Fragma 2.0 User's Guide

A dependency management and release automation system
for heterogeneous development environments

Table of Contents

1

About

fragma was created by Ovidiu Feodorov <ovidiu@novaordis.com>, Nova Ordis LLC

The manual was written by Ovidiu Feodorov and Joel VanderKwaak (3DGeo Inc.)

This manual was updated for fragma release 2.0.CR2

Published on Jan 08, 2008

2

Introduction

Fragma is a dependency management and release automation system. It was designed to function in a heterogeneous development environment and faciliate integration of artifacts produced using various programming languages, such as Fortra, C, Python, Perl, Java and shell scripting.

Testing and Release Environment (fragma)'s goals are to:

- Introduce *formal dependency management*
- Insure consistent artifact distribution and sharing, via a versioned *artifact repository*
- Introduce an *unified build infrastructure*
- Automate and speed up the release process (goal to be attained by the fragma 3 release)
- Introduce a unified testing framework and continous integration (goal to be attained by the fragma 3 release)

3

Concepts

Fragma is a dependency management and release automation system designed to interconnect with and augment standard software development tools such as GNU Make, Apache ANT, Subversion, Bugzilla or JIRA. Fragma's ultimate goal is trivialize release cycle overhead by taking over all dull repetitive work associated with the technicalities of a release, letting developers concentrate on just two fundamental activities: developing functionality and associated tests. Everything else should be automated and easy, and Fragma aims to fulfill this goal.

If a software development project uses third party dependencies, produces versioned artifacts, stores its source code and configuration in a revision control system, tests its artifacts on a continuous integration server, has formal releases managed with the help of an issue system such as JIRA or Bugzilla, then Fragma can be used "glue" between these components.

Fragma is written entirely in Java, but it was designed to be a generic tool that can be used with software projects based on Fortran, C, Python, shell scripts and pretty much any other programming language and development environment. In order to address the specific needs of projects that produce platform-dependent artifacts, Fragma has the concept of "artifact attribute", which can be the platform the artifact was compiled for, the compiler used in compilation, even specific compiler flags. Or, Fragma can disregard artifact attributes altogether, and handle platform-independent Java jars or Adobe PDF documents, or even datasets that ship as a seismic processing tool example.

3.1. Dependency Management

One of the most important Fragma feature is that it manages dependencies. It is a known fact that a lot of functionality needed by an application, any application, "floats around" in form of already available libraries and components, open-source or not. The most obvious example is logging for Java project: there are at least several logging libraries out there, shared under all sorts of licenses, all available for integration with your project.

Fragma provides a formal mechanism a project can use to declare its dependencies. Once declared, Fragma will insures that dependencies are available where and when the project needs them. All background work implied by actually transferring dependencies over the network, from a repository, insuring their consistency and installing them in a location accessible to the project is transparently handled by the tool.

A project declares its dependencies as metdata, in an XML file. XML format was the first obvious choice, for various reasons, but nothing precludes Fragma from extracting the metadata from other media (text files, database, etc.). An important element regarding the dependency metadata is that is part of your *project state*, hence it must be handled the same way your source code is, stored and tagged within the same version control system.

The dependencies are uniquely identified by an identifier that must be unique in the context of an artifact repository, and a version. Once declared, Fragma automates the process of "fetching" the dependencies from a remote repository and storing them in a file-bases cache, local to the project.

3.1.1. Transitive Dependency Resolution

Dependency artifacts are themselves the end result of software development projects, and those projects can have in turn their own dependencies, which are needed in order for the "first level dependencies" to function. Project usually end up declaring trees of dependencies. This is a very common situation, and it is usually dealt with by developers by manually "flattening" the dependency tree and picking the "right" dependency versions. If one of the "root" dependency changes, or even just its version changes, by upgrading, for example, the manual resolution process has to be repeated, files manually updated, and so on.

Fragma automates this process. A project only needs to declare its root dependencies, fragma transitively walks the dependency tree and identified the full transitive closure of required dependencies. Fragma is also selective in what is needed to be transferred, it only transfers the minimal transitive closure of the dependency tree.

3.1.2. Version Conflict Detection

Since dependency management is ultimately "flattening of the dependency tree", which results in a dependency list, conflicts may arise in the process. For example, if our project depends on `A version 1.0` and `B version 1.0`, and in turn A depends on `B version 2.0`, a version conflict arises. We cannot have at the same time `B version 1.0` and `B version 2.0` in the dependency list and a decision as to what to keep needs to be taken. Fragma signals this kind of conditions and it may assist in solving the conflict, though fully automatic conflict resolution is not possible, because dependency ultimately means functionality, and it is up to developers to decide on functionality.

