Project Dependency Trees schema

Status	DRAFT
Version	
Issue(s)	
Sources	
Developer(s)	Stephen Connolly

Status

This RFC is currently in the DRAFT state. Nothing in this RFC has been agreed or confirmed.

Contents

Introduction

The Project Dependency Trees artifact defines all the side artifacts of a project as well as each artifacts tree of dependencies. This can be used by consumers to decide what the consumers effective tree of dependencies is as well as allowing consumers to perform intelligent substitutions in the tree. By providing the entire tree we can reduce the number of requests a consumer needs to make in order to resolve all the artifacts the consumer requires.

There are a number of issues with the current Project Object Model used by Maven:

- We do not have a good way to evolve the model or even change the model version
- We do not have a good way to model the differences in dependencies for the individual artifacts that get deployed as part of the project
- We do not have a good way to augment the information if we are deploying artifacts from the project non-atomically
- The model is weakly specified with regards to conflict resolution and exposes Java native assumptions about conflict resolution.
- The model does not allow for reproducible builds while simultaneously allowing range specification for dependencies
- The model exposes build time information that is irrelevant to consumers

The aim of the Project Dependency Trees model is to resolve these issues.

Project Dependency Trees documents are intended to be machine generated based on build time information. Build tools not able to generate these documents solely from build information are considered in need for corresponding enhancements.

Model evolution

One of the top level elements of the Maven POM is the `modelVersion` element that specifies the model version for the POM. To date there have been two model versions `3.0.0` and `4.0.0`. In both cases, a critical issue for changing the model version is that older clients cannot parse the newer model. This required the forking of Central (which is why central is `repo.maven.org/maven2` because `4.0.0` was introduced with Maven 2 and Maven 1 clients could not parse the new model version)

Obviously, newer clients can always be written to parse older model versions, but given that Central is now a resource used by multiple build tools, not all of which run on the JVM or are maintained by the Apache Maven community, we need to ensure that any solution does not break the ability of other clients to **consume** the artifacts published to central.

NOTE: while we need to ensure that artifacts can be consumed by older clients, we do not have to ensure that the older clients get the exactly correct dependency tree. Rather we should make the best effort possible to give older clients as good a dependency tree as we can give them.

Model evolution will be handled in two ways, based on the type of client:

- Legacy clients are clients that are not aware of the Project Dependency Trees model
- Modern clients are clients that are aware of the Project Dependency Trees model, but need not necessarily be aware of the latest `modelVersion`s deployed by the newest version of Maven.

Legacy clients

Legacy clients cannot be aware of the Project Dependency Trees model. For this reason, any project that deploys a Project Dependency Trees model will also deploy a `modelVersion` `4.0.0` POM which is the best-effort translation of the Project Dependency Trees model for the primary artifact of the project.

Modern clients

As part of the process of evolving a Project Dependency Trees schema, each new version of the model will be accompanied by an XSLT transformation(s) that will be published into Central at a defined set of coordinates. This will allow a modern client to convert a schema - that is newer than the highest modelVersion it was built against - into the newest modelVersion that it supports. In general the XSLT transformation will essentially strip out elements that are not understood by older clients, though it is possible that more adventurous transformations may be included.

The rationale for choosing XSLT as the transformation... and consequently forcing the Project Dependency Trees model to be expressed in XML is that XSLT is currently the only cross-platform transformation engine available across the JVM, Ruby, .NET, C/C++ native code and JavaScript runtimes.

Artifact Dependency differentiation

The current POM provides a single scoped dependency tree that is then universally applied to all artifacts produced by the project. This does not align correctly with what the artifacts produced by a single project actually require.

To illustrate by example:

> *Consider a project that builds a Java Web application that can be run standalone or as part of an EAR. Under current best-practice we would advise separating the project into multiple modules:*
>
> - *A module to build the JAR file that contains the compiled code and corresponding resources*
> - *A module to build the WAR file for consumption as part of an EAR - this needs to be skinny as the common dependencies will be shared across all the modules within the EAR*
> - *A module to build the WAR file for standalone - this needs to be fat and is built from the skinny WAR by adding in the common dependencies*
>
> *There are other ways to skin this cat, but what we really want to have is that there is a single project that produces:*
>
> - *A JAR of the compiled code - we may want to reuse this*
> - *A skinny WAR which exposes transitive dependencies of the common dependencies that are required to be present in the EAR*
> - *A fat WAR which does not expose any transitive dependencies - perhaps other than the servlet container and JVM level requirements*
> - *A test JAR that allows for re-use of the unit tests of the compiled code*
> - *An integration test JAR that allows for extending and running the WAR acceptance tests*
> - *A source JAR for the main JAR*
> - *A javadoc JAR for the main JAR*
> - *A source JAR for the test JAR*
> - *A javadoc JAR for the test JAR*
> - *A source JAR for the integration test JAR*
> - *A javadoc JAR for the integration test JAR*
>
> *Each of these artifacts will have different effective dependencies, for example the test JAR will have a dependency on the main JAR and then a dependency on the test framework, etc.*
>
> *The way the `modelVersion 4.0.0` POM handled these different dependencies was via `<scope>` tags. The issue with scope tags is that the valid scopes become part of the model version and the information about which scopes apply to which artifacts has been lost to the build process by the time the artifacts is consumed by a consumer.*

To solve this issue, the Project Dependency Trees will list the effective consumption required dependencies of each artifact produced by the project.

NOTE the Project Dependency Trees will have no concept of scope. This may cause issues for artifact types where for example a different dependency is transitive during compilation compared with execution. The current thinking is that such situations will be exceedingly rare if they ever occur, and that as such, for least surprise, the consumer will have to configure their build tool to address this issue if it ever arises.

