JHOVE2: Next-Generation Architecture for Format-Aware Characterization

JHOVE2 User's Guide

Version: 2.0.0
Issued: February 22, 2011
Status: Final

Copyright © 2010 by The Regents of the University of California, Ithaka Harbors, Inc., and The Board of Trustees of Leland Stanford Junior University. All rights reserved.

This manual is licensed under the Creative Commons Attribution-Share Alike 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

JHOVE2 Project Team

California Digital Library
Stephen Abrams
Patricia Cruse
John Kunze
Isaac Rabinovitch
Marisa Strong
Perry Willett

Portico
John Meyer
Sheila Morrissey

Stanford University
Richard Anderson
Tom Cramer
Hannah Frost

Library of Congress
Martha Anderson
Justin Littman

With help from
Walter Henry
Nancy Hoebelheinrich
Keith Johnson
Evan Owens

Preface

JHOVE2 is a Java framework and application for next-generation format-aware characterization. Characterization is the process of examining a formatted digital source unit and automatically extracting or deriving representation information about that source unit that is indicative of its significant nature and useful for purposes of classification, analysis, and use. For more information, visit http://jhove2.org.

Note: JHOVE2 uses the term "source unit" to refer to the unit of characterization. A source unit can be an entire file, a subset of a file, or a set of files that collectively form a single coherent work.

The *JHOVE2 User's Guide* documents procedures for installing and running JHOVE2. Although JHOVE2 is designed to run on any system that supports Java 6, the manual confines itself to systems running common versions of Microsoft Windows, Macintosh OS X, Linux, and Solaris.

Note: To avoid repetition, names of files are generally specified with POSIX file separators (**/**). Windows file separators (****) are used only when the file is Windows-specific or different under Windows.

Note: In the PDF version of this document, all references to sections, figures, tables and page numbers are clickable links.

Acknowledgments

The JHOVE2 project is funded by the Library of Congress as part of its National Digital Information Infrastructure Preservation Program (NDIIPP).

Version History

Version	Date	Notes
2.0.0	September 10, 2010	First public release of document.
2.0.0	February 22, 2011	Add configuration information for persistence management

Contents

Preface 3
Acknowledgments 3
Version History 3
Contents 4
Figures 5
Tables 6
System Requirements 7
Installation 8
Setting Up Java 8
Check the Current Java Installation 8
Install or Update Java 9
Locating Your Java Software 9
Locate Java on Windows 10
Locate Java on Linux 10
Locate Java on OS X 11
Locate Java on Solaris 11
Downloading and Extracting JHOVE2 12
Customizing the JHOVE2 Scripts 13
Set JAVA_HOME 13
Set Other Variables 14
Installing OpenSP 15
Install OpenSP on Windows Using Native Binaries 16
Install Cygwin OpenSP on Windows 16
Install OpenSP on Linux 18
Install OpenSP on OS X 18
Install OpenSP on Solaris 19
Configuring JHOVE2 19
Safely Editing XML Configuration Files 19
Configure Digest Output 19
Configure Memory Management 21
Configure the SGML Module 25
Configure Unit and Displayer Properties 26

Defining Assessment Rules 28
Conceptual Outline of an Assessment Ruleset 28
Encoding Assessment Rules Using `arules` 30
An Example of `arules` Input 30
Rules Files Usage and Syntax 31
Composing Predicates in MVEL 33
Running JHOVE2 36
Output Options 37
Temporary File Options 37
I/O Buffer Options 37
Help Options 38
How JHOVE2 Identifies Source Units 39

Figures

Figure 1: Windows Java Directory Example 10
Figure 2: JHOVE2 Home Directory Listing 13
Figure 3: Cygwin Package Selection Dialog 17
Figure 4: Package Dialog With OpenSP Selected for Installation 18
Figure 5: Distributed Digest Configuration 20
Figure 6: Tag for MD2 Digester 20
Figure 7: Digest Configuration with MD1 added. 21
Figure 8: Specifying `persistence.properties` for Persistent memory manager 22
Figure 9: Specifying Beans for Persistent memory manager 22
Figure 10: Specifying the directory to which JHOVE2 saves memory objects 23
Figure 11: Specifying `persistence.properties` for In-Memory memory manager 23
Figure 12: Specifying Beans for In-Memory memory manager 24
Figure 13: SGML Module Configuration for Windows Native Binaries 25
Figure 14: SGML Module Configuration for Cygwin, Linux, OS X and Solaris 25
Figure 15: Specifying the SGML Catalog File 25
Figure 16: Unit Property Example 26
Figure 17: Displayer Property Example 27
Figure 18: `arules` Input for XmlRuleSet 30
Figure 19: Assessment Output 31
Figure 20: MockModule Properties 34
Figure 21: MockReportable 35

Tables

Table 1: Linux JDK and JRE Directory Examples 11
Table 2: Provided Digester Beans 20
Table 3: Displayer Property Keywords 27

System Requirements

JHOVE2 has two system requirements:

- JHOVE2 as a whole requires a Java 6 runtime.
- The OpenSP SGML parser is required to fully characterize SGML source units. This requirement can be ignored if detailed SGML characterization is not required.
- If you want to rebuild JHOVE2 from the provided source, you will need the following:
  - A full Java SE 6 development kit (not just a runtime).
  - The Apache Maven project tool, available from http://maven.apache.org/.

Note: JHOVE2 has been tested on Sun's implementation of Java 6 on Windows, Solaris, and Linux, and on Apple's implementation of Java 6 on Mac OS X. JHOVE2 is designed to work with any implementation that is fully compliant with the official specification for Java 6 (including implementations running on other operating systems) but implementations other than the ones listed above have not been tested.

Installation

Installation consists of the following steps:

- Setting up Java (page 8).
- Downloading and extracting the JHOVE2 archive (page 12).
- Customizing the JHOVE2 script, if necessary (page 13).
- Installing OpenSP if necessary (page 15).
- Configuring JHOVE2, if necessary (page 15).

Setting Up Java

Setting up Java for use with JHOVE2 consists of the following steps:

- Check the Current Java Installation (page 8).
- Install or Update Java (page 9).

Check the Current Java Installation

To check for an existing installation of Java 6, open your system's command line and enter this command:

```
java –version
```

This should produce version information similar to this text:

```
java version "1.6.0_20"
Java(TM) SE Runtime Environment (build 1.6.0_20-b02)
Java HotSpot(TM) Client VM (build 16.3-b01, mixed mode, sharing)
```

The "java version" string follows a standard. If the first part of the string is "1.6" (or a larger number) then you are running Java 6 (or later), and JHOVE2 will run.

Since multiple versions of Java can be installed, it is possible for the version string to indicate an older version even if Java 6 is installed. If you think this may be the case, check the value of **JAVA_HOME**; if the variable is set, try unsetting it, as described in the section "Set **JAVA_HOME**".

If the system reports that the **java** command could not be found, it probably means that Java is not installed. If in doubt, install Java 6; this might mean having multiple versions of Java installed, but this should not present problems.

> Note: If you've previously installed Java, but the **java** command produces a "not found" message, it's possible that that the Java tools are not in your execution path. You don't need to add Java to your execution path to run JHOVE2, but you do need to verify that you have Java 6 or later. Refer to the documentation for your Java software for more information.

Install or Update Java

Java can be installed or updated from two sources: installation files downloaded from Sun web sites and through the package or update managers available in most operating systems.

> Note: The Windows Update Manager does not support Java; Windows users must download an installation file.

> Note: Apple includes Java in OS X; new versions are usually available only through the update manager. Sometimes beta releases are available through the Apple Developer web site at http://connect.apple.com.