3.2. Integration with a Revision Control System

A revision control system that maintains versioned copies of source code files and assists with integration of work in a distributed development team is essential for a large scale development project. The revision control system can be seen as the keeper of versioned snapshots of the code base, and accessing past snapshots is essential to rebuild the exact version of a project that has been shipped.

The key to the process of recreating already released artifacts is the *revision control tag*. During the artifact generation and release process, the revision control tag of the source snapshot that the artifact is build based on is embedded in the artifact itself. This way, the revision control tag can be always retrieved from the artifact itself and can be used to check out the exact snapshot from the repository. Fragma acts as a bridge between the revision control system and the build mechanism, extracting this information and passing it along. Also, since it manages project's version and dependency metadata, Fragma is in the best position to pass this information along to the build system so the build system can embed it into the artifacts.

3.3. Artifact Publishing and Release Automation

A project's artifacts can ultimately become a dependency of other projects, and Fragma consistently uses the same metadata language to expose project identification and versioning information: each Fragma-enabled project declares its own identifier and version in the same central metadata file, and Fragma is the first choice when you need to "publish" the artifacts of your project. In a way, publishing is the "inverse" operation of fetching. The artifacts can go into the same artifact repository the dependencies were fetched from, which is the most typical situation, or some other repository instance.

The artifact publishing process is just a step of an unified release process, which involves building, testing, tagging, publishing, advertising the releases and changes, and Fragma can eliminate the need for almost any manual intervention in this process.

3.4. Interaction with an Issue Tracking System

Fragma could potentially act as a mediator between an issue tracking system such as JIRA or Bugzilla and the release process, allowing for fully automated releases. For example, the JIRA "version" could be automatically "released" when the real release occurs, release notes could be automatically generated and e-mailed to interested users, etc. This functionality will be supported in a future release.

4

Installation

4.1. Prerequisites

In order to install and use Fragma, Java 5 or higher must be installed and available on the system.

The location of the Java installation must be reflected by the `JAVA_HOME` environment variable. This variable must be defined in the environment of the users who intend to run fragma. fragma runtime will look up `JAVA_HOME` to locate `$JAVA_HOME/bin/java`, which is needed for execution.

4.2. Shared Installation

4.2.1. Shared installation based on the value of `FRAGMA_HOME` (recommended)

1. **Unzip the Release Bundle**

   Unzip `fragma-2.0.CR2.zip` into a shared directory, for example `/usr/local`. It is important to insure that all users that are supposed to have access to fragma can access and have read permissions on the shared directory.

   Unziping the installation bundle will create a `fragma-2.0.CR2` directory that contains everything fragma requires to run. This manual is available under the `doc` subdirectory of the installation.

2. **Create a Generic Symbolic Directory Link**

   In order to facilitate further upgrades, it is good practice to create a generic symbolic directory link called `fragma`, pointing to the installation directory `fragma-2.0.CR2`. Following this convention, and recreating the link after each upgrade, the latest fragma version will always be available under `/usr/local/fragma`, which simplifies administration.

```
cd /usr/local
ln -s fragma-2.0.CR2 fragma
```

3. **Publish the Installation Location as `FRAGMA_HOME`**

   Make sure that an evironment variable named `FRAGMA_HOME` is visible to every user that is supposed to access fragma. The `FRAGMA_HOME` environment variable should point to the generic installation directory (`/usr/local/fragma` in this case).

4. **Create a Link to the Executable from a Directory already in PATH**

   In order to prevent the PATH value from growing overly large, it's a good practice to create a link to the fragma executable from a directory already available in PATH.

   We can rely on the fact that the latest fragma executable is already available under /usr/local/fragma/bin, so the symbolic link to the executable should point to this location. Provided that /usr/local/bin or /usr/bin is already in your PATH, you should:

```
cd /usr/bin
ln -s /usr/local/fragma/bin/fragma fragma
```

5. **Installation Validation**

   At this point, interactive help can be accessed by executing the following command from an arbitrary location:

```
fragma help
```

   To insure that you're actually running the intended fragma version, run:

```
fragma version
```

4.2.2. Shared installation based on the value of PATH

1. **Unzip the Release Bundle**

   Unzip fragma-2.0.CR2.zip into a shared directory, for example /usr/local. It is important to insure that all users that are supposed to have access to fragma can access and have read permissions on the shared directory.

   Unziping the installation bundle will create a fragma-2.0.CR2 directory that contains everything fragma requires to run. This manual is available under the doc subdirectory of the installation.

2. **Create a Generic Symbolic Directory Link**

   In order to facilitate further upgrades, it is good practice to create a generic symbolic directory link called fragma, pointing to the installation directory fragma-2.0.CR2. Following this convention, and recreating the link after each upgrade, the latest fragma version will always be available under /usr/local/fragma, which simplifies administration.

```
cd /usr/local
ln -s fragma-2.0.CR2 fragma
```

3. **Add Installation Location to PATH**

Modify the `PATH` environment variable for every user who is supposed to access fragma, so it includes `/usr/local/fragma/bin`.