Non-atomic deployment

The primary driver of non-atomic deployment is the production of platform specific artifacts for the project. In this regard, the Project Dependency Trees model assumes that deployments will be at least atomic per platform. "At least atomic per platform" means that the initial deployment may include multiple platforms and subsequent additional deployments will be atomic per platform. For example:

> *The foo project produces some non-platform specific artifacts as well as artifacts for the `os-x`, `windows` and `linux` platforms. A build on a say `os-x` may be able to use cross-compiling tooling to produce artifacts for the `linux` platform (e.g. you can use `rpmbuild` on a mac, so you could create the RPM installer when building the project on **some** macs). Thus if we perform the release from a mac, our initial deployment will include the non-platform specific artifacts (e.g. the standalone WAR file for the web application) as well as some platform specific artifacts (e.g. the OS-X installer and perhaps the RPM & DEB installers for linux systems). At the `com.example:foo::1.0` coordinates we would deploy:*
>
> - *a POM for `modelVersion 4.0.0` compatibility*
> - *the Project Dependency Trees where the top level `<project>` tag **does not have a platformId**. There will be an `<artifacts>` tag as well as `<artifacts platformId="os-x">` and `<artifacts platformId="linux">` detailing the artifacts that were produced as part of the initial atomic deployment*
> - *the non-platform specific artifacts will be deployed in `com.example:foo::1.0` as they are associated with the `modelVersion 4.0.0` pom coordinates*
> - *a POM for `modelVersion 4.0.0` compatibility will be deployed at `com.example:foo:os-x:1.0` (which maps to GAV `com.example:foo-os-x:1.0` in the `modelVersion 4.0.0` coordinates) detailing the dependencies of the os-x artifacts (as different platforms are most likely to have the biggest differences in dependencies, it makes sense to give each platform its own modelVersion 4.0.0 POM to assist legacy consumers get as close to the correct dependency tree as we can)*
> - *the os-x specific artifacts will be deployed at `com.example:foo:os-x:1.0`*
> - *a POM for `modelVersion 4.0.0` compatibility will be deployed at `com.example:foo:linux:1.0`*
> - *the linux specific artifacts will be deployed at `com.example:foo:linux:1.0`*
>
> *Later, we perform a checkout of the tag from SCM on a windows machine and perform the build and deployment of the windows specific artifacts.*
>
> - *a POM for `modelVersion 4.0.0` compatibility will be deployed at `com.example:foo:windows:1.0` as following the pattern from above*
> - *the windows specific artifacts will be deployed at `com.example:foo:windows:1.0` again following the pattern from above*
> - *the original Project Dependency Trees cannot be redeployed as that would break the atomic deployment as well as breaking the* write once *principle of Maven release repositories, thus a Project Dependency Trees file will be deployed at `com.example:foo:windows:1.0` in this case the `<project>` tag **must have a platformId**, specifically `<project platformId="windows">` and there must be one and only one `<artifacts>` tag contained within the `<project>` tag, i.e. `<artifacts platformId="windows">`. The metadata for either `groupId` or `groupId:artifactId` - which can be updated - will, in addition to detailing the available versions, detail the `platformIds` available for each version.*

NOTE: as the platformId is the unit that separates atomically deployable components, it will be up to the tooling providers to agree on what values of individual `platformIds` mean for that specific tooling. The example above used high-level operating systems as `platformIds`, but without prejudice, we could equally have `os-x-10.10`, `os-x-10.9`, `linux-fedora-25`, `linux-fedora-24`, `linux-rhel-6`, `linux-centos-6`, `linux-ubuntu-12.04`, `windows-server-2012`, etc. Similarly we could have the platform differentiate in other ways, e.g. `java7`, `java8`, `android`, etc. Or perhaps the platform could differentiate artifacts that target different runtimes, such as `tomcat`, `jetty`, `weblogic`, `geronimo`, etc where the major difference in those platform specific artifacts is the dependency trees.

NOTE: while different projects can follow different conventions for what the different `platformIds` are used for, as the dependency's `platformId` is part of the dependency tree, the project can perform the appropriate mapping of its transitive dependencies platform identifiers into its own convention, so deviations in conventions between projects should not prove fatal.

Conflict resolution

The `modelVersion 4.0.0` POM mixes build time dependency specification with consumption time dependency specification. This has the effect of significantly complicating the dependency model within the POM:

- Dependencies can be specified in the POM directly
- Dependencies can be inherited from parent POMs
- Dependencies can be added via profiles
- Transitive dependencies need to be traversed and processed and built
- Versions can be specified in the `<dependencies>` section, or `<dependencyManagement>` or imported by a `<scope>import</scope>` dependency in the dependency management.
- Versions can be specified using a `${property}` which can cause confusion as depending on where the dependency comes from, the valid origins for the property to be used with property expansion can be unclear.

- Conflict resolution is by "POM" order where the "first" dependency wins... but also the "child" wins over the "parent"... but the parent's <dependencies> entries come before the child's!!!
- If "POM" order fails, then conflict resolution is by tree depth, such that nearest to the root wins.
- The author suspects that there is more unspecified (or perhaps weakly specified) behavioural madness...
- The end result of dependency resolution is typically a flattened list of dependencies

The above set of "rules" make it hard for other toolings to process the dependencies of a POM correctly, and consequently there are many many examples of real world POMs where various hacks have been used to tame the effective dependency tree in order to produce the required transitive tree for consumers.

The Project Dependency Trees simplifies the work of consumers by explicitly providing the fully intended resolved tree to be used by consumers. There is no requirement for a consumer of a project's artifacts to consult any transitive dependencies (though if the consumer has a better understanding of a specific transitive dependency `modelVersion` the consumer *may want to consult*, it does not have to).

The consumer is then free to decide how to resolve conflicts, and because the tree has been provided, in the event that conflict resolution requires dependency substitution, the tree can be pruned safely (whereas with a flattened list, safe substitution would not be possible as we could end up retaining orphaned transitive dependencies)

The consumer is also free to decide if conflicts need to be resolved at all. For example, an OSGi container can correctly manage multiple versions of the same module whereas the Java 9 modulepath can only have one version of any specific module. When a project produces a JAR artifact that contains both the OSGi module metadata as well as the Java 9 module info, it has no way of knowing whether the consumer will want to apply a "single version per module" rule or "all versions of each module" rule and nor should it, only the consumer can know how conflicts should be resolved.

Conflict resolution is also related to the next issue.

