# EMMA Reference Manual

# EMMA Reference Manual

# Table of Contents

# List of Tables

# Chapter 1. Installation and System Requirements

**Supported JRE versions.** EMMA has been implemented to work in any J2SE runtime environment. For performance reasons, EMMA tools and runtime can benefit from (but do not require) J2SE APIs available in J2SE versions 1.3 and 1.4. EMMA command line tools, ANT tasks, and runtime have been tested in a variety of JREs from Sun Microsystem, IBM, and BEA.

**Supported ANT versions.** EMMA ANT tasks work with Apache ANT 1.4.1 and later versions.

**External library dependencies.** EMMA has no external Java or native library dependencies.

**Operating system.** EMMA is a pure Java application and does not use JVMPI or other profiling interfaces requiring native libraries. It should provide identical functionality on any operating system supported by J2SE v1.2+[1].

**EMMA distribution.** EMMA is contained in two Java class archives (found in the `lib` subdirectory of EMMA distribution):

`emma.jar`
: Contains the implementation of EMMA core components command line tools, and *EMMA runtime classes* (EMMA classes that are needed by Java application code that has been instrumented for coverage).

`emma_ant.jar`
: Contains the implementation of EMMA ANT tasks (this archive depends on `emma.jar` and does not overlap with it in content).

**General installation considerations.** "Installing" EMMA simply implies making `emma.jar` and `emma_ant.jar` available to the Java Runtime Environment (JRE) and Apache ANT runtime, as appropriate.

There are two distinct runtime cases for EMMA:

a. Execution of an EMMA command line tool or ANT task.

b. Execution of some Java code that has been instrumented for coverage. Note that every EMMA-instrumented class becomes dependent on EMMA runtime classes (contained in `emma.jar`).

**Installing EMMA core/runtime library.** Accordingly, to run EMMA command line tools or EMMA-instrumented applications you need to add `emma.jar` to the appropriate JRE classpath. You can do it either via the `-cp` JVM option or by adding `emma.jar` as an *installed JRE extension* (by copying `emma.jar` to `lib/ext` subdirectory of your JRE or by setting the `java.ext.dirs` JVM system property to include the `lib` subdirectory of EMMA distribution. See Sun's documentation on Installed Extentions [http://java.sun.com/docs/books/tutorial/ext/basics/install.html] and `java.ext.dirs` [http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html#installed] property

[1] In a Sun Microsystems-compatible JRE prior to version 1.3 the runtime coverage data is dumped (in the offline coverage mode) only when the JVM is terminated via `Ctrl-C` or an equivalent signal.

for more details).

## The JRE extension option is preferred

It is *highly recommended* to install `emma.jar` as a JRE extension. This simplifies EMMA usage with application containers (IBM Websphere, BEA WebLogic, etc). Furthemore, installed JRE extensions are *trusted* by default: the instrumented application classes will automatically have the necessary runtime permissions for dumping coverage data files. Note that the JRE used by an application container may not necessarily be the same one you use from command line or ANT.

**Setting up EMMA ANT tasks.** To run EMMA ANT tasks, one additional configuration step inside `build.xml` is required:

```
<!-- EMMA distribution directory: -->
<property name='emma.dir' value='your EMMA install location' />

<path id='emma.lib' >
  <fileset dir='${emma.dir}' includes='lib/*.jar' />
</path>

<taskdef resource='emma_ant.properties' classpathref='emma.lib' />
```

## EMMA lib path

The `build.xml` snippet shown above defines a path element with `emma.lib` reference id. Although this is not strictly necessary (the `<classpath>` element nested inside the `<taskdef>` could have worked just as well), such a path element usually comes in handy elsewhere in the `build.xml`.

# Chapter 2. EMMA Tool Reference

## 1. Overview

EMMA functionality is implemented by a set of components responsible for class instrumentation, metadata and coverage data processing, and coverage report generation. Each component has adapters that expose its functionality as an ANT task and a command line tool. To reflect this design, each of the following reference subsections starts by detailing the overall functionality of a given component, followed by its concrete ANT and command line usage.

The following table summarizes EMMA ANT tasks and their equivalent command line tools:

**Table 2.1. EMMA ANT tasks and command line tools**

| ANT task | Command line tool | Functionality | Output |
|---|---|---|---|
| *on-the-fly processing mode* | | | |
| <**emmajava**> | **emmarun** | Executes a standalone Java application in EMMA instrumenting classloader without a separate instrumentation phase. | One or several coverage reports and (optionally) a coverage session raw data file. |
| *offline processing mode* | | | |
| <**instr**> subtask | **instr** command | Instruments a set of classes in a given list of directories and/or archives and output. | Instrumented classes and archives and a coverage metadata file. |
| <**report**> subtask | **report** command | Combines class metadata and coverage runtime data to produce coverage reports. | One or several coverage reports (plain text, HTML, XML). |
| <**merge**> subtask | **merge** command | Merges and compacts several metadata, coverage, or session data files. | A single merged coverage session data file. |

**Subtasks and commands.** For reasons that include modularity and consistency of common option behavior EMMA offline coverage tools are used as subtasks of the "umbrella" <**emma**> ANT task:

```
<emma ...>
  <merge ...>
    ...
  </merge>
  <report ...>
    ...
  </report>
</emma>
```

or as subcommands of the "umbrella" **emma** command line command:

```
>java emma instr -ip out/classes/ ...
>java emma report -in coverage.em,coverage.ec ...
```

Note that in the ANT case, <**emma**> can contain an arbitrary sequence of subtasks (including mutliple subtasks of the same kind, which are executed in the exact sequence as they are specified). In addition to the already mentioned advantages, this allows entire blocks of EMMA tool invocations to be enabled/disabled and configured using attributes of the parent <**emma**> ANT task.