To download Java software for Windows, Linux, or Solaris, visit http://java.sun.com/javase/downloads/. This page gives you the option of downloading the Java SE Development Kit (JDK) or the Java Runtime Environment (JRE). The JRE, which is a smaller package, is sufficient to run JHOVE2. If you plan to develop JHOVE2 modules, download the JDK, which includes basic developer tools, including a compiler.

For information on installing Java through the update or package manager, refer to your operating system documentation.

Locating Your Java Software

In most cases, JHOVE2 is able to locate the installed Java software without special configuration. If JHOVE2 is unable to locate the correct version of Java, set the **JAVA_HOME** variable to point to the Java home directory, as described in the section "Set JAVA_HOME". Before you can set this variable, you have to know where the Java software is installed. This section provides instructions for various platforms:

- "Locate Java on Windows" (page 10).
- "Locate Java on Linux" (page 10).

- "Locate Java on OS X" (page 11)
- "Locate Java on Solaris" (page 11)

Locate Java on Windows

On Windows, Java is usually installed under the directory **`C:\Program Files\Java`**. This directory can contain multiple versions of Java. Each Java home directory name begins with "jdk" or "jre", and includes a version string. Figure 1 is a listing for a **`C:\Program Files\Java`** on a system that contains two JDK versions and two JRE versions.

Figure 1: Windows Java Directory Example

```
Jun 02  09:31 AM    <DIR>          jdk1.5.0_22
Jun 02  11:39 AM    <DIR>          jdk1.6.0_20
Jun 02  09:36 AM    <DIR>          jre1.5.0_22
Jun 02  11:31 AM    <DIR>          jre6
```

On this system **`JAVA_HOME`** could be set either to **`C:\Program Files\Java\jdk1.6.0_20`** or to **`C:\Program Files\Java\jre6`**. Neither of the two Java 5 directories works, since JHOVE2 requires Java 6.

Locate Java on Linux

There are three common locations for Java under Linux, depending on how it was installed.

- If Java was installed by a package manager, it is normally in a subdirectory of **`/lib/jvm`**.
- If Java was installed by Sun's RPM-based installation script, it is normally in a subdirectory of **`/usr/java`**.
- If Java was installed by Sun's plain installation script, it is in a subdirectory of the script's working directory.

In each case, the subdirectory name consists of a descriptive string followed by a version number. Table 1 contains some examples.

Table 1: Linux JDK and JRE Directory Examples

Directory	Notes
/lib/jvm/java-1.6.0-openjdk-1.6.0.0	Java 6 JDK or JRE from the OpenJDK project, installed by a package manager.
/lib/jvm/java-6-gcj-4.4	Java 6 JRE for GNU's GCJ compiler, installed by a package manager.
/lib/jvm/java-6-sun-1.6.0.20	Java 6 JDK or JRE from Sun, installed by a package manager.
/usr/java/jdk1.6.0_20	Java 6 JDK from Sun, installed by a script.
/usr/java/jre1.6.0_20	Java 6 JRE from Sun, installed by a script

Sun scripts create directory names that begin "jdk" or "jre", depending on whether a JDK or JRE is installed. Package managers do not follow this convention. Package managers also follow different conventions for organizing individual directories within a Java software installation.

You do not need to know all the different ways these directories can be organized. Simply remember that **JAVA_HOME** must point to a directory that contains a **bin** directory (with Java executables) and a **lib** directory (with Java libraries). If you look in the home directory of a Java software installation and don't see **bin** or **lib** directories, look for a **jre** directory, and point **JAVA_HOME** at that. As an example, consider the OpenJDK directory in Table 1.

- If the directory contains **bin**, **lib**, and **jre** subdirectories, set **JAVA_HOME** to **/lib/jvm/java-1.6.0-openjdk-1.6.0.0.**
- If the **bin** and **lib** subdirectories are missing, set **JAVA_HOME to /lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre**

Locate Java on OS X

OS X provides a command line tool, **/usr/libexec/java_home**, for discovering the Java home directory. Multiple versions of Java can be installed; the specific Java version used is configured by the Java Preferences Application. For more information, see *Technical Q&A QA1170: Important Java Directories on Mac OS X* at

http://developer.apple.com/mac/library/qa/qa2001/qa1170.html

Locate Java on Solaris

Solaris comes with Java preinstalled; the Java home directory is **/usr/j2se**.

Sun also provides downloadable scripts and compressed archives for installing Java. These create a Java installation as a subdirectory of the current directory. This subdirectory is the home directory of the Java installation; it is named **jdk-*xxxx*** or **jre-*xxxx***, where ***xxxx*** is a version string.

Downloading and Extracting JHOVE2

JHOVE2 ZIP archives and compressed TAR archives are available at http://jhove2.org. On Windows, support for ZIP archives is built into Windows Explorer. On other platforms, a wide variety of GUI and command line tools are available. This section describes the most common command line tools.

The archive files are named **jhove2-*X.X.X*.zip** or **jhove2-*X.X.X*.tar.gz**, where ***X.X.X*** is the version string. Similarly, all files in the archives have relative path names that begin with the directory **jhove2-*X.X.X***, where ***X.X.X*** is the version string. The following examples assume that the version string is **2.0.0**, which may be different from the current version string. The examples extract the archives without specifying a destination directory, so the directory **jhove2-2.0.0** is created in the current working directory.

- GNU tar: this utility is preinstalled in OS X, Solaris and most Linux systems. On Solaris, GNU tar has a different name and is not on the default execution path.

  The following example works on Linux and OS X. It extracts a JHOVE2 compressed TAR archive to the working directory:

  ```
  tar -xzvf jhove2-2.0.0.tar.gz
  ```

  Here is the Solaris version of the same example:

  ```
  /usr/sfw/bin/gtar -xzvf jhove2-2.0.0.tar.gz
  ```

  Note: GNU tar archives are not fully compatible with the default Solaris tar.

- INFO-ZIP utilities: available on most platforms, preinstalled in popular Linux distributions. The following example extracts a JHOVE2 ZIP archive to the working directory:

  ```
  unzip jhove2-2.0.0.zip
  ```

Once the archive is extracted, the current directory should contain a new directory **jhove2-2.0.0**. The directory listing should be similar to the one shown in Figure 1.

Figure 2: JHOVE2 Home Directory Listing

```
ANTLR-LICENSE.txt     JHOVE(1)LICENSE.txt  LICENSE.txt
APACHE-LICENSE.txt    jhove2.cmd           Log4j-LICENSE.txt
arules.cmd            jhove2.sh            MVEL-LICENSE.txt
arules.sh             jhove2_doc.cmd       OpenSP-LICENSE.txt
config                jhove2_doc.sh        pom.xml
DROID-LICENSE.txt     jhove2_dpfg.cmd      README.txt
env.cmd               jhove2_dpfg.sh       Spring-LICENSE.txt
env.sh                jhove2_upfg.cmd      src
GEOTOOLS-LICENSE.txt  jhove2_upfg.sh       W3C-TEST-SUITE-LICENSE.txt
JARGS-LICENSE.txt     JUNIT-LICENSE.txt    WorldBordersDataset-License.txt
JDOM-LICENSE.txt      lib
```

Customizing the JHOVE2 Scripts

JHOVE2 is invoked using a script. This script is available in two forms: **`jhove2.cmd`**, suitable for Windows, and **`jhove2.sh`** suitable for Linux, OS X, and Solaris. In most cases, the JHOVE2 scripts should run correctly without customization. Some possible exceptions:

- If JHOVE2 is unable to find the basic Java 6 libraries, the **`JAVA_HOME`** environment variable must be set (page 13).
- Other variables might need to be set to help the scripts find the Java launcher tool, the JHOVE2 home directory, and the various library JAR files required to run JHOVE2 (page 14).