Version ranges and reproducible builds

One of the main issues with version ranges in the `modelVersion 4.0.0` POM is that they produce an irreproducible build, as the consumer will re-resolve the version range every time it builds the dependency tree, and as such may resolve a different version.

The utility of version ranges comes into play when performing conflict resolution. If a consumer has to pick a single version of each dependency, the range information allows that version selection to be performed safely... i.e. if I have transitive dependencies on `com.example:foo::[1.0,)`, `com.example:foo::[1.2,2.0)` and `com.example:foo::[1.1,1.4.5],[1.4.7,)` then I can construct the effective safe range of `[1.2,1.4.5],[1.4.7,2.0)` and select the appropriate single version.

The irreproducibility of version ranges is still somewhat of an issue though. We can resolve the irreproducibility of builds by recognising that it is really a trade-off choice that the consumer should make.

The consumer should be able to choose between:

- Selecting the lowest matching version in the range - i.e. should be stable
- Selecting the highest matching version in the range - i.e. to pick up bug fixes automatically
- Selecting the lowest matching version in the range that was actually resolved by a dependency
- Selecting the highest matching version in the range that was actually resolved by a dependency

The Project Dependency Trees model enabled consumers to make this choice by providing not only the version range but also the resolved version of each dependency. The version range can then be used to guide conflict resolution and the resolved version information can be used as hints to pre-select the exact version to use if the consumer wants a reproducible build.

Build time information

The Project Dependency Trees model removes all the build time information that was previously exposed from the `modelVersion 4.0.0` POM, thus there is no <build>, <profiles> or <reporting> sections.

This points to a legitimate concern about how to handle project inheritance while moving the POM beyond `modelVersion 4.0.0`. The solution here is to define two classes of compatibility.

- The `modelVersion 4.0.0` POM will always be deployed (at least until such time as there are effectively no more `modelVersion 4.0.0` consumers)
- The Project Dependency Trees provides for "best effort" forward compatibility with newer modelVersions in order to ensure that older clients can at least consume artifacts from newer model trees (the older consumers may have to apply hacks such as <exclusions> or explicitly listing required dependencies in order to consume the dependency correctly... just as a `modelVersion 4.0.0` POM consumer does today, but the artifacts can be consumed)
- The build time information is only required from parent / mix-in projects. To use a parent / mix-in you must be building with a tool that understands the modelVersions up to and including the highest modelVersion of the parent project and any mix-in projects

In other words:

- Parent and Mix-In inheritance is backwards compatible but not forwards compatible
- Dependency trees have backwards and forwards compatibility (though the forwards compatibility is with restrictions of what can be mapped)

Thus only projects that are intended to be consumed as either parent projects or as mix-in projects would deploy their newer `modelVersion` POM.

OPEN QUESTION: do we deploy the newer `modelVersion` POM as the `groupId:artifactId::version::pom` or as `groupId:artifactId::version:build:pom`? The first form ensures that the POM cannot be used as a parent by `modelVersion 4.0.0` projects as they will blow up immediately, however there has been an established practice of using `<packaging>pom</packaging>` for projects that produce non-standard artifacts and want to opt-out of the standard lifecycle binding, and thus we would break consumption of those "side" artifacts by legacy clients. Perhaps the solution is to follow the second form (i.e. it gets deployed with `<classifier>build</classifier>` and either put a Maven enforcer execution into the `modelVersion 4.0.0` POM or use the `<prerequisites>` tag to try and at least alert that the parent is invalid.

Open Questions

- Hervé Boutemy: Should we move away from the term Project for this document but use Product, as Software Product in Agile?
  Issue: a "project" is either a "Top Level Project" (ex: Maven), either a "mono-module project" (ex: shared-utils), either one "module of a multi-module project" (ex: maven-artifact module of Maven core)
  And even see Wikipedia "Project (disambiguation)" article: "Project, a temporary endeavor undertaken to create a unique product or service"
  A build file (pom.xml when used by Maven to build an artifact) or description of attributes of an artifact (pom.xml when in repository) is clearly not temporary: in agile methodology, this issue has been fixed by having a **product**, with a product manager.

Project Dependency Trees

`<project>` element

The project dependency trees model consists of a top level `<project>` tag and three types of immediate children elements:

```
<project modelVersion="..." groupId="..." artifactId="..." [platformId="..."] version="...">
  <generator .../>
  <information .../>
  <license .../>
  <artifacts .../>
</project>
```

The following are mandatory elements:

- `modelVersion` attribute - containing the model version of the project dependency trees, which can be used by consumers to select an XSLT tranformation to apply against the model if they need to translate a newer modelVersion to one that the consumer can parse.
- `groupId` attribute - containing the groupId of the project
- `artifactId` attribute - containing the artifactId of the project
- `version` attribute - containing the version of the project
- `artifacts` element - at least one element must be present, there are uniqueness constraints on this element relating to `platformId` attributes
- `generator` element - there must be exactly one `generator` element present. This element identifies the build tool that generated the Project Dependency Trees document.

The following are optional elements:

- `platformId` attribute - this is only present for additional atomic deployments of platform specific artifacts taking place after the initial deployment. When present there must be exactly one artifacts element and it must have the matching platformId as specified on the project element.
- `information` element - there can be at most one of these, it contains additional information about the project and its artifacts
- `license` elements - there can be any number of these, each element represents a set of licensing terms under which the project's artifacts are made available.

  In the vast majority of cases, projects are covered by a single set of license terms. Those cases will have a single `<license>` element that provides the SPDX expression for the license terms, e.g. `<license spdx="(LGPL-2.0 AND GPL-2.0)">` which would indicate that portions of the code are LGPL and other portions are GPL.

  In other cases, projects are dual or multi-licensed. Those cases will have multiple `<license>` elements where the consumer is free to select **any one** of those expressions as the license terms that they will be complying with, thus `<license spdx="GPL-2.0"/><license spdx="(BSD-2-Clause AND Apache-2.0)"/>` would indicate that the consumer **either** has to comply with the terms of the GPL **or** both BSD and Apache licenses. While SPDX syntax would allow for `<license spdx="(GPL-2.0 OR BSD-2-Clause AND Apache-2.0)"/>` as an equivalent expression, dual licensing seems important enough that it should be separated out explicitly in separate elements as the priority rules of SPDX expression syntax - while clear and unambiguous - can be confusing to the uninitiated with regard to `OR` being lower priority than `AND`.