4. **Installation Validation**

   At this point, interactive help can be accessed by executing the following command from an arbitrary location:

```
fragma help
```

   To insure that you're actually running the intended fragma version, run:

```
fragma version
```

4.3. Private Installation

For a private installation (useful when experimenting with new fragma versions, for example), repeat any of the procedures above in a private, non-shared, directory and make sure that the private fragma `bin` directory comes first in your `PATH`, or your `FRAGMA_HOME` reflects the location of the private installation.

5

A Java "Hello World" Example

This section contains a step-by-step guide to configure a simple Java project to use Fragma. The Java project uses ant as build tool, must have access to an artifact repository, which it uses as a dependency source as well as a publishing outlet.

This section is still work in progress.

6

A C "Hello World" Example

PythonUtils `project.xml` file example:

```
<project>

    <id>PythonUtils</id>

    <name>Collection of mpi utils for moving data around a cluster</name>

    <version>1.0.GA</version>

    <repository>
        <download url="http://delta/repository"/>
        <upload url="ftp://delta" username="fragma" password="fragma"/>
    </repository>

    <dependencies>

        <!--
            precompiled libraries + headers
        -->
        <project id="SEPlib" version="6.4.4" prefix="lib"
                 architecture="current"
                 platform="current"
                 artifacts="SEPlib.tar.gz"/>

        <project id="mpich" version="1.2.7" prefix="lib"
                 architecture="current"
                 platform="current"
                 artifacts="mpich.tar.gz"/>

        <!--
             make-based buildsystem
        -->
        <project id="buildsystem" version="1.0.Beta6" prefix="3dgeo"
                 artifacts="buildsystem.tar.gz"/>

        <!--
             common internal libraries
        -->
        <project id="libMVA" version="1.0.Alpha" prefix="3dgeo"
                 artifacts="libMVA.tar.gz"/>
        <project id="PickLib" version="1.0.ALPHA" prefix="3dgeo"
                 artifacts="PickLib.tar.gz"/>
        <project id="tdg_lib" version="1.0.GA" prefix="3dgeo"
                 artifacts="tdg_lib.tar.gz"/>
        <project id="licensing" version="1.0.CR1" prefix="3dgeo"
                 artifacts="licensing.tar.gz"/>

    </dependencies>

    <artifacts>
```

```
            <!--
                A directory declared as artifact is automatically tar.gzipped.
            -->
            <artifact name="PythonUtils" local-path="./output"/>

        </artifacts>

    </project>
```

This section is still work in progress.

7

The Project Metadata File

The project metadata file (`project.xml`) is located in a project's home directory, on the same level as Ant's `build.xml` or GNU make `makefile`. fragma will always try to load the project metadata from the current directory, and will fail loudly if it doesn't find it.

It is possible to configure Fragma to look up the `project.xml` file from a different directory. For that, use `-basedir` option, as shown in the following example:

```
$ cd /home/ovidiu
$ fragma -basedir /home/ovidiu/example info
```

The project file contains several major logical sections, but the physical location of the XML tags doesn't matter, they can be specified in any order.

7.1. Project Identification Section

The project identification section contains the metadata that uniquely identifies the project (component) in the context of an artifact repository. The section contains:

Project ID
: Is an alphanumeric string, without spaces, which must be unique per artifact repository. It is common practice to use the JIRA identifier of the project as project ID. Arbitrary values are allowed, but using the JIRA identifier enables JIRA integration and automatic releases.

Project Name
: Is a "human readable" name of the project, appropriate to be displayed in the "About" box of a GUI, or similar. The project name is free format and can include spaces.

Project Version
: The project version information in restricted Dewey format (`major.minor.update.qualifier`). The "major", "minor" and "update" components are numeric. "minor", "update" and "qualifier" are optional. The qualifier is usually a common release qualifier such as "Alpha", "Beta", "CR","GA" or "SP", but it can be any string.

Project Prefix
: Is the relative path from the root of the artifact repository to the location of the project in the repository. The prefix is optional. If not specified, the implied semantics is that the project lives in the artifact repository root.

An example showing how to use these configuration elements is available below:

```
<project>

        <id>TREX</id>
        <name>A Fragma example project</name>
        <version>1.0</version>
        <prefix>examples</prefix>

        ....

</project>
```

7.2. Dependency Section

This is where the project formally declares its dependencies. The project *only needs to declare the roots of its dependency trees*. For example, if our project depends on A, and A depends on B, we only need to declare the dependency on A. Fragma will automatically walk the dependency tree and fetch all "implied" dependencies.

