEASE` dependencies

Given that an open ended version specification such as `[1.0,)` will use a `RELEASE` version dependency if available, there is no reason to allow and resolve such a version for a dependency.

Version specifications for Parents

Parent references are handled differently to dependencies. They do not need to incorporate ranges, and are generally treated as unversioned. See the [Release Management](#) document for a discussion of this.

Reproducibility

To make sure releases remain reproducible, it is necessary to keep all of the information determined to make the release, removing ranges, snapshots and parent references. However, it is also desirable to retain the original information and that is what should be deployed to the repository so that when it is later used a more intelligent decision can be made about what might be required. If the versions were resolved in the remote repository, the benefit would disappear as instead of transitively getting something in a given range, or the latest snapshot, the version resolved at release time would be retrieved.

The process for a release will be:

1. modify version in POM to release version (as is done now)
2. add a wholly resolved pom under the name `release-pom.xml`
3. commit, tag
4. remove `release-pom.xml`
5. bump POM version to next dev version (as is done now)
6. commit

This means that the presence of `release-pom.xml` indicates a released version, and it should be present on a tag and in a distribution bundle. Maven should recognise its existence and use it instead, disabling transitive dependencies. This behaviour could potentially be changed by a CLI parameter.

The reason for selecting a separate file was so that the diffs against the POM were not too large, and so releases could be easily compared by comparing versions of `release-pom.xml`. It also easily lets Maven select how to operate with it.

Everything is resolved in the released POM: all plugin versions will be filled in, dependency ranges and snapshots (if allowed) replaced by the actual versions, any transitive dependencies included, and the parent reference removed as all elements from the assembled model will be included in the release POM.

Scope resolution

This is how it works in 2.0.x when two dependencies have different scopes in the graph

nearest\farthest	compile	provided	runtime	system	test
compile	compile	compile	compile	compile	compile
provided	compile	provided	runtime	provided	provided
runtime	compile	runtime	runtime	runtime	runtime
system	compile	system	system	system	system
test	compile	test	runtime	test	test

Note that there are some bugs that may prevent this behaviour like [MNG-1895](#)