`<generator>` element

The generator element identifies the build tool that created the document.

```
<generator name="..." version="..." url="..."/>
```

The following are mandatory elements

- `name` the human readable name of the build tool, e.g. `Apache Maven`, `Apache Buildr`, `Rake`, `Gradle`, etc.
- `version` the version of the build tool
- `url` the url of the home page of build tool, e.g. `http://maven.apache.org`, `http://buildr.apache.org`, `http://rake.rubyforge.org`, `http://gradle.org`, etc. This is not the download URL for the build tool

There are no optional elements

`<information>` element

The information element consists of optional information about the project and its artifacts.

TODO: decide what, if any, additional content can go in here, SCM, Issue trackers, URLs, Mailing Lists, etc.

```
<information>
  <name .../>
  <description .../>
</information>
```

There are no mandatory elements

The following are optional elements:

- `name` element - containing the name of the project (or when the `<information>` tag is scoped to a specific `<artifact>` overriding the project name for that specific artifact)
- `description` element - containing the description of the project (or when the `<information>` tag is scoped to a specific `<artifact>` the description of that specific artifact)

`<license>` element

The license element consists of information about one set of licensing terms that the project and its artifacts is made available under.

```
<license spdx="..."/>
```

There is one mandatory element

- `spdx` attribute - containing a SPDX expression for a single set of terms that the project and its artifacts (or when the `<license>` tag is scoped to a specific `<artifact>` overriding the project licenses for that specific artifact) are made available under. The SPDX expression may not contain `OR` operators. Instead the SPDX expression must be normalized to remove `OR` operators and instead present each disjoint set of licenses as a separate `<license>` tag.

There are no optional elements

`<artifacts>` element

The artifacts element consists of details of all the artifacts produced by the project. The artifacts are partitioned by platformId.

```
<artifacts [platformId="..."]>
  <artifact .../>
</artifacts>
```

There is one mandatory element:

- `artifact` element - there must be at least one `artifact` element within an `artifacts` element. There is no upper limit. Each `artifact` element corresponds to an artifact that has been deployed with the Project Dependency Trees

There is one optional element:

- `platformId` attribute - the presence of this attribute indicates that all the contained `artifact` elements are platform specific artifacts for the specified `platformId`. The absence of this attribute indicates that all the contained `artifact` elements are non-platform specific artifacts.

  NOTE: if an artifact supports a subset of all the platforms, we currently will envision that as non-platform specific as it targets more than one platform. An alternative solution may be to come up with a generic platform identifier that covers the multiple platforms or define a concatenation rule for `platformIds`, but addressing this concern otherwise is currently seen as premature optimization.

Uniqueness constraints apply to the `artifacts` element.

- If the enclosing `<project>` element has the `platformId` attribute specified then there must be one and only one `<artifacts>` element and that element must have the `platformId` attribute specified and the `platformId` must match exactly that of the `<project>` elements.

- If the enclosing `<project>` element has no `platformId` attribute then there may be at most one `<artifacts>` element without a `platformId` attribute and for any additional `<artifacts>` elements the `platformId` element must be unique.

`<artifact>` element

The artifact element consists of the details of a specific artifact produced by the project.

```
<artifact type="..." [classifier="..."]>
  <information .../>
  <license .../>
  <component .../>
  <provides .../>
  <requires .../>
  <supports .../>
</artifact>
```

There is one mandatory element:

- `type` attribute - this is the file type of the artifact. **NOTE:** this is the actual file extension that the artifact is deployed with, not the "packaging" type nor the "dependency" type which would require consumers to have awareness of all "packaging" to file type mappings.

The following are optional elements:

- `classifier` attribute - this is used to indicate side artifacts such as `sources` or `javadoc`, etc.
- `information` element - there can be at most one of these, this is used to override the `project` level information for this specific artifact. This tag effectively inherits from the `project` level tag.
- `license` elements - there can be any number of these. If there are no `license` elements then the license terms of this specific artifact are the license terms expressed in the `project` level tag. If there are any `license` elements in an `artifact` then only those `license` elements apply to the artifact.
- `component` elements - there can be any number of these. If present they are a *hint* to the consumer about `type` specific components that are present within the artifact and that may need to be considered during conflict resolution. For example, with a JAR artifact, the components may be Java 9+ module identifiers. Consumers may ignore the `component` elements if they choose.
- `provides` elements - there can be any number of these. If present they indicate that this artifact embeds equivalent content to the named dependency. The exact meaning of "embeds" is dependent on the type of artifact and the type of dependency.
- `requires` elements - there can be any number of these. If present they indicate that the consumer has a mandatory transitive dependency.
- `supports` element - there can be any number of these. If present they indicate that the consumer has an optional transitive dependency.

`<component>` element

The component element consists of hints to the consumer of type specific components that are present within the artifact for consideration during conflict resolution.

```
<component id="..."/>
```

There is one mandatory element:

- `id` attribute - this is an identifier, the conventions of how this identifier is used will be established by the tooling around the specific artifact types.

There are no optional elements:

One anticipated usage of the `component` element is for JAR artifacts that the `id` would correspond to the Java 9+ module identifiers as in Java 9+ the module identifier must be unique on the module path and hence conflict resolution will be required to process the dependency tree into a flattened modulepath with validation of uniqueness of component identifiers enforced.

`<provides>` element

The provides element indicates that this artifact embeds equivalent content to the named dependency

```
<provides groupId="..." artifactId="..." [platformId="..."] [version="..."] range="..." type="..."
[classifier="..."]/>
```

The following are mandatory elements

- `groupId` attribute - the groupId of the embedded dependency
- `artifactId` attribute - the artifactId of the embedded dependency
- `range` attribute - the version range of the embedded dependency - this will either be a hard range, e.g. `[1.0]` where the exact dependency has been explicitly embedded or a compatibility range, e.g. `[1.0,2.0)` where there is an "aliasing" or equivalence within an agreed API contract

- `type` attribute - this is the file type of the artifact. **NOTE:** this is the actual file extension that the artifact is deployed with, not the "packaging" type nor the "dependency" type which would require consumers to have awareness of all "packaging" to file type mappings.