A dependency declaration contains, at minimum

- The dependency project ID
- The dependency version

as illustrated in the following example:

```
<project>

        ....

        <dependencies>
            <project id="log4j" version="1.2.8"/>
        </dependencies>

        ....

</project>
```

In the case of the minimal declaration shown in the example above, Fragma will look up the "log4j" dependency in the artifact repository, and if second-level dependencies are detected in the repository, will walk the dependency tree and will fetch all artifacts declared by those dependencies. This is a recursive process, and Fragma can walk arbitrary depth trees. More about repository metadata files is provided in "The Artifact Repository" section below.

Multiple dependency roots are of course allowed:

```
<project>
```

```
                ....

                <dependencies>
                     <project id="log4j" version="1.2.8"/>
                     <project id="junit" version="3.4"/>
                </dependencies>

                ....

        </project>
```

If a project exposes multiple artifacts, and you only need a sub-set of those artifacts with your project, the name of those artifacts can be specified as a comma-separated value of the `artifacts` attribute:

```
        <project>

                ....

                <dependencies>
                     <project id="log4j" version="1.2.8"
                              artifacts="log4j.jar, log4j-src.jar"/>
                </dependencies>

                ....

        </project>
```

7.2.1. Artifact Attributes

A Java project dependency management is a lot simpler than the similar process for a project that uses native dependencies, because Java artifacts are platform independent. The artifact name is sufficient to uniquely identify it, provided we know the project id and the version.

For the native artifacts world, details such as architecture, platform, threading support, compiler, etc. suddenly become relevant. Artifacts compiled for different platforms may be, and most likely are, incompatible. By *artifact attribute* we understand an artifact characteristic that differentiates it from equivalent instances that essentially provide the same functionality, and possibly, makes it incompatible with others. For example, the *architecture* for which a specific executable has been compiled is an attribute of the executable artifact. An executable compiled for a Linux platform won't run on a Windows platform and vice-versa, even if the code base used for compilation is the same, so the functionality it offers is essentially the same.

One of Fragma's functions is to stage pre-processed artifacts and expose them to interested projects so naturally Fragma must be able to discern requests that were made for artifact with specific attributes. The project that makes the request must be able to express that it wants a specific attribute or combination of attributes for a specific artifact. The requesting project must be also able to recourse to defaults, if it chooses so.

The attribute specification and management mechanism is currently sufficiently flexible to allow adding new "relevant" attributes in the future, without breaking the backward compatibility and rendering deployed project metadata unusable.

Supported Artifact Attributes

7.2.1.1. Architecture

The `architecture` attribute represents the system architecture the artifact is intended to run on. Legal values for this attribute are given by the output of '`uname -s`' UNIX command.

Example: `Linux`

To request an artifact built for a specific architecture, provided it is available in the repository, use a declaration similar to:

```
<project>

   ....

   <dependencies>
       <project id="EXAMPLE" version="1.0" architecture="Linux" .../>
   </dependencies>

   ....

</project>
```

To indicate than an artifact was built for a specific architecture and to propagate this information to repository when publishing the artifact, use a declaration similar to:

```
<project>

   ....

   <artifacts>
       <artifact name="libutil.a" architecture="Linux" .../>
   </artifacts>

   ....

</project>
```

More details about artifact publishing are available in the "Artifact" section, below.

There are cases when architecture is irrelevant for specific artifacts, such as Java jar archives, compilable source code bundles, or Adobe PDF document files. In these situations, the *absence* of the `architecture` attribute explicitly specifies this fact. If the attribute is missing, Fragma assumes that the architecture does not matter and it will try to locate and provide an architecture-independent artifact, if available.

When specifying the architecture in a `project.xml` file, '`current`' is a supported value, meaning "the architecture of the system on which Fragma client is currently running". In this case, Fragma will attempt to locate and download an artifact with consistent attributes.

7.2.1.2. Platform

The `platform` attribute represents the machine hardware the artifact is intended to run on. Legal values for this attribute are given by the output of '`uname -m`' UNIX command.

Example: `i686`

To request an artifact built for a specific platform, provided it is available in the repository, use a declaration similar to:

```
<project>

   ....

   <dependencies>
       <project id="EXAMPLE" version="1.0" ... platform="x86_64" .../>
   </dependencies>

   ....

</project>
```

To indicate than an artifact was built for a specific architecture and to propagate this information to repository when publishing the artifact, use a declaration similar to:

```
<project>

   ....

   <artifacts>
       <artifact name="libutil.a" ... platform="x86_64" .../>
   </artifacts>

   ....

</project>
```

More details about artifact publishing are available in the "Artifact" section, below.