The following subsections cover these two use cases.

Set `JAVA_HOME`

In most cases, JHOVE2 will work correctly without **`JAVA_HOME`** being set. If JHOVE2 seems to be using the wrong version of Java, begin by making sure **`JAVA_HOME`** has not been set to a bad value by the system configuration. Do this from the command line. In Windows (note the trailing %):

```
echo %JAVA_HOME%
```

On Linux, Solaris, or OS X:

```
echo $JAVA_HOME
```

If **`JAVA_HOME`** is not set, these commands will produce a blank line or the message "JAVA_HOME: `Undefined variable.`" `If` **`JAVA_HOME`** seems to point to an old or missing version of Java, try unsetting the variable. On Windows, use this command:

```
set JAVA_HOME=
```

On Linux, OS X, or Solaris, the command depends on your user shell. For **sh** or **bash**:

```
unset JAVA_HOME
```

For **csh** or **tcsh**:

```
unsetenv JAVA_HOME
```

If you determine that **JAVA_HOME** needs to be set, discover the correct value, as described in the section "Locating Your Java Software". Then set the variable with the appropriate command. Here is an example for Windows:

```
set JAVA_HOME=c:\Program Files\Java\jdk1.6.0_20
```

Here is an example for **sh** or **bash**:

```
JAVA_HOME=/lib/jvm/java-1.6.0-openjdk-1.6.0.0; export JAVA_HOME
```

Here is an example for **csh** or **tcsh**:

```
setenv JAVA_HOME /lib/jvm/java-1.6.0-openjdk-1.6.0.0
```

You can change the value of **JAVA_HOME** assigned by your system:

- On Windows, run the System Properties application (**sysdm.cpl** from the command line), go to the "Advanced" tab, and click on "Environment Variables" button. Make sure that "User Variables" and "System Variables" do not have conflicting values for **JAVA_HOME**.
- On Linux and Solaris, add the appropriate commands to the user or system initialization scripts.
- On OS X, edit the appropriate property list. For more information, see the section "Environment Variables" in the document *Runtime Configuration Guidelines* at http://developer.apple.com/mac/library/documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/EnvironmentVars.html.

Set Other Variables

The JHOVE2 scripts are designed to support customization by user editing of the code that sets the following variables:

JAVA

Specifies the command that launches the Java runtime. If **JAVA_HOME** is not set, **JAVA** defaults to **java**, so that the normal launcher tool is assumed to be in the execution path. If **JAVA_HOME** is set, **JAVA** is set to the full path name of the launcher executable in the **bin** subdirectory.

`JHOVE2_HOME`

Specifies the directory that contains the library, configuration, and other files created when JHOVE2 was installed. By default, the scripts assume that this is the directory that contains the scripts.

`CP`

Class path that is passed to the **`JAVA`** command. By default, this includes all the JAR files in the **`lib`** subdirectory of **`JHOVE2_HOME`**.

JHOVE2 scripts set these variables by calling **`env.cmd`** (Windows) or **`env.sh`** (Linux, OS X, and Solaris). Edit these files to customize the way these variables are set in all JHOVE2 scripts.

Installing OpenSP

Note: OpenSP is a third-party open source SGML parser. It is required for full characterization of SGML files. If OpenSP is not available, the SGML module provides a limited characterization, which includes a message about its failure to run OpenSP.

Note: After OpenSP is installed, configure the JHOVE2 SGML module to find it. For more information, see "Configure the SGML Module" on page 21.

This section describes OpenSP installation procedures on the following platforms:

- Windows:
  - To install the native binaries provided by the OpenJade project, refer to "Install OpenSP on Windows Using Native Binaries" on page 16.
  - To install the Cygwin version of OpenSP, refer to "Install Cygwin OpenSP on Windows" on page 16.

  Note: After OpenSP is installed, configure the JHOVE2 SGML module to find it. For more information, see "Configure the SGML Module" on page 21.

- Linux: "Install OpenSP on Linux on page 18.
- OS X: "Install OpenSP on OS X" on page 18.
- Solaris: "Install OpenSP on Solaris" on page 19.

Install OpenSP on Windows Using Native Binaries

The OpenJade project provides Windows binaries for OpenSP. Unlike the Cygwin version, this native version does not require a special runtime. To install this version of OpenSP:

1. Download the ZIP file containing the binaries from http://sourceforge.net/projects/openjade/.
2. Extract the binaries to a convenient location, such as **`C:\Program Files\OpenSP`**.

Note: This version of OpenSP does not come with a catalog file. In any case SGML catalog files are normally customized to support the specific SGML file type being parsed. JHOVE2 itself comes with a catalog file that supports HTML files in **`src\test\resources\examples\sgml\catalog`**.

Note: After OpenSP is installed, configure the JHOVE2 SGML module to find it. For more information, see "Configure the SGML Module" on page 21.

Install Cygwin OpenSP on Windows

Cygwin is a Linux-compatible runtime environment for Windows. Though software must be specially compiled to run under Cygwin, precompiled Cygwin applications are available, including OpenSP.

This section describes installation of the Cygwin version of OpenSP on a system that does not currently have Cygwin installed. It assumes the user has no experience with Cygwin or Linux environments.

Installation of Cygwin OpenSP consists of these steps:

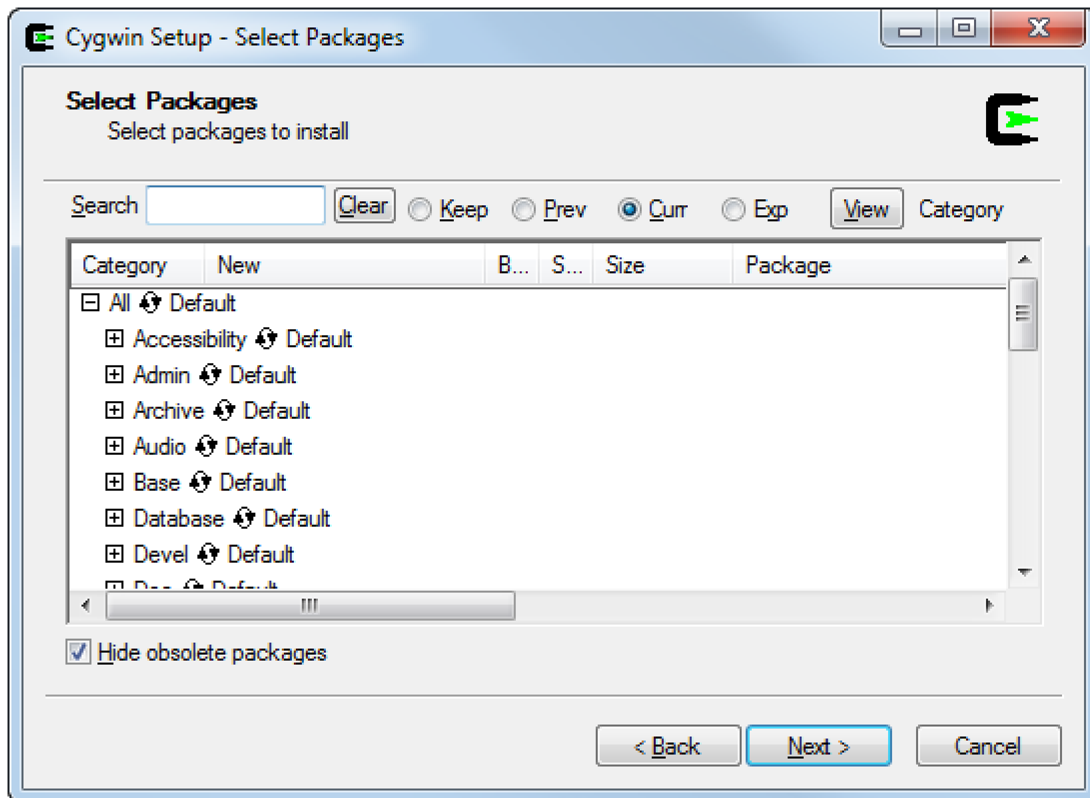
1. Visit http://cygwin.org and download **`setup.exe`**. This program is used to download and install Cygwin itself and also Cygwin applications. It does not require installation; simply save the file to a convenient location.
2. Run **`setup.exe`**.
3. A dialog box with information about the Cygwin installation process appears. Click on "Next".
4. A dialog box titled "Choose a Download Source" appears. Choose "Install from Internet" and click on "Next".
5. A dialog box titled "Select Root Install Directory" appears. Leave these values unchanged and click on "Next".
6. A dialog box titled "Select Local Package Directory" appears. Choose a convenient directory location and click on "Next".