The following are optional elements

- `platformId` attribute - the platformId of the embedded dependency
- `classifier` attribute - the classifier of the embedded dependency
- `version` attribute - the version of the embedded dependency. When present, the `range` attribute should probably be a hard range for this version only. When absent, this indicates that there has been an "aliasing" and as such the `range` attribute should reflect the compatibility constraints of the alias implementation.

Some examples may assist in the relative use-cases of the `range` and `version` attributes.

- `org.slf4j:log4j-over-slf4j` provides an alternative set of implementations of the `log4j:log4j::[1.0,2.0)` APIs. It does not actually include any content from any of the log4j jars. Rather the exact same public API contract has been re-implemented. Thus the `org.slf4j:log4j-over-slf4j` artifact might well state: `<provides groupId="log4j" artifactId="log4j" range="[1.0,2.0)" type="jar" />` there is no `version` attribute because the content has not been replicated
- `ch.qos.logback:logback-classic` is an SPI implementation for `org.slf4j:slf4j-api`. As `slf4j-api` requires that there is at most one SPI implementation on the classpath, it may be useful for all SPI implementations to declare `<provides groupId="org.slf4j" artifactId="slf4j-spi-impl" range="[1.7.0,1.8.0)" type="jar"/>` (this would also need slf4j-api to have a `<supports groupId="org.slf4j" artifactId="slf4j-spi-impl" range="[1.7.0,1.8.0)" type="jar"/>` tag to trigger conflict resolution. If slf4j-api did not have an internal fallback implementation then it would use a `<requires>` tag instead of a `<supports>` tag.
- überjars which basically aggregate multiple jar files would specify the `version` tag. Thus `org.hamcrest:hamcrest-all::1.3:jar` would have the `<provides groupId="org.hamcrest" artifactId="hamcrest-core" version="1.3" range="[1.3]" type="jar"/>` because it literally duplicates the exact content of `org.hamcrest:hamcrest-core::1.3`. The range needs to be a hard range as it has been embedded directly and is an intrinsic part of the artifact.

`<requires>` element

The requires element indicates a mandatory transitive dependency.

```
<requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..." type="..."
[classifier="..."] [modelVersion="..."]>
  <component .../>
  <license .../>
  <provides .../>
  <requires .../>
  <supports .../>
  <including>
    ...
  </including>
  <excluding>
    ...
  </excluding>
</requires>
```

The following are mandatory elements:

- `groupId` attribute - the groupId of the dependency
- `artifactId` attribute - the artifactId of the dependency
- `version` attribute - the version of the dependency that was resolved at build time and is the recommended default version of the dependency to use. The child elements of this `requires` element represent the information for the specified version of the dependency
- `range` attribute - the version range of the dependency. This may be the `modelVersion 4.0.0` style "unspecified here is a hint" style version, e.g. `1.0`. Preference would be that it uses the `version` attribute as a lower bound, but any valid Maven version range is acceptable.
- `type` attribute - this is the file type of the artifact. **NOTE:** this is the actual file extension that the artifact is deployed with, not the "packaging" type nor the "dependency" type which would require consumers to have awareness of all "packaging" to file type mappings.

The following are optional elements:

- `platformId` attribute - the platformId of the dependency
- `classifier` attribute - the classifier of the dependency
- `modelVersion` attribute - if the dependency's Project Dependency Tree uses a `modelVersion` less than or equal to the `modelVersion` of the root `project` tag in this Project Dependency Tree then this attribute must be omitted, otherwise the tag will contain the `modelVersion` of the dependency's Project Dependency Tree. The presence of this attribute is an indicator to the consumer that the contained elements were the result of an XSLT transformation of the dependency's tree and thus, if the consumer understands this newer `modelVersion` then a more correct view of the dependency tree could be obtained by fetching and parsing the dependency's Project Dependency Tree directly and substituting the parsed contents.
- `license` elements - there can be any number of these, they reflect the license terms of the dependency as detailed from the dependency's tree.
- `component` elements - there can be any number of these. If present they are a *hint* to the consumer about `type` specific components that are present within the dependency and that may need to be considered during conflict resolution. For example, with a JAR artifact, the components may be Java 9+ module identifiers. Consumers may ignore the `component` elements if they choose.
- `provides` elements - there can be any number of these. If present they indicate that this dependency embeds equivalent content to the named dependency. The exact meaning of "embeds" is dependent on the type of artifact and the type of dependency.
- `requires` elements - there can be any number of these. If present they indicate that the dependency has a mandatory transitive dependency.

- `supports` element - there can be any number of these. If present they indicate that the dependency has an optional transitive dependency.
- `including` element - there can be at most one of these. If present it indicates that the dependency has been augmented by its consumer to "correct" the dependency tree.
- `excluding` element - there can be at most one of these. If present it indicates that the dependency has been augmented by its consumer to "correct" the dependency tree.

`<supports>` element

The supports element indicates an optional transitive dependency.

```
<supports groupId="..." artifactId="..." [platformId="..."] [version="..."] range="..." type="..."
[classifier="..."]/>
```

The following are mandatory elements

- `groupId` attribute - the groupId of the dependency
- `artifactId` attribute - the artifactId of the dependency
- `range` attribute - the version range of the dependency. This may be the `modelVersion 4.0.0` style "unspecified here is a hint" style version, e. g. `1.0`. Preference would be that it uses the `version` attribute as a lower bound, but any valid Maven version range is acceptable.
- `type` attribute - this is the file type of the artifact. **NOTE:** this is the actual file extension that the artifact is deployed with, not the "packaging" type nor the "dependency" type which would require consumers to have awareness of all "packaging" to file type mappings.