In the situation the platform is irrelevant for a specific artifact, the absence of `platform` attribute explicitly expresses that. When specifying the platform in a `project.xml` file, '`current`' is a supported value, meaning "the platform of the system on which Fragma client is currently running". In this case, Fragma will attempt to locate and download an artifact with consistent attributes.

7.2.1.3. Compiler

The `compiler` attribute represents the compiler used to build the artifact. Can be specified using the `compiler` attribute. The attribute value is a free-format string specifying the compiler type and possibly version.

Example: `gcc`, `gcc-4.2.2`

To request an artifact built with a specific compiler, provided it is available in the repository, use a declaration similar to:

```
<project>

   ....

   <dependencies>
       <project id="EXAMPLE" version="1.0" ... compiler="gcc-4.2.2" .../>
   </dependencies>

   ....

</project>
```

To indicate than an artifact was built with a specific compiler and to propagate this information to repository when publishing the artifact, use a declaration similar to:

```
<project>

   ....

   <artifacts>
       <artifact name="libutil.a" ... compiler="gcc-4.2.2" .../>
   </artifacts>

   ....

</project>
```

More details about artifact publishing are available in the "Artifact" section, below.

If a compiler is specifically requested, and no corresponding artifact is available, Fragma will report an error. If no compiler is specifically requested, the default artifact for the corresponding architecture/platform will be provided.

7.2.1.4. Threading Support

The fact that the artifact has native threading support can be specified using the `threading` attribute. Legal values are '`yes`' or '`no`'.

To request an artifact that has threading support, provided it is available in the repository, use a declaration similar to:

```
<project>

   ....
```

```
    <dependencies>
        <project id="EXAMPLE" version="1.0" ... threading="yes" .../>
    </dependencies>

    ....

</project>
```

To indicate than an artifact was built with threading support and to propagate this information to repository when publishing the artifact, use a declaration similar to:

```
<project>

    ....

    <artifacts>
        <artifact name="libutil.a" ... threading="yes" .../>
    </artifacts>

    ....

</project>
```

More details about artifact publishing are available in the "Artifact" section, below.

If a threading type is specifically requested, and no corresponding artifact is available, Fragma will report an error. If no threading type is specifically requested, the default artifact for the corresponding architecture/platform will be provided.

7.2.1.5. `glibc` Version

The version of the glibc library the artifact was compiled against can be specified using the `glibc` attribute. Legal values are glibc versions.

To request an artifact that was built for a specific glic version, provided it is available in the repository, use a declaration similar to:

```
<project>

    ....

    <dependencies>
        <project id="EXAMPLE" version="1.0" ... glibc="2.3.3" .../>
    </dependencies>

    ....

</project>
```

To indicate than an artifact was built for a specific glibc version and to propagate this information to repository when publishing the artifact, use a declaration similar to:

```
<project>

   ....

   <artifacts>
       <artifact name="libutil.a" ... glibc="2.3.3" .../>
   </artifacts>

   ....

</project>
```

More details about artifact publishing are available in the "Artifact" section, below.

If a glibc version is specifically requested, and no corresponding artifact is available, Fragma will report an error. If no glibc type is specifically requested, the default artifact for the corresponding architecture/platform will be provided.

7.2.1.6. Licensing Support

Normally, licensing support should be added at linking time, and only for end-user artifacts, such as product artifacts. In such cases, "licensed" artifacts can be requested by using the `licensing` attribute. Legal values are '`yes`' or '`no`'.

To request an artifact that has licensing support, provided it is available in the repository, use a declaration similar to:

```
<project>

   ....

   <dependencies>
       <project id="EXAMPLE" version="1.0" ... licensing="yes" .../>
   </dependencies>

   ....

</project>
```

To indicate than an artifact was built with licensing support and to propagate this information to repository when publishing the artifact, use a declaration similar to:

```
<project>

   ....
```

```
    <artifacts>
        <artifact name="libutil.a" ... licensing="yes" .../>
    </artifacts>

    ....

</project>
```

More details about artifact publishing are available in the "Artifact" section, below.

If the licensing attribute is specifically set to "yes", and no corresponding "licensed" artifact is available, Fragma will report an error. If no licensing is specifically requested, the default *unlicensed* artifact for the corresponding architecture/platform will be provided.

7.2.1.7. Compilation Flags

7.2.1.8. Generic Attributes

The attribute management mechanism is currently sufficiently flexible to allow using new "unknown" attributes.