**EMMA properties cascade.** Some aspects of EMMA tool/task behavior can be modified using properties which can be set in a variety of ways: via JVM system properties, via an external file, via ANT (sub)task attributes or command line tool options. The task-subtask organization helps with this EMMA configuration as well. The general rule of thumb here is that properties within a more specific context (an ANT subtask vs <**emma**> parent task, a tool setting vs a global JVM property, etc) inherit their values from the surrounding contexts, but can also override them. For complete details see EMMA property lookup order. [36]

In the remainder of this manual, the same tool will be frequently referred to as <***ANT_tool_name***>/***command_line_tool_name***.

**Overall EMMA processing sequence.** The general sequence of steps when using EMMA depends on your chosen coverage mode:

- When using EMMA's on-the-fly coverage mode, there is little to do in addition to how you would normally run your application or testsuite: you use <**emmajava**>/**emmarun** as the new startup class and optionally tell it where to place the coverage report(s)[1].
- When using EMMA's offline coverage mode, the general sequence of tool/task invocations is:
  1. In one or several passes, use <**instr**>/**instr** to instrument class directories and archive files.
  2. Execute your application or test suite using the instrumented classes (one or several runs).
  3. Optionally, merge and compact all *metadata* and *runtime coverage* files using <**merge**>/**merge**.
  4. In one or several passes, use <**report**>/**report** to combine class metadata with the desired runtime coverage data profile(s) and generate the desired coverage report(s).

**EMMA data files are untyped.** EMMA tools use binary data files for storing instrumentation and runtime coverage results. EMMA files are not typed: they do not require a particular name or extension. Furthermore, each data file is a mini-database that acts as an envelope for an arbitrary sequence of metadata and/or runtime coverage data dumps. It is up to the user how to structure their work with EMMA: either accumulate everything in a single file or use a dedicated file for every tool. The only restriction is that EMMA files can only grow (once new data is merged in, it cannot be removed).

# 2. <emmajava>/emmarun

<**emmajava**>/**emmarun** — instrumented application runner (on-the-fly instrumentation mode).

## 2.1. Description

[1]Usually, <**emmajava**> (**emmarun**) runs correspond to completely independent application/testsuite runs. However, its `-raw` option could be used to retain coverage session data files across different runs, to be processed later by tools like <**report**>/**report** and <**merge**>/**merge**.

**<emmajava>/emmarun** pairs an advanced custom classloader with a combination of core internal components of **<instr>/instr** and **<report>/report** to form a unique convenience tool in EMMA kit: an application runner that instruments classes *on the fly* and generates coverage reports without any need for a separate build or any intermediate work files alltogether.

**<emmajava>/emmarun** convenience is especially apparent in command line mode, because its option names intentionally mimick the familiar **java** options: a Java application launch command line could be made coverage-enabled by inserting a single new word (**emmarun**) into the command line. Similarly, the ANT version of this tool is an extension of ANT's standard <java> task. **<emmajava>/emmarun** can execute as little as a single Java class or as much as a complex Swing application made up of hundreds of classes, all with equally small instrumentation overhead.

Thinking of **<emmajava>/emmarun** as a combination of EMMA class instrumentor and report generator is a good way to remember its ANT attributes and command line options: in the reference sections that follow most of them are documented as identical to their namesakes in **<instr>/instr** and **<report>/report**.

**What gets instrumented.** The default **<emmajava>/emmarun** behavior is to instrument only the classes that are loaded by the JVM for the running application. The resulting report will not even mention classes that were never loaded and which potentially decrease your coverage percentages. If your objective is to get a complete coverage report (as you would from an offline combination of **<instr>/instr** and **<report>/report**) you should use the option for a full classpath scan, possibly in combination with some coverage filters.

**Compatibility.** At runtime, **<emmajava>/emmarun**'s instrumenting classloader installs itself as the application classloader, bypassing the standard system classloader. It uses a smart class delegation strategy, whereby it automatically detects JRE core and extension classes without having to filter by class names (the frequently used, but inadequate, delegation strategy). Coupled with full support for `Class-Path` manifest entries and `-jar` option, the resulting EMMA runtime will correctly run most standalone Java programs. However, certain cases are exceptions:

- Java code referencing `java.lang.ClassLoader.getSystemClassLoader()` (either directly or via `ClassLoader.findSystemClass()` and related methods) instead of using the current or thread context loaders will bypass **<emmajava>/emmarun** classloader, causing subsequent classes to be loaded at the wrong node of the classloader hierarchy. Such coding patterns should really be considered bugs and are not supported. Such code could be patched up on the fly during instrumentation, but a reliable solution is expensive in terms of processing. Switching to offline instrumentation is an easy workaround.