7. A dialog box titled "Select Your Internet Connection" appears. This is used to configure any proxy you must use to access the Internet. In most cases "Direct Connection" will work; if it does not, contact your system administrator for proxy information.

8. A dialog box titled "Choose a Download Site" appears. Any site that provides a reasonably responsive connection will do. If your choice turns out to be unresponsive, back up the process to this dialog and choose another site.

9. There is a short delay while a package list is downloaded. Once this download is complete, a package selection window appears, similar to the one in Figure 3.

Figure 3: Cygwin Package Selection Dialog

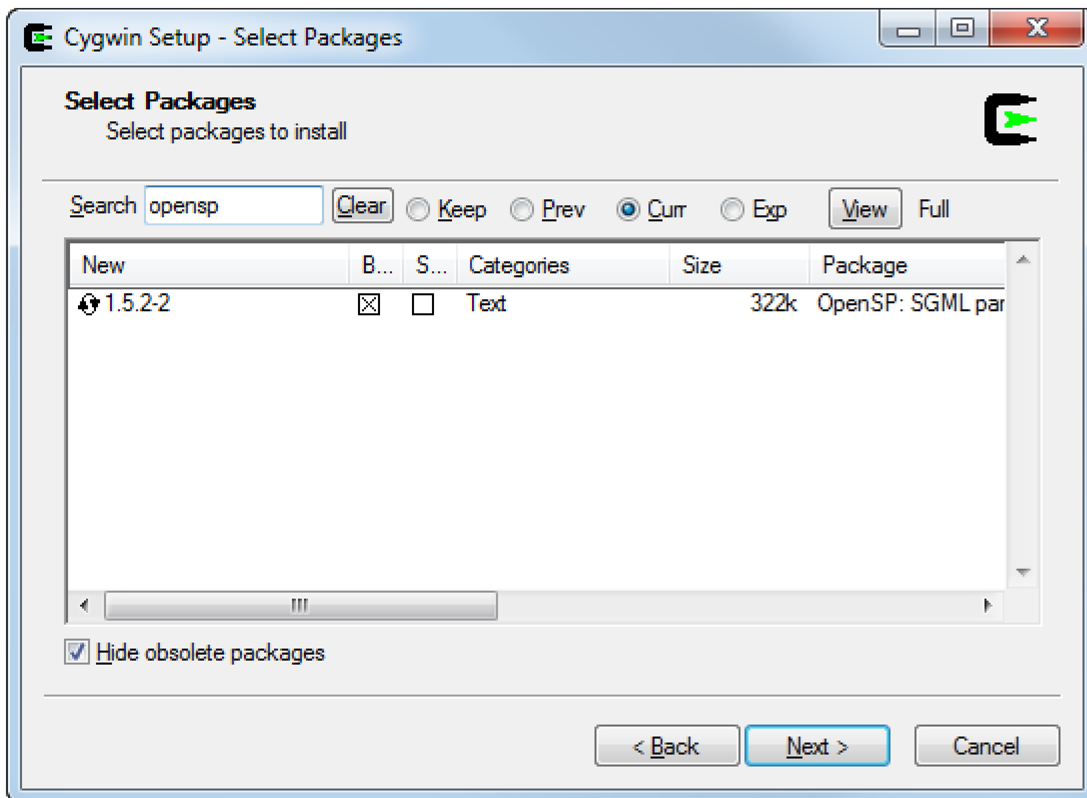


10. Use the "View" button to cycle between list views until the "Full" view appears (see the text next to the button).

11. In the "Search" text box, type "OpenSP". All packages that don't have this string in their descriptions disappear.

12. If the first column for the OpenSP package says "Skip", click on it. The word "Skip" should change to a version string and the dialog should look like Figure 4.

Figure 4: Package Dialog With OpenSP Selected for Installation



13. Click on "Next". a progress dialog appears.

14. After several minutes, the final Cygwin dialog appears. You can choose to create icons that give you access to the Cygwin command line interface. In any case, click "Finish."

Note: After OpenSP is installed, configure the JHOVE2 SGML module to find it. For more information, see "Configure the SGML Module" on page 21.

Install OpenSP on Linux

OpenSP is available in repositories for leading Linux distributions. Search your package manager for "opensp".

Note: After OpenSP is installed, configure the JHOVE2 SGML module to find it. For more information, see "Configure the SGML Module" on page 21.

Install OpenSP on OS X

OpenSP for OS X is available from a variety of sources. Precompiled versions are available from the Fink Project (http://finkproject.org/) and from MacPorts (http://www.macports.org/). Source code, which

can be compiled with Apple's Developer Tools, is available from the OpenJade project (http://openjade.sourceforge.net/).

Install OpenSP on Solaris

To install OpenSP on Solaris, you must download and compile the source (available on http://openjade.sourceforge.net/). The necessary GNU tools are available provided Solaris was installed at the Developer level or above. Make sure that your execution path includes **`/usr/sfw/bin`**.

Note: After OpenSP is installed, configure the JHOVE2 SGML module to find it. For more information, see "Configure the SGML Module" on page 21.

Configuring JHOVE2

The **`config`** directory contains files that configure many options for the JHOVE2 framework and application. Most of these options are of interest only to programmers and are beyond the scope of this manual. This section describes options that are of interest to users of the JHOVE2 application. It contains the following subsections:

- "Safely Editing XML Configuration Files" on page 19.
- "Configure Digest Output" on page 19.

Safely Editing XML Configuration Files

Most JHOVE2 configuration files are in XML format and must be edited with care. To avoid making XML files unreadable, consider using these safeguards:

- Make backup copies of all files before editing them.
- Keep changes to a minimum. In many cases, the only change necessary is enclosing or disclosing text using XML comments.
- Make a few small changes at a time, and then test them before proceeding.
- If practical, use a validating XML editor. If the editor allows invalid XML to be entered, be sure to run the validator before saving the file.

Configure Digest Output

Note: Before editing any of JHOVE2's XML configuration files, read "Safely Editing XML Configuration Files" on page 19.

If the **`-k/--calc-digests`** option is specified on the command line, JHOVE2 calculates message digests (also known as hash values) for each file. The specific digests that are calculated are specified in **`config/spring/module/digest/jhove2-digest-config.xml`**. As distributed, the software supports the digests listed in Table 2.