7.2.2. Declarative Support for Artifact Attributes

The generic syntax for declaring a dependency that requires specific attributes is:

```
    <project>

            ....

            <dependencies>
                 <project id="EXAMPLE" version="1.0"

                           attribute_name_1="attribute_value_1"
                           attribute_name_2="attribute_value_2"
                           ...
                           attribute_name_n="attribute_value_n"

                           artifacts="artifact1, artifact2, ... artifactn"/>
            </dependencies>

            ....

    </project>
```

The following equivalent alternatives are also supported:

```
    <project>

            ....
```

```
        <dependencies>
             <project id="EXAMPLE" version="1.0"
                       artifacts="artifact1, artifact2, ... artifactn"/>

                    <attribute name="attribute_name_1"
                               value="attribute_value_1"/>
                    <attribute name="attribute_name_2"
                               value="attribute_value_2"/>
                    ...
                    <attribute name="attribute_name_n"
                               value="attribute_value_n"/>
             </project>
        </dependencies>

        ....

</project>
```

```
<project>

        ....

        <dependencies>
             <project id="EXAMPLE" version="1.0"/>

                    <artifact>artifact1</artifact>
                    <artifact>artifact2</artifact>
                    ...
                    <artifact>artifactn</artifact>

                    <attribute name="attribute_name_1"
                               value="attribute_value_1"/>
                    <attribute name="attribute_name_2"
                               value="attribute_value_2"/>
                    ...
                    <attribute name="attribute_name_n"
                               value="attribute_value_n"/>
             </project>
        </dependencies>

        ....

</project>
```

7.2.3. Attribute-less artifacts

Fragma supports attribute-less artifacts, for which attribute such architecture or platform are irrelevant. This is the case for Java jar files. A jar dependency can be specified as:

```
<project>

        ....

        <dependencies>
```

```
                    <project id="log4j" version="1.2.8"
                             artifacts="log4j.jar, log4j-src.jar"/>
                </dependencies>

                ....

        </project>
```

where the absence of any attribute implies a cross-platform artifact.

7.2.4. Dependency Post-Processing

The capability of performing arbitrary post-processing on dependency artifacts after download is a very natural requirement for a dependency management system. Post-processing support exists in Fragma at the time of the writing, but is unfortunately quite rudimentary. Fragma can only expand GZIP files and extract the content of TAR/GZ archives. However, future releases will add much better (we hope) support for generic dependency post-processing.

7.2.4.1. Automatic TAR.GZ Extraction

If a dependency artifact is a TAG.GZ archive, Fragma will automatically extract the content of the archive after download.Fragma will recognize a TAR.GZ archive after its "`tar.gz`" extension.

No extra configuration or procedural steps are required for enable this behavior.

For more informations on how to publish TAR.GZ archives, see the "Automatic Directory Archival" section.

7.2.4.2. Automatic GZIP Expansion

The same automatic behavior applies to GZIP archives. If an artifact is identified as a GZIP archive after its "`.gz`" extension, Fragma will automatically expand it after download.

For more informations on how to publish compressed files, see the "Optional File Compression" section.

7.3. Artifact Section

This is the section of the project metadata file where the project formally declares its "publishable" artifacts. These artifacts will be uploaded into the artifact repository as the result of a "publish" command.

An example of a typical artifact section follows:

```
<project>

        ....

        <artifacts>
             <artifact name="TREX.jar" local-path="./output/lib"/>
        </artifacts>
```

```
        ....

</project>
```

An artifact can be uploaded once and only once in the artifact repository; the rationale for this decision is as follows:

The state of your project at a certain moment in time is given by the snapshot of the code base and the snapshot of the dependencies. If you release and ship, and later you find a problem with that specific release, you want to be able to recreate the state of the project at the moment of the release. This is necessary so you can compile, debug, replicate and hopefully fix the problem, on the *exact* same codebase and dependencies you used to create the shipped artifacts in the first place.

Given a SVN tag, you are able to pull the *exact* code base snapshot. Part of the code base snapshot is your dependency configuration, maintained in the project metadata file. Based on the content of your project metadata file, you will be able to pull the *exact* dependency configuration.

Now, if you modify one of those dependencies by overwriting it in the repository, the possibility of recreating the state of the project, as it was at the moment of tagging, is compromised. Therefore the safest way of keeping state is just adding new artifacts, never deleting or modifying existing ones.

7.3.1. Publishing Artifacts with Attributes

The example above is one of the simplest possible, in that the artifact being published is platform and architecture independent, so only one artifact instance (the `TREX.jar` file) will be uploaded to the repository. However, one may be want to able to specify that an artifact has been built using specific attributes, and propagate this piece of information into the repository.

There are two ways to declare attributes while publishing artifacts:

7.3.1.1. Explicit Attribute Declaration

Attributes can be explicitly declared in the artifact section of `project.xml`, for each artifact to be published, as in the following example:

```
<project>

        ....

        <artifacts>

              <artifact name="utilities.sa" local-path="./output/lib"
                        architecture="Linux"
                        platform="i686"
                        threading="yes"/>

        </artifacts>

        ....

</project>
```

```
```

Based on the above artifact configuration, Fragma will upload the file "`utilities.sa`" (which it expects to find in `./output/lib`) and associate it with the given `architecture`, `platform` and `threading` attributes.