- Java applications designed around their own custom classloaders and classpaths most likely will not work with **<emmajava>/emmarun** (at best, they will run fine but coverage instrumention will not occur). Application containers (Apache tomcat, BEA Weblogic, IBM Websphere, etc) are the common case here. Again, switching to offline instrumentation is an easy alternative.

**<emmajava> is always forked.** ANT's in-process classloading model is not sufficiently JRE-compatible. ANT's class delegation in the standard **<java>** task in `fork='false'` (in-process) mode is based on name matching (with a hardcoded set of name filters) and inevitably fails for applications that depend on non-standard JRE extensions. To support EMMA deployment as a JRE extension **<emmajava>** always forces `fork='true'` to ensure correct execution (unless its `enabled` attribute makes it a pass-through).

**Internal EMMA properties that affect classloading and class instrumentation.** Several EMMA property settings affect instrumentation and classloading behavior done by **<emmajava>/emmarun**:

- instr.do_suid.compensation []

- instr.exclude_synthetic_methods []
- instr.exclude_bridge_methods []

Changing the default classloading behavior should be done by experienced Java users only. Most of instrumentation-related properties should normally be left with their default values. instr.do_suid.compensation [] can be set to `false` to gain extra instrumentation processing speed when runtime execution does not involve class de-serialization from existing files or serialization across JVMs.

# 2.2. ANT usage

<**emmajava**> task is an implicit combination of <**instr**> and <**report**> tasks and most of its attributes and nested elements are the same as for those two tasks combined.

## Parameters specified as attributes

| Attribute | Description | Required |
|---|---|---|
| [stock ANT <java> task attributes [http://ant.apache.org/manual/CoreTasks/java.html]] | | |
| [common EMMA task attributes] | | No |
| `libclasspath` | A *path-like structure* containing EMMA core (`emma.jar`). | Yes, unless EMMA is installed as the JRE extension |
| `libclasspathref` | Same as `libclasspath`, but given as a reference to a path defined elsewhere. | Yes, unless EMMA is installed as the JRE extension |
| `fullmetadata` | Indicates whether the entire classpath should be added to the coverage metadata (default: `false`). | No |
| `dumpsessiondata` | Indicates whether the session (metadata+coverage) data resulting from this coverage run should be dumped to a file. Useful for post-run coverage report generation (default: `false`). | No |
| `sessiondatafile`, `outfile` | If `dumpsessiondata='true'`, overrides the location to store session data (default: file `coverage.es` in the current directory). Ignored otherwise. | No |
| `merge` | Indicates whether the session data should be merged into the destination `sessiondatafile`, if any (default: `true`). Any existing data is clobbered otherwise. | No |
| `filter` | Adds a coverage filter. See Section 6.2, “Coverage filters” [29] for general description of EMMA coverage filters and Section 6.2.1, “filter syntax: ANT” [30] specifically for ANT syntax. This attribute plays a role equivalent to the same attribute of <**instr**>. | No |
| `sourcepath` | A *path-like structure* that can be used to point the HTML report generator in | No |

| Attribute | Description | Required |
|---|---|---|
| | **emmarun** to the location of your Java source files. It is interpreted as a list of directories (separated by the OS-specific classpath separator or comma) containing `.java` source files. The local path names within each directory should reflect class package names. This attribute is equivalent to the same attribute of <**report**> task. | |
| `sourcepathref` | Same as `sourcepath`, but given as a reference to a path defined elsewhere. | No |
| `units` | Equivalent to the same attribute of <**report**>. | No |
| `depth` | Equivalent to the same attribute of <**report**>. | No |
| `columns` | Equivalent to the same attribute of <**report**>. | No |
| `sort` | Equivalent to the same attribute of <**report**>. | No |
| `metrics` | Equivalent to the same attribute of <**report**>. | No |
| `encoding` | Equivalent to the same attribute of <**report**>. | No |

## Parameters specified as nested elements

| Element | Description | Required |
|---|---|---|
| [stock ANT <java> task nested elements [http://ant.apache.org/manual/CoreTasks/java.html]] | | |
| [common EMMA task nested elements] | | No |
| `<filter>` | Adds a coverage filter. See Section 6.2, “Coverage filters” [29] for general description of EMMA coverage filters and Section 6.2.1, “filter syntax: ANT” [30] specifically for ANT syntax. This nested element plays a role equivalent to the same element of <**instr**>. | No |
| `<sourcepath>` | A *path-like structure* that can be used to point the HTML report generator in <**emmajava**>/**emmarun** to the location of your Java source files. This element is equivalent to the same nested element of <**report**> task. | No |
| `<txt>` | Equivalent to the same nested element of <**report**>. | No |
| `<html>` | Equivalent to the same nested element of <**report**>. | No (`<txt>` implied by default) |
| `<xml>` | Equivalent to the same nested element of <**report**>. | No (`<txt>` implied by default) |

**`<txt>`, `<html>`, and `<xml>` nested elements