Table 2: Provided Digester Beans

Digest	Type	Enabled Initially
Adler-32	Array	No
CRC-32	Array	Yes
MD2	Buffer	No
MD5	Buffer	Yes
SHA-1	Buffer	Yes
SHA-256	Buffer	No
SHA-384	Buffer	No
SHA-512	Buffer	No

The XML code that enables the CRC-32, MD5, and SHA-1 digests looks like Figure 5. It contains two lists of software components known as "digesters". Each digester, if enabled, generates a particular kind of digest for each source unit. Notice that some digesters are "array digesters" and others are "buffer digesters". This is an implementation detail that only concerns us because the two kinds of digesters appear in separate lists in the XML.

Figure 5: Distributed Digest Configuration

```
<property name="arrayDigesters">
    <list value-type="org.jhove2.module.digest.ArrayDigester">
        <ref bean="CRC32Digester"/>
    </list>
</property>
<property name="bufferDigesters">
    <list value-type="org.jhove2.module.digest.BufferDigester">
        <ref bean="MD5Digester"/>
        <ref bean="SHA1Digester"/>
    </list>
</property>
```

Each digester is configured by a **bean** tag later in the file. For example, the MD2 digester is configured by the XML in Figure 6.

Figure 6: Tag for MD2 Digester

```
<!-- MD2 message digester bean -->
<bean id="MD2Digester" class="org.jhove2.module.digest.MD2Digester"
      scope="prototype"/>
```

The **bean** attribute in the **ref** tag refers to an **id** attribute in the **bean** tag. Suppose we want to enable generation of MD2 digests. Since the MD2 is calculated by a buffer digester, we add a **ref** tag to the list of buffer digesters that points to the **bean** tag with **id** "MD2Digester". The result looks like Figure 7.

Figure 7: Digest Configuration with MD1 added.

```
<property name="arrayDigesters">
    <list value-type="org.jhove2.module.digest.ArrayDigester">
        <ref bean="CRC32Digester"/>
    </list>
</property>
<property name="bufferDigesters">
    <list value-type="org.jhove2.module.digest.BufferDigester">
        <ref bean="MD1Digester"/>
        <ref bean="MD5Digester"/>
        <ref bean="SHA1Digester"/>
    </list>
</property>
```

Configure Memory Management

Note: Before editing any of JHOVE2's XML configuration files, read "Safely Editing XML Configuration Files" on page 19.

This section describes a minimal configuration procedure required to configure memory management in the JHOVE2 application. Most configuration options are not described. For complete documentation of memory configuration management, refer to information for developers on the JHOVE2 wiki.

When processing a single file source, or a directory or container file such as a ZIP file that does not contain very many children (other files, directories, etc.), JHOVE2 can keep all the information it accumulates about those sources in the application's short term memory. However, if a large number of sources or a deeply nested container such as a directory or ZIP file is being processed, JHOVE2 has to augment its short-term memory by "persisting" the information to disk until it has completed processing all the sources, and written all the output about them.

There are two files that used to configure memory management in JHOVE2: an ASCII properties file (**config/properties/persistence/persistence.properties**), and an XML configuration file (**config/spring/persist/jhove2-persist-config.xml**). These files may be edited to configure JHOVE2 to use either "persistent" or "in-memory" memory manager (the default, "out-of-the-box" configuration, is the "persistent" memory manager).

Configure JHOVE2 Using "Persistent" Memory Manager

Figure 8: Specifying `persistence.properties` for Persistent memory manager

```
#classname org.jhove2.config.spring.SpringInMemoryPersistenceManagerFactory
classname org.jhove2.config.spring.SpringBerkeleyDbPersistenceManagerFactory
```

- Make sure the in-memory " classname" property setting is commented out with the pound ("#") sign
- Make sure the persistent "classname" property setting starts at the beginning of the line, without any comment ("#") code preceding it.

Figure 9: Specifying Beans for Persistent memory manager

```
<!--  Beans for in-memory persistence -->
<!--
<bean id="SourceFactory"             class="org.jhove2.persist.inmemory.InMemorySourceFactory"
scope="prototype"/>
<bean id="ApplicationModuleAccessor"
class="org.jhove2.persist.inmemory.InMemoryApplicationModuleAccessor"/>
<bean id="FrameworkAccessor"         class="org.jhove2.persist.inmemory.InMemoryFrameworkAccessor"
scope="prototype"/>
<bean id="FormatModuleAccessor"
class="org.jhove2.persist.inmemory.InMemoryFormatModuleAccessor" scope="prototype"/>
<bean id="FormatProfileAccessor"
class="org.jhove2.persist.inmemory.InMemoryFormatProfileAccessor" scope="prototype"/>
<bean id="AggrefierAccessor"         class="org.jhove2.persist.inmemory.InMemoryAggrefierAccessor"
scope="prototype"/>
<bean id="IdentifierAccessor"
class="org.jhove2.persist.inmemory.InMemoryIdentifierAccessor" scope="prototype"/>
<bean id="BaseModuleAccessor"
class="org.jhove2.persist.inmemory.InMemoryBaseModuleAccessor" scope="prototype"/>
-->

<!--  Beans for BerkeleyDB persistence -->

<bean id="SourceFactory"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbSourceFactory" scope="prototype"/>
<bean id="ApplicationModuleAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbApplicationModuleAccessor"/>
<bean id="FrameworkAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbFrameworkAccessor" scope="prototype"/>
<bean id="FormatModuleAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbFormatModuleAccessor" scope="prototype"/>
<bean id="FormatProfileAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbFormatProfileAccessor" scope="prototype"/>
<bean id="AggrefierAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbAggrefierAccessor" scope="prototype"/>
<bean id="IdentifierAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbIdentifierAccessor" scope="prototype"/>
<bean id="BaseModuleAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbBaseModuleAccessor" scope="prototype/>
```

- Make sure the beans for "in-memory persistence" have been commented out
- Make sure the beans for "BerkeleyDB persistence" have NOT been commented out

Figure 10: Specifying the directory to which JHOVE2 saves memory objects

```
<bean id="BerkeleyDbPersistenceManager"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbPersistenceManager"
     scope="singleton">
     <property name="envHome" value="C:\"/>
     <property name="entityStoreName" value="JHOVE2_Entity_Store"/>
```

- Make sure the "value" attribute of the "envHome" property is set to the name of the directory where you want the memory manager to save the contents of JHOVE2 memory while it is processing your source(s). Be sure to specify a directory to which you have permission to write files.

Configure JHOVE2 Using "In-Memory" Memory Manager

Figure 11: Specifying `persistence.properties` for In-Memory memory manager

```
classname org.jhove2.config.spring.SpringInMemoryPersistenceManagerFactory
#classname org.jhove2.config.spring.SpringBerkeleyDbPersistenceManagerFactory
```

- Make sure the in-memory " classname" property setting starts at the beginning of the line, without any comment ("#") code preceding it.
- Make sure the persistent "classname" property setting has been commented out with a comment ("#") code