7.3.1.2. Implicit Attributes

If the legacy build system or any other module that produces artifacts names them according to Fragma's attribute naming convention, Fragma will automatically recognize and extract attributes from the respective artifact file names. Consequently, the attributes will be propagated into the artifact repository, upon upload.

In order to publish attributes using the implicit attribute naming convention, the artifact should be declared as follows:

```
<project>
        ....
        <artifacts>
              <artifact name="utilities.sa" local-path="./output/lib"/>
        </artifacts>
        ....
</project>
```

For the above example to work, the artifact bearing attributes encoded in its name (i.e. `a=Linux,p=i686,t@utilities.sa`) should me made available under `./output/lib`.

> **Warning**
> Implicit and explicit attribute declaration are mutually exclusive. Declaring attributes for the same artifact both in `project.xml` and within the artifact's file name is interpreted by Fragma as an error.

7.3.2. Artifact Pre-Processing

Fragma is capable of performing simple automatic pre-preprocessing tasks on artifacts just before they are published in the artifact repository. At the time of the writing, such capabilities are limited to automatical directory archival and optional individual file compression. It is quite possible that future releases will come with more sophisticated (and better configurable) artifact pre-processing features.

7.3.2.1. Automatic Directory Archival

If a directory is declared as artifact, Fragma will automatically turn it into a TAR.GZ archive prior publishing. No extra configuration or procedural steps are necessary.

For example, if "`my-directory`" is a directory and at the same time a project artifact declared as follows

```
<artifact name="my-directory" local-path="./output"/>
```

the directory will be archived and published in the artifact repository as "`my-directory.tar.gz`". The archival process is automated, as well as the expansion upon download. For more details on how archived artifacts are handled on download, see the " Automatic TAR.GZ Extraction" section.

7.3.2.2. Optional File Compression

Fragma can optionally compress individual files declared as artifacts and store them in compressed format in the artifact repository.

In order to enable compression, use the `compress` attribute in the artifact declaration. The attributes carries a value representing the compression type. At the time of the writing, only GZIP compression is supported (designated as "`gz`").

Assuming that the project declares a standalone executable as artifact, and you wish to store that executable in compressed format, the artifact declaration should look similar to:

```
<artifact name="myExecutable.exe"
          local-path="./output"
          compress="gz"
          [other_optional_attributes]/>
```

Artiacts using implicit attributes can be compressed on-the-fly also. In order to compress an artifact declaring its attributes implicitly, insert "`compress=gz`" among artifact's implicit attributes.

Example:

```
a=current,p=current,compress=gz@myExecutable.exe
```

For more details about implicit artifact attributes, see the "Implicit Attributes" section.

7.4. Compatibility Section

Expressing backward and forward compatibility via metadata, and then implement support for this behavior in Fragma is still work in progress.

7.5. Global Configuration Section

The global configuration section specifies global elements such as the location and credentials of the artifact repository, per-project Fragma logging behavior, etc.:

7.5.1. Repository Specification

The repository specification section contains the address (URL) of the artifact repository, used by Fragma both as a source of dependency artifacts and a publishing destination.

It is not unconceivable, but actually rather common to use different URLs to do fetching and publishing. One of the reasons for this behavior is that we may want to use different network protocols for fetching and publishing, or we may want to use completely separated physical repositories alltogether.

In case the access URL is the same for downloading from and uploading to repository, the repository configuration section looks similar to:

```
<project>

        ....

        <repository>file://opt/fragma/repository</repository>

</project>
```

If you need different URL to download from and upload to the repository, you can use a configuration similar to:

```
<project>

        ....

        <repository>

              <download url="http://example.com"/>

              <upload url="ftp://example.com"
                      username="fragma"
                      password="unfortunatelyinclear"/>

        </repository>

</project>
```

At the time of the writing we support the following protocols:

- File copy for both download and upload (file://).
- HTTP for download (http://).
- FTP for upload (ftp://).

8

The Artifact Repository

8.1. Artifact Repository Update Policy

The current Fragma implementation assumes that *once an artifact is in the repository, it does not change*. New releases can be added, of course, but if the artifact of an existent release is replaced in the repository, current Fragma version won't be able to detect that. Change detection logic could be added (https://jira.3dgeo.com/browse/TRE-72), but this is not a high priority, as modifying repository artifacts is strongly discouraged and should never actually happen.

Intermediate artifacts (such as CRs) *could* be removed from the repository to keep it at a reasonable size, but if this is done, you need to be aware of the fact that build will fail for any dependent projects who happen to declare a dependency on that specific artifact version.