The following are optional attributes

- `platformId` attribute - the platformId of the dependency
- `version` attribute - the version of the dependency that was resolved at build time and is the recommended default version of the dependency to use.
- `classifier` attribute - the classifier of the dependency

`<including>` element

The including element indicates that a dependency has been augmented by its immediate consumer.

```
<including>
  <component .../>
  <provides .../>
  <requires .../>
  <supports .../>
</including>
```

There are no mandatory elements

The following are optional elements:

- `component` elements - there can be any number of these. If present they are a *hint* to the consumer about `type` specific components that are present within the dependency and that may need to be considered during conflict resolution. For example, with a JAR artifact, the components may be Java 9+ module identifiers. Consumers may ignore the `component` elements if they choose.
- `provides` elements - there can be any number of these. If present they indicate that this dependency embeds equivalent content to the named dependency. The exact meaning of "embeds" is dependent on the type of artifact and the type of dependency.
- `requires` elements - there can be any number of these. If present they indicate that the dependency has a mandatory transitive dependency.
- `supports` element - there can be any number of these. If present they indicate that the dependency has an optional transitive dependency.

`<excluding>` element

The including element indicates that a dependency has been augmented by its immediate consumer.

```
<excluding>
  <component .../>
  <provides .../>
  <requires .../>
  <supports .../>
</excluding>
```

There are no mandatory elements

The following are optional elements:

- `component` elements - there can be any number of these. If present they indicate that this dependency and its transitive dependency tree - from the point of view of the immediate parent - should have the corresponding `<component>` elements expunged. The `id` can be either exact match or * style wildcard matches.
- `provides` elements - there can be any number of these. If present they indicate that this dependency and its transitive dependency tree - from the point of view of the immediate parent - should have the corresponding `<provides>` elements expunged. The `groupId`, `artifactId`, etc coordinates can be either exact matches or * style wildcard matches.
- `requires` elements - there can be any number of these. If present they indicate that this dependency and its transitive dependency tree - from the point of view of the immediate parent - should have the corresponding `<requires>` elements expunged. The `groupId`, `artifactId`, etc coordinates can be either exact matches or * style wildcard matches.
- `supports` element - there can be any number of these. If present they indicate that this dependency and its transitive dependency tree - from the point of view of the immediate parent - should have the corresponding `<supports>` elements expunged. The `groupId`, `artifactId`, etc coordinates can be either exact matches or * style wildcard matches.

Example

The following is a pseudo-example of a Project Dependency Tree

```
 <project modelVersion="..." groupId="..." artifactId="..." [platformId="..."] version="...">
    <generator name="Apache Maven" version="5.0.0" url="http://maven.apache.org"/>
    <information>
        <!-- container for descriptive information -->
        [<name>...</name>]
        [<description>...</description>]
        ...
    </information>
    <license spdx="..."/>
    <license spdx="..."/>
    ...
    <license spdx="..."/>
    <artifacts [platformId="..."]>
        <artifact type="..." [classifier="..."]>
            <information>
                <!-- optional element if need to override root level information for specific artifacts -->
            </information>
            <!--
              components are internal packaging constructs used by the packaging type but requiring more
general validation
              e.g. for Java 9+ the ids could be the module ids if we wanted to validate that the module ids
were unique in the
              effective tree.
            -->
            <component id="..."/>
            <component id="..."/>
            ...
            <component id="..."/>
            <!--
              If the artifact has a different set of licenses from those defined at the project level, we
define the licenses
              of this artifact here. Otherwise we defer to the licenses defined at the top level of the project.
              licensing is a top level concern, and legitimately can vary per artifact. Let's not solve license
compatibility,
              rather leverage https://spdx.org/
            -->
            <license spdx="..."/>
            <license spdx="..."/>
            ...
            <license spdx="..."/>
            <!--
              provides is a marker that we have duplication of content. This could be because we are much like
the many servlet-api jar
              files where there are many GAV's of the same javax.servlet:servlet-api:3.0 thus we could have the
case where

              org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec:jar:1.0.2.Final PROVIDES javax.servlet:
servlet-api:3.0
              org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec:jar:1.0.1.Final PROVIDES javax.servlet:
servlet-api:3.0
              org.jboss.spec.javax.servlet:jboss-servlet-api_3.0_spec:jar:1.0.0.Final PROVIDES javax.servlet:
servlet-api:3.0
              org.mortbay.jetty:servlet-api-3.0:jar:7.0.0pre2 PROVIDES javax.servlet:servlet-api:3.0
```

```
              similarly

              org.slf4j:log4j-over-slf4j:jar:1.7.21 PROVIDES log4j:log4j:[1.0,2)

              The consumer of the tree can then decide if/when/how to collapse redundant nodes as they see fit.

              TODO: decide optionality of version and range attributes
            -->
            <provides groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."] type="..."
[classifier="..."]>
                <!-- no elements here as we have "rebundled" hence implicitly promoted up one level-->
            </provides>
            <provides groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."] type="..."
[classifier="..."]/>
            ...
            <provides groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."] type="..."
[classifier="..."]/>
            <!--
              requires are the mandatory dependencies. This is effectively a recursive artifact where the GAV
is not inherited and
              where we have discarded the information section. If you want those details, fetch that project's
dependencies trees.
            -->
            <requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..." type="..."
[classifier="..."]>
                <component id="..."/>
                <license spdx:id="..."/>
                <provides groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."]
type="..." [classifier="..."]/>
                <requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..."
type="..." [classifier="..."]>
                    ...
                </requires>
                <supports groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."]
type="..." [classifier="..."]/>
            </requires>
            <requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..." type="..."
[classifier="..."]>
                ...
            </requires>
            ...
            <requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..." type="..."
[classifier="..."]>
                ...
            </requires>
            <!--
              supports are the optional dependencies. We list them here to aid in conflict resolution. We do
not include a nested tree
              as a consumer would only pull them in if the consumer already has its own a requires for them, so
we really only
              need to validate the range.

              TODO: decide optionality of range attribute
              TODO: decide if we want a version attribute
            -->
            <supports groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."] type="..."
[classifier="..."]/>
            <supports groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."] type="..."
[classifier="..."]/>
            <supports groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."] type="..."
[classifier="..."]/>
            <including>
                <component id="..."/>
                <provides groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."]
type="..." [classifier="..."]/>
                <requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..."
type="..." [classifier="..."]>
                    ...
                </requires>
                <supports groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."]
type="..." [classifier="..."]/>
```

```
            </including>
            <excluding>
                <component id="..."/>
                <provides groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."]
type="..." [classifier="..."]/>
                <requires groupId="..." artifactId="..." [platformId="..."] version="..." range="..."
type="..." [classifier="..."]/>
                <supports groupId="..." artifactId="..." [platformId="..."] version="..." [range="..."]
type="..." [classifier="..."]/>
            </excluding>
        </artifact>
        <artifact ...>
            ...
        </artifact>
        ...
        <artifact ...>
            ...
        </artifact>
    </artifacts>
    <!-- if the project does not specify a platformId then we can include additional platform details that were
part of the atomic deployment -->
    <artifacts platformId="...">
        ...
    </artifacts>
        ...
    <artifacts platformId="...">
        ...
    </artifacts>
</project>
```