Figure 12: Specifying Beans for In-Memory memory manager

```
<!--  Beans for in-memory persistence -->
<bean id="SourceFactory"            class="org.jhove2.persist.inmemory.InMemorySourceFactory"
scope="prototype"/>
<bean id="ApplicationModuleAccessor"
class="org.jhove2.persist.inmemory.InMemoryApplicationModuleAccessor"/>
<bean id="FrameworkAccessor"         class="org.jhove2.persist.inmemory.InMemoryFrameworkAccessor"
scope="prototype"/>
<bean id="FormatModuleAccessor"
class="org.jhove2.persist.inmemory.InMemoryFormatModuleAccessor" scope="prototype"/>
<bean id="FormatProfileAccessor"
class="org.jhove2.persist.inmemory.InMemoryFormatProfileAccessor" scope="prototype"/>
<bean id="AggrefierAccessor"         class="org.jhove2.persist.inmemory.InMemoryAggrefierAccessor"
scope="prototype"/>
<bean id="IdentifierAccessor"
class="org.jhove2.persist.inmemory.InMemoryIdentifierAccessor" scope="prototype"/>
<bean id="BaseModuleAccessor"
class="org.jhove2.persist.inmemory.InMemoryBaseModuleAccessor" scope="prototype"/>

<!--  Beans for BerkeleyDB persistence -->
<!--
<bean id="SourceFactory"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbSourceFactory" scope="prototype"/>
<bean id="ApplicationModuleAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbApplicationModuleAccessor"/>
<bean id="FrameworkAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbFrameworkAccessor" scope="prototype"/>
<bean id="FormatModuleAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbFormatModuleAccessor" scope="prototype"/>
<bean id="FormatProfileAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbFormatProfileAccessor" scope="prototype"/>
<bean id="AggrefierAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbAggrefierAccessor" scope="prototype"/>
<bean id="IdentifierAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbIdentifierAccessor" scope="prototype"/>
<bean id="BaseModuleAccessor"
class="org.jhove2.persist.berkeleydpl.BerkeleyDbBaseModuleAccessor" scope="prototype />
-->
```

- Make sure the beans for "in-memory persistence" have NOT been commented out
- Make sure the beans for "BerkeleyDB persistence" have been commented out

Configure the SGML Module

Note: Before editing any of JHOVE2's XML configuration files, read "Safely Editing XML Configuration Files" on page 19.

This section describes a minimal configuration procedure required to get the SGML module working. Most configuration options are not described. For complete documentation of the SGML module, refer to the *JHOVE2 SGML Module Specification*.

The SGML module configuration file is **`config/spring/module/format/sgml/jhove2-sgml-config.xml`**. This file must be edited as follows:

1. Set the **`sgmlParser`** property to point to the parser you installed, as described in "Installing OpenSP" on page 15. This is controlled by a block of XML code that looks like Figure 13.

   Figure 13: SGML Module Configuration for Windows Native Binaries

```
        <!-- if using OpenJade unix exe on Unix or via Cygwin on Windows -->
<!--        <property name="sgmlParser" ref="OpenSpWrapper"/> -->
        <!-- if using OpenJade Windows dll and executable -->
        <property name="sgmlParser" ref="OpenSpWrapperForWindowsExe"/>
```

   Notice that there are two tags named **`sgmlParser`**, with the first tag surrounded by SGML comment marks (**`<!--`** and **`-->`**). The tag that is not commented out has the attribute **`ref="OpenSpWrapperForWindowsExe"`**, meaning that the Windows native binaries are being used. If you are using OpenSp for Windows/Cygwin, Linux, OS X, or Solaris, comment out the second tag and uncomment the tag with **`ref="OpenSpWrapper"`**, as in Figure 14.

   Figure 14: SGML Module Configuration for Cygwin, Linux, OS X and Solaris

```
        <!-- if using OpenJade unix exe on Unix or via Cygwin on Windows -->
        <property name="sgmlParser" ref="OpenSpWrapper"/>
        <!-- if using OpenJade Windows dll and executable -->
<!--        <property name="sgmlParser" ref="OpenSpWrapperForWindowsExe"/> -->
```

2. Specify the name of your SGML catalog file. This is controlled by the XML in Figure 15.

   Figure 15: Specifying the SGML Catalog File

```
<bean id="SgmlCatalogPath" class="java.lang.String">
    <constructor-arg type="java.lang.String"
                     value="/usr/local/etc/sgml/catalog"/>
</bean>

<bean id="WindowsSgmlCatalogPath" class="java.lang.String">
    <constructor-arg type="java.lang.String"
                     value="C:\PROGRA~1\OpenSp\catalog"/>
</bean>
```

Notice that there are two **bea**n tags. The first, with **id="SgmlCatalogPath"**, specifies the catalog file on Linux, OS X, Solaris, and when the Cygwin version of OpenSP is used; otherwise it is ignored. The second, with **id="WindowsSgmlCatalogPath"**, specifies the catalog file when the native Windows version of OpenSP is used; otherwise it is ignored. In both cases, the name of the catalog file is specified in the inner **constructor-arg** tag using the **value** attribute.

Notice how the Figure 15 specifies the **C:\Program Files** folder with an 8.3 name, **C:\PROGRA~1**. This is necessary because the attribute cannot contain a space character. If you are unsure of the 8.3 name for a Windows file, use the command **dir /x** to discover it.

When the Cygwin version of OpenSP is used, file names can be specified with either Windows or POSIX file names. In Figure 15, a POSIX name (**/usr/local/etc/sgml/catalog**) is used. Under Cygwin, the Windows equivalent (**c:\cygwin\usr\local\etc\sgml\catalog**) would also work. Under "pure" POSIX systems (Linux, Solaris, OS X) only the POSIX name works.

Configure Unit and Displayer Properties

This section explains how to use Java property settings to select output values and labels. These properties are initialized from files in the **config/properties** directory.

There are two kinds of properties set this way: *unit* properties and *displayer* properties. Unit property files appear in **config/properties/unit**; displayer property appear in **config/properties/ displayer**.

Unit properties simply supply a label that appears next to certain numeric outputs. Changing the property simply changes the label; it does not affect the number displayed. An example appears in Figure 16.

Figure 16: Unit Property Example

```
http\://jhove2.org/terms/property/org/jhove2/core/source/ZipFileSource/Size      byte
```

The property name begins with the first non-blank character on the line, and is terminated by the first following blank. Property names are specified as URLs that specify the class unambiguously based on the domain name of the source (**jhove2.org**) and the fully-qualified name of the Java package and class

where the property is defined. Java property file conventions require that the colon (**:**) in the URL be escape with a backslash (****).

The property value appears after the trailing blank. This value specifies a label string associated with the numeric value of the property.

An example of a displayer property appears in Figure 17.

Figure 17: Displayer Property Example

```
http\://jhove2.org/terms/property/org/jhove2/core/format/Format/Specifications Never
```

As with unit properties, the line contains a property URL and value, separated by one or more blanks. Here the property value is a keyword. Displayer property keywords are described in Table 3.

Table 3: Displayer Property Keywords

Keyword	Meaning
Always	Value always displayed (default if property not set).
Never	Value never displayed.
IfTrue	Value displayed if it is Boolean "True". Non-Boolean value always displayed..
IfFalse	Value displayed if it is Boolean "False". Non-Boolean value always displayed.
IfNegative	Value displayed if it is numeric and less than 0. Nonnumeric value always displayed.
IfPositive	Value displayed if it is numeric and greater than 0. Nonnumeric value always displayed.
IfNonNegative	Value displayed if it is numeric and greater than or equal to 0. Nonnumeric value always displayed.
IfNonPositive	Value displayed if it is numeric and less than or equal to 0. Nonnumeric value always displayed.

Notice that if the keyword filters for Boolean or numeric data, data that does not match the data type is always displayed.

The property files provided with JHOVE2 provide settings for all Displayer properties, including those that normally are set to **Always**. Since **Always** is the default, setting these properties in the property files is redundant. However, listing all properties simplifies maintenance.

Defining Assessment Rules

Assessment rules are defined by the JHOVE2 user, and make assertions about the properties of source units. The Assessment Module produces a narrative description for a source unit that depends on whether each assertion is true or false for the source unit. This section is a practical guide to defining and implementing assessment rules as part of your local JHOVE2 configuration. For a more detailed explanation of assessment, refer to the *JHOVE2 Assessment Module Specification*.