This section is still work in progress.

9

Integration with the 3DGeo Build System

3DGeo's legacy build system, based on GNU Make, is used to compile and link C, F77 and F90 source code. This build system integrates with fragma via an automatically-generated text file, '`.fragma.metadata`', created during the fragma "fetch" operation. This file, located in the same directory as `project.xml`, exports both version and dependency path information. This information is made available to the legacy build system by importing the file in the project `Makefile`.

To update CVS-based projects replace the following `Makefile` line:

```
include buildsystem/Makedefaults
```

with these lines:

```
include .fragma.metadata
BUILDSYSTEMDIR := $(buildsystem)
include $(BUILDSYSTEMDIR)/Makedefaults
```

and ensure that the following line at the end of the `Makefile`:

```
include buildsystem/Makevars
```

is replaced with:

```
include $(BUILDSYSTEMDIR)/Makevars
```

Note that the legacy build system should be added to the `project.xml` file as an additional dependency:

```
<project>
        ....
```

```
          <dependencies>

               <!--
                    make-based buildsystem
               -->
               <project id="buildsystem"
                        version="1.0.Beta7"
                        prefix="3dgeo"
                        artifacts="buildsystem.tar.gz"/>

          </dependencies>

          ....

  </project>
```

After these changes have been made, running '`make`' on the command line will cause the project to be compiled and linked as before, generating machine- and architecture-specific executables. Dependencies, rather than being checked out from CVS, will be placed in user work space by fragma; these dependencies will be versioned and defined explicitly in the `project.xml` file, as will project version information.

The legacy build system automatically generates and compiles the self-doc file for each project. This self-doc is based on help files under the project `doc` directory, and project-specific information exported by fragma into the `.fragma.metadata` file.

The following keyword mappings will be made during the transition to fragma and SVN, ensuring backwards-compatibility with existing help files:

```
      @@projectname@@ -> NAME
      @@releasename@@ -> NAME
      @@release@@ -> VERSION
      @@build@@ -> SVN_REVISION
```

For example, if the help file contains the following:

```
@@projectname@@ @@release@@-@@build@@, built @@date@@

NOTE:

  /~~~~~~~~/| @@releasename@@ @@release@@-@@build@@
 /________/ | Built: @@date@@
|         | |
|     /###| | AUTHORS:
|####/oooo| |
|    ooo/ | |
|_________|/
```

and the `.fragma.metadata` file contains:

```
ID=FAST
NAME="FAST"
VERSION=1.0.TRE_TEST
SVN_REVISION=r120
```

The auto-generated self doc will contain:

```
FAST 1.0.TRE_TEST-r11234, built 12-20-2007 15:52


  /~~~~~~~~/| FAST 1.0.TRE_TEST-r11234
 /________/ | Built: 12-20-2007 15:52
|         | |
|     /###| | AUTHORS:
|####/oooo| |
|    ooo/ | |
|_________|/
```

10

Integration with Ant

As per 2.0.CR2, fragma does not support formal integration with Ant, but this feature will be added soon.

11

Integration with SVN

As per 2.0.CR2, fragma only supports reading the SVN release tag from the local SVN work area. However, SVN integration be extensively improved in the upcoming releases.

12

Integration with JIRA

As per 2.0.CR2, fragma does not support formal integration with JIRA, but this feature will be added soon.

13

Command Line Reference

13.1. `help`

Displays the interactive help.

13.2. `info`

Displays current project information (project name, project version, etc.). The "current project" is defined by the '`project.xml`' metadata file located in the directory fragma is run from, or by the `project_dir/project.xml` file, if '`-basedir`' global option is used.

13.3. `fetch`

Updates the content of the local dependency cache with the latest artifacts available in the artifact repository. A physical file transfer is only performed if the required dependency is not already available locally.

13.4. `publish`

Publishes current project's declared artifacts into the artifact repository.

In order to publish *all* artifacts declared in `project.xml`, from your project directory run:

```
$ fragma publish
```

fragma also allows publishing individual artifacts that are not necessarily declared in `project.xml`. However, in this case, publishing has to be made from the context of a valid project, because this is where fragma reads the repository's upload URL from. This limitation may be removed in a future release. The syntax to publish a standalone artifact is:

```
cd <your-project>
fragma publish -projectid <pid> -version <version> [-prefix <prefix>] [attributes] <filename>
```

where `[attributes]` are the usual `-architecture`, `-platform`, etc.

The name the artifact will be published under will be the name of the file being uploaded.

Example:

```
cd TREX

fragma publish -projectid TREX -version 1.0 -prefix example \
-architecture current -platform current ./output/libutil.sa
```

13.5. `version`

Displays fragma version information (current version, SVN revision tag and release date).