Constructing a Project Dependency Trees model

The following process will be used to construct a Project Dependency Trees model:

1. For each artifact, construct the list of direct mandatory dependencies
2. For each artifact, construct the list of direct optional dependencies
3. For each artifact, construct the list of embedded dependencies
4. For each artifact, construct the list of licenses
5. For each artifact, construct the list of components
6. Construct the Project Dependency Trees of the embedded dependencies
7. Process the embedded dependency trees.
    a. Any `provides` elements should be appended to the list of embedded dependencies.
    b. Any `requires` elements should be appended to the list of direct mandatory dependencies.
    c. Any `supports` elements should be appended to the list of direct optional dependencies.
    d. Any `component` elements should be appended to the list of components
    e. Licensing will be assumed to have been correctly defined by the project when the decision was made to embed dependencies
8. Process the list of direct optional dependencies, removing any duplicates with the list of embedded dependencies (i.e. if in promoting an embedded dependency's supports tags we support it and have also embedded it, then we just have embedded it)
9. Process the list of direct mandatory dependencies, removing any duplicates with the list of embedded dependencies (i.e. if in promoting an embedded dependency's provides tags we provide it and have also required it, then we just have provided it)
10. Process the list of direct optional dependencies, removing any duplicates with the list of mandatory dependencies (i.e. if in promoting an embedded dependency's supports tags we support it and have also required it, then we just have required it)
11. Construct the Project Dependency Trees of the mandatory dependencies

The principle is that

- `provides` tags effectively ensure that the dependency is "promoted up".
- `requires` tags retain the tree nesting
- `supports` tags do not detail any transitive dependencies as this only comes into play if the consumer already has a `requires` dependency on the same coordinates and needs to perform conflict resolution.

If a dependency is missing a Project Dependency Trees model, then the same process can be used to construct that model from the `modelVersion 4.0.0` POM

TODO define the process for constructing the effective `modelVersion 4.0.0` POM

TODO decide if we should include some "well known" conventions, e.g. that:

- the `javadoc` and `source` jar files probably do not have any dependencies
- the `test-jar` probably has the test scope dependencies
- that `war` artifacts probably have <`provides`> tags for all `jar` dependencies
- etc

The above would probably be generally useful but would complicate the specification of the process for other consumers

Test data set

TODO: We need a cohort of Project Dependency Tree documents for implementations to validate their parsers against and to validate their convertion into `modelVersion 4.0.0` POMs (though we may do this using a provided XSLT file, implementations will still need to validate that their have configured their XSLT engine correctly)

TODO: We need a cohort of `modelVersion 4.0.0` POMs for implementations to validate their generation of effective Project Dependency Trees in the absence of a deployed Project Dependency Trees document along with the expected effective Project Dependency Tree documents

TODO: We need a cohort of sample transformations of trees to ensure that implementations can correctly aggregate Project Dependency Trees when building new projects that depend on other projects.

Schema

Here is a draft XML schema:

TODO: Add the `including` and `excluding` elements to the schema

```
 <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001
/XMLSchema" >
  <xsd:simpleType name=”coordinate”>
    <xsd:restriction base=”xsd:string”>
      <!-- TODO add pattern for groupId/artifactId/platformId/version/type/classifier valid values -->
    </xsd:restriction>
  </xsd:simpleType>
  <xs:element name="project">
    <xs:annotation>
      <xs:documentation source="version">5.0.0+</xs:documentation>
      <xs:documentation source="description">
        The <code>&lt;project&gt;</code> element is the root of the project
        dependency trees.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="modelVersion" type="xs:string"/>
      <xs:attribute name="groupId" type="coordinate"/>
      <xs:attribute name="artifactId" type="coordinate"/>
      <xs:attribute name="version" type="coordinate"/>
      <xs:attribute name="platformId" type="coordinate" use="optional"/>
      <xs:all>
        <xs:element ref="generator" minOccurs="1" maxOccurs="1"/>
        <xs:element ref="information" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="license" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="artifacts" minOccurs="1" maxOccurs="unbounded"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="generator">
    <xs:annotation>
      <xs:documentation source="version">5.0.0+</xs:documentation>
      <xs:documentation source="description">
        The <code>&lt;generator&gt;</code> element identifies the build tool
        responsible for creating this document
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="version" type="xs:string"/>
      <xs:attribute name="url" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="information">
    <xs:annotation>
      <xs:documentation source="version">5.0.0+</xs:documentation>
      <xs:documentation source="description">
        The <code>&lt;information&gt;</code> element is a container for
        descriptive information about either all the artifacts in a project or
```

```
      a specific artifact.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:all>
      <xs:element name="name" type="xs:string" maxOccurs="1"/>
      <xs:element name="description" type="xs:string" maxOccurs="1"/>
      <!-- TODO add additional elements -->
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="license" xmlns:spdx="http://spdx.org/rdf/terms#">
  <xs:annotation>
    <xs:documentation source="version">5.0.0+</xs:documentation>
    <xs:documentation source="description">
      The <code>&lt;license&gt;</code> element defines one of the licenses
      under which the artifacts are made available. Where a license is
      attached to the <code>&lt;project&gt;</code> element this defines the
      default licenses for all artifacts in the project. Where a license is
      attached to an <code>&lt;artifact&gt;</code> element this signifies
      that the specific artifact is covered by the
      <code>&lt;license&gt;</code> elements defined within that
      <code>&lt;artifact&gt;</code> element. Licenses are identified using
      the <a href="http://spdx.org">SPDX</a> identifiers
   </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="spdx" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="artifacts">
  <xs:annotation>
    <xs:documentation source="version">5.0.0+</xs:documentation>
    <xs:documentation source="description">
      The <code>&lt;artifacts&gt;</code> element is a container for
      details of artifacts. When the <code>&lt;artifacts&gt;</code> attribute
      is missing, then the artifacts listed are not platform specific.
      The <code>&lt;artifacts&gt;</code> must be unique with respect to their
      <code>&lt;platformId&gt;</code>, i.e. it cannot be repeated.
      If the <code>&lt;project&gt;</code> element has a
      <code>&lt;platformId&gt;</code> then there must be only one
      <code>&lt;artifacts&gt;</code> element and it must have the matching
      <code>&lt;platformId&gt;</code>.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="platformId" type="coordinate" use="optional"/>
    <xs:sequence>
      <xs:element ref="artifact" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="artifact">
  <xs:annotation>
    <xs:documentation source="version">5.0.0+</xs:documentation>
    <xs:documentation source="description">
      The <code>&lt;artifact&gt;</code> element represents an artifact
      associated with the project.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="type" type="coordinate"/>
    <xs:attribute name="classifier" type="coordinate" use="optional"/>
    <xs:all>
      <xs:element ref="information" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="license" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="component" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="provides" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="requires" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="supports" minOccurs="0" maxOccurs="unbounded"/>
    </xs:all>
```

```
  </xs:complexType>
</xs:element>
<xs:element name="component">
  <xs:annotation>
    <xs:documentation source="version">5.0.0+</xs:documentation>
    <xs:documentation source="description">
      The <code>&lt;component&gt;</code> element represents a type specific
      component that is present within the artifact. For example a "jar"
      artifact might list the Java 9+ modules that are included within
      the "jar". Other file types can use the component according to the
      conventions of that file type. The component information is intended
      to assist build time tools in conflict detection when resolving
      the composite dependency tree according to the build tools
      dependency resolution strategy.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
</xs:element>
<xs:element name="provides">
  <xs:annotation>
    <xs:documentation source="version">5.0.0+</xs:documentation>
    <xs:documentation source="description">
      The <code>&lt;provides&gt;</code> element represents a semantic
      equivalence with another artifact. There are several ways the element
      can be used.
      <nl>
        <li>
          When an artifact directly includes the same content as another
          project's artifacts, for example there are some "jar" files that
          will embed other artifacts to produce a so-called "uber-jar".
        </li>
        <li>
          When an artifact re-implements the API of another project's
          artifact. For example: log4j-over-slf4j reimplements the log4j
          API.
        </li>
        <li>
          When a set of projects are co-operating to provide multiple
          implementations of a "virtual" project artifact. For example:
          slf4j-log4j, slf4j-jul, and logback could all be considered
          as providing a slf4j-impl virtual project artifact. There would
          be no actual project at the slf4j-impl coordinates, but
          slf4j-api could declare a requirement on the "virtual" project
          artifact in order to ensure that an implementation is available
          to consumers of the API
        </li>
      </nl>
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:attribute name="groupId" type="coordinate"/>
    <xs:attribute name="artifactId" type="coordinate"/>
    <xs:attribute name="platformId" type="coordinate" use="optional"/>
    <xs:attribute name="version" type="coordinate"/>
    <xs:attribute name="range" type="xs:string"/>
    <xs:attribute name="type" type="coordinate"/>
    <xs:attribute name="classifier" type="coordinate" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="requires">
  <xs:annotation>
    <xs:documentation source="version">5.0.0+</xs:documentation>
    <xs:documentation source="description">
      The <code>&lt;requires&gt;</code> element represents a hard dependency
      on another project's artifact. If the <code>&lt;version&gt;</code>
      attribute is missing then this indicates that the dependency is
      a virtual dependency, and there must be no child elements.
      The <code>&lt;modelVersion&gt;</code> attribute must only be present
      if the dependent project's <code>&lt;modelVersion&gt;</code> is newer
```

```
        than the <code>&lt;modelVersion&gt;</code> specified on the root
        <code>&lt;project&gt;</code> element. The presence of this element
        indicates that the child information was the result of an XSLT
        transformation of a newer <code>&lt;modelVersion&gt;</code> and
        indicates that a build tool understanding the newer
        <code>&lt;modelVersion&gt;</code> may want to fetch the dependencies
        tree and process it directly in order to obtain the most correct
        model of the dependency.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="groupId" type="coordinate"/>
      <xs:attribute name="artifactId" type="coordinate"/>
      <xs:attribute name="platformId" type="coordinate" use="optional"/>
      <xs:attribute name="version" type="coordinate" use="optional"/>
      <xs:attribute name="range" type="xs:string"/>
      <xs:attribute name="type" type="coordinate"/>
      <xs:attribute name="classifier" type="coordinate" use="optional"/>
      <xs:attribute name="modelVersion" type="xs:string" use="optional"/>
      <xs:all>
        <xs:element ref="license" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="component" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="provides" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="requires" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="supports" minOccurs="0" maxOccurs="unbounded"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="supports">
    <xs:annotation>
      <xs:documentation source="version">5.0.0+</xs:documentation>
      <xs:documentation source="description">
        The <code>&lt;supports&gt;</code> element represents a soft dependency
        on another project's artifact. This element is provided in order to
        allow build time tools to perform conflict resolution when determining
        the effective tree from multiple dependencies.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="groupId" type="coordinate"/>
      <xs:attribute name="artifactId" type="coordinate"/>
      <xs:attribute name="platformId" type="coordinate" use="optional"/>
      <xs:attribute name="version" type="coordinate"/>
      <xs:attribute name="range" type="xs:string"/>
      <xs:attribute name="type" type="coordinate"/>
      <xs:attribute name="classifier" type="coordinate" use="optional"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```