Note: This section focuses on an example assessment of a particular source unit, the **pom.xml** file used to build JHOVE2. Like many XML files, this file omits an XML declaration. Without this declaration, DROID fails to recognize the file, and the XML module is never invoked. For the purposes of this example, we've added the following declaration to the beginning of the file:

```
<?xml version="1.0"?>
```

This section consists of the following subsections:

- "Conceptual Outline of an Assessment Ruleset" on page 28.
- "Encoding Assessment Rules Using **arules**" on page 30.
    - "An Example of **arules** Input" on page 30.
    - "Rules Files Usage and Syntax" on page 31.
    - "Composing Predicates in MVEL" on page 33.

Conceptual Outline of an Assessment Ruleset

The local assessment rules for your configuration are grouped into one or more *rulesets*. To understand how rulesets work, it is useful to view it in outline form:

- A ruleset consists of:
    - A *name*.
    - A *description.*
    - An *object filter*, which is the name of a JHOVE2 format module. Rules in the ruleset are evaluated against source units characterized by the format module. For example, a ruleset for XML files would have an object filter of **org.jhove2.module.format. xml.XmlModule.**

- One or more rules. Each rule consists of:
  - A *name*.
  - A *description*.
  - A *consequent*. This is the narrative description when the source unit conforms to the rule.
  - An *alternative*. This is the narrative description when the source unit does not conform to the rules.
  - One or *predicates*, which are statements about the source unit that evaluate to a Boolean true or false.
  - A quantifier, which is one of the following:
    - "Any of", meaning the source unit conforms when any predicate is true.
    - "All of", meaning the source unit conforms when all predicates are true.
    - "None of", meaning the source unit conforms when none of the predicates are true.

This outline omits the fact that a ruleset or rule can be *enabled* (it is processed normally) or *disabled* (it is ignored). This feature allows you to fine-tune assessment without major editing.

Suppose you want a ruleset that implements the following rules for XML files:

- The XML file is standalone. Narrative result: "yes" or "no".

- The XML file is acceptable: either the file has been validated, or its validation status is unknown (because, for example, the DTD is inaccessible) and the file is well-formed. Narrative result: "acceptable" or "not acceptable".

We can express these rules as follows:

- Ruleset
  - Name: XmlRuleSet
  - Object filter: org.jhove2.module.format.xml.XmlModule
  - Description: Rules for XML files
  - Rule
    - Name: XmlStandaloneRule
    - Description: Does XML declaration specify standalone status?
    - Consequent: yes
    - Alternative: no
    - Quantifier: All of
    - Predicates
      - The value of the object property **`xmlDeclaration.standalone`** is true.
  - Rule
    - Name: XmlAcceptableRule

- Description: Is the XML file acceptable?
- Consequent: acceptable
- Alternative: not acceptable
- Quantifier: Any of
- Predicates:
  - The value of the object property **valid** is true.
  - The value of the object property **valid** is undefined and the object **wellFormed** is true.

Encoding Assessment Rules Using arules

The Assessment Module expects assessment rules to be provided as Spring bean XML definitions in **config/spring/module/assess/jhove2-ruleset-xml-config.xml**. The format of this file is documented in the *JHOVE2 Assessment Module Specification*. To avoid the complications of directly maintaining this file, you can use the **arules** tool.

An Example of arules Input

With **arules**, each ruleset is defined in a single file using a simple syntax. Figure 18 show the **arules** file for the XmlRuleSet example.

Figure 18: *arules* ***Input for XmlRuleSet***

```
ruleset XmlRuleSet enabled org.jhove2.module.format.xml.XmlModule
desc Ruleset for XML module

rule XmlStandaloneRule enabled
desc Does XML Declaration specify standalone status?
cons Is Standalone
alt Is Not Standalone
quant all
pred xmlDeclaration.standalone == "yes"

rule XmlAcceptableRule enabled
desc Is the XML status acceptable?
cons Acceptable
alt Not Acceptable
quant any
pred valid.name() == "True"
pred (valid.name() == "Undetermined")
     && (wellFormed.name() == "True")
```

The syntax of a rules file is described in the next section. Assuming the input is in a file named **xmlrules** the Windows command for generating the equivalent XML is:

```
arules xmlrules > jhove2-ruleset-xml-config.xml
```

On Linux, OS X, and Solaris:

```
./arules.sh xmlrules > jhove2-ruleset-xml-config.xml
```

Note: Both commands assume that the working directory is the JHOVE2 home directory. If this directory is in your path, the Windows command works anywhere, and the Linux/OS X/Solaris command works anywhere with the **./** omitted.

When the resulting **jhove2-ruleset-xml-config.xml** is copied to **config/spring/module/assess** and jhove2 is run against an XML file, output will include text similar to

Figure 19: Assessment Output

```
    Module {AssessmentModule}:
     AssessmentResultSets:
      AssessmentResultSet:
       RuleSetName: XmlRuleSet
       RuleSetDescription: Ruleset for XML module
       ObjectFilter: org.jhove2.module.format.xml.XmlModule
       BooleanResult: false
       AssessmentResults:
        AssessmentResult:
         RuleName: XmlStandaloneRule
         RuleDescription: Does XML Declaration specify standalone status?
         BooleanResult: false
         NarrativeResult: Is Not Standalone
         AssessmentDetails: ALL_OF { xmlDeclaration.standalone == "yes" =>
false; }
        AssessmentResult:
         RuleName: XmlAcceptableRule
         RuleDescription: Is the XML status acceptable?
         BooleanResult: true
         NarrativeResult: Acceptable
         AssessmentDetails: ANY_OF { valid.name() == "True" =>
true;(valid.name() == "Undetermined") && (wellFormed.name() == "True") =>
false; }
```

Rules Files Usage and Syntax

The command line usage for **arules** on Windows is:

arules *file*...

On Linux, OS X, and Solaris:

```
./arules.sh file...
```

Note: Both commands assume that the working directory is the JHOVE2 home directory. If this directory is in your path, the Windows command works anywhere, and the Linux/OS X/Solaris command works anywhere with the **./** omitted.

Each file contains a single ruleset; **arules** combines them and writes generated XML to the standard output. The file consists of a series of statements that must begin in the first column; indented lines are continuations of the previous statement. The following statements are used:

ruleset ***name enabled filter***

Defines the beginning of a ruleset. Must be the first statement in the file. Arguments:

- ***name*** is the ruleset name. The name must be a valid XML ID, meaning that it can include letters, digits, "**_**", "**:**", "**.**" or "**-**" and the first character cannot be "**.**" or "**-**". The ruleset name must be different from that of any other ruleset or any rule.
- ***enabled*** is one of the keywords **enabled** or **disabled**, indicating whether the ruleset is applied.
- ***filter*** is the object filter: theURL that specifies the fully-qualified name of the JHOVE2 format module to which the ruleset is applied.

rule ***name enabled***

Defines the beginning of a rule, which is terminated by a following rule or the end of the file. Arguments:

- ***name*** is the rule name. It follows the same conventions as a ruleset name, and must also be unique.
- ***enabled*** is one of the keywords **enabled** or **disabled**, indicating whether the ruleset is applied.

desc ***description***

Specifies the description for a ruleset or rule. This description applies to the immediately preceding **ruleset** or **rule** statement. There must be exactly one description for each ruleset or rule.

`cons` ***`consequent`***

Specifies the rule consequent. This is the narrative description if the rule holds for the source unit. There must be exactly one consequent for each rule.

`alt` ***`alternative`***

Specifies the rule alternative. This is the narrative description if the rule does not hold for the source unit. There must be exactly one alternative for each rule.

`quant` ***`quantifier`***

Specifies the rule quantifier, which must be one of the following keywords:

- **any** specifies that the rule holds when any of the predicates are true for the source unit.
- **`all`** specifies that the rule holds when all the predicates are true for the source unit.
- **none** specifies that the rule holds when none of the predicates are true for the source unit.

There must be exactly one quantifier for each rule.

`pred` ***`predicate`***

Specifies a predicate, which is an MVEL expression with a Boolean value. There must be at least one predicate in each ruleset.

Note: The **`desc`**, **`cons`**, **`alt`**, **`quant`**, and **`pred`** statements that follow a **`rule`** statement can be in any order. The order used in Figure 18 is probably the easiest to understand, but is not mandatory.

Predicates are expressed in the MVEL expression language. The syntax for MVEL is substantially the same as for Java, with enhanced access for property values.

Composing Predicates in MVEL

MVEL syntax is similar to Java's. The most important difference is that property references are simplified and enhanced. This section provides an overview of MVEL as it is used in assessment predicates. For complete documentation of MVEL features, refer to the MVEL Language Guide at http://mvel.codehaus.org/Language+Guide+for+2.0.

One important extension is that MVEL comparison operators compare object values, not identities. For example we can use the following expression to directly examine the value of a String property:

```
xmlDeclaration.standalone == "yes"
```

In Java, we would have to use this expression (which also works in MVEL):

```
xmlDeclaration.standalone.equals("yes")
```

One important limitation of predicate expressions is that classes outside the **`java.lang.*`** packages cannot be used directly. (This is a limitation of JHOVE2, not of MVEL.) This is seen in Figure 18, where the **`valid`** and **`wellFormed`** properties are converted to strings before being examined.

```
valid.name() == "Undetermined"
```

Note: For enumerated types, use **`name()`**, which returns the string value of the constant name. This is preferable to **`toString()`**, which returns a human-readable value that is not guaranteed to be consistent.

This is necessary because these properties contain an enumerated type declared in **`org.jhove2.module.format`**. On the other hand, **`xmlDeclaration.standalone`** is of type **`String`**, and can be accessed directly.

For more examples, let's look at two classes in JHOVE2's unit testing suite. The first is **`MockModule`**, which simulates the functionality of a JHOVE2 format module. (Figure 20 shows a simplified declaration of **`MockModule`**, where the protected fields correspond to properties.) The other is **`MockReportable`**, which defines types used by **`MockModule`**. (Figure 21 shows a simplified declaration of **`MockReportable`**.)

Figure 20: MockModule Properties

```
public class MockModule extends AbstractModule {
    protected MockEnum mpEV;
    protected String mpString;
    protected long mpLong;
    protected boolean mpBoolean;
    protected MockReportable mpReportable;
    protected List<String> mpListString;
    protected Set<String> mpSetString;
    protected List<Reportable> mpListReportable;
    //Constructor, methods, and static fields omitted.
}
```

Figure 21: MockReportable

```
public class MockReportable extends AbstractReportable {

    public enum MockEnum {
        EV0, EV1, EV2, EV3, EV4
    }

    protected MockEnum cpEV;
    protected String cpString;
    protected long cpLong;
    protected boolean cpBoolean;
    protected MockReportable cpReportable;
    protected List<String> cpListString;
    //Constructor, methods, and static fields omitted.
}
```

These predicate expressions examine MockModule properties using idioms that are common to Java and MVEL:

```
mpBoolean
mpLong != 10
mpEv.name() == "EV0"
mpListString.size() == 4
```

String comparisons are enhanced by their simplified usage with comparison operators and also by a regular expression operator:

```
mpString == "Equality Now!"
mpString <= "U"
mpString ~= "[A-Za-z][A-Za-z0-9]*"
```

The **`contains`** operator searches strings and collections:

```
mpString contains "mock"
mpListString contains "mock turtle"
```

Running JHOVE2

The **JHOVE2**command line follows these patterns:

Windows:

```
jhove2 options arguments ...
```

Linux, OS X, and Solaris:

```
./jhove2.sh options arguments ...
```

Where ***options*** is zero or more of the options described below, and *arguments ...* is one or more file names, directory names, and URLs indicating remote files. (URLs can use the **http**, **https**, **ftp**, or **file** schemas.) JHOVE2 searches for unitary and aggregate source units within this list. For specifics, see "How JHOVE2 Identifies Source Units." on Page 39.

> Note: Both commands assume that the working directory is the JHOVE2 home directory. If this directory is in your path, the Windows command works anywhere, and the Linux/OS X/Solaris command works anywhere with the **./** omitted.

> Note: If an argument contains blanks, it must be surrounded by quote marks. On Windows, double quote marks (**"..."**) are required. On other platforms, use single quote marks (**'...'**) to avoid additional shell substitutions.

All options have a short form and a long form; the two forms are interchangeable. For example, the following three command lines are functionally equivalent to each other:

```
jhove2 –i –o output.txt jupiter.tif
jhove2 --show-identifiers –-output output.txt jupiter.tif
jhove2 –i –-output output.txt jupiter.tif
```

Options that take an argument (such as **–o** and **--output** in the above examples) must have a space between the option and the argument. Short form options that don't take an argument can be grouped together (**-ikT** instead of **–i –k –T**).

Command line options are divided into the following categories:

- "Output Options" on page 37.

- "Temporary File Options" on page 37.
- "I/O Buffer Options" on page 37.
- "Help Options" on page 38.

Output Options

-d *format*
--display *format*
    Display format. Choices are: **Text** (default), **JSON**, and **XML**.

-i
--show-identifiers
    Show property identifiers. These are unique formal identifiers, generated by JHOVE2.

-k
--calc-digests
    Calculate message digests (hash values) for each source unit. The distributed configuration uses CRC-32, MD5, and SHA-1 functions. To configure this feature, see "Configure Digest" on page 19.

-o *file*
--output *file*
    Send output to ***file***. By default, output goes to standard output.

Temporary File Options

-t *temp*
--temp *temp*
    Create temporary files in ***temp*** directory. By default, the temporary file directory is specified by the Java runtime.

-T
--save-temp-files
    Do not delete temporary files.

I/O Buffer Options

-b *size*
--buffer-size *size*
    Set ***size*** of I/O buffers in bytes. Default buffer size is 131,072 bytes (16 KiB).

-B *type*
--buffer-type *type*
Set buffer type to ***type***. Choices are **Direct** (default), **Indirect**, and **Mapped**.

Help Options

-h
--help
Display help message.

How JHOVE2 Identifies Source Units

To JHOVE2, a source unit is any entity that can be independently characterized. A source unit might be a single file, a byte stream within a file, or a collection of files.

Many source units are *aggregate*: they contain other source units. An aggregate source unit might be:

- A *container,* which contains an arbitrary collection of files that may or may not be related. Examples of containers are file system directories, ZIP or TAR archive files, and directories within archive files.
- A *file set,* the set of files on a given JHOVE2 command line. As with containers, the files in a file set may or may not be related.
- A *wrapper file,* which contains a set of related byte stream source units. Examples include TIFF files, which contains byte streams such as ICC color profiles or XMP metadata.
- *A clump*, which consists of a set of logically related files within a container or file set. A clump may or may not include all the files within a container or file set. For example, a GIS shapefile consists of a **`.shp`** file that is always accompanied by **`.shx`** and **`.dbf`** files, and may be accompanied by additional files.

  Note: Some clumps, such a shape files, are identified in part by the filename extensions of their constituent files. If these files are retrieved via a URL from a server that does not preserve extensions, then the clump cannot be identified.

Source units that don't contain other source units are *unitary*.

The user does not specify how source units are structured; JHOVE2 simply characterizes every unitary or aggregate source unit it can find. Note that individual files can be characterized as part of more than one aggregate source unit; in particular, the file sets for various clumps within a container or file set can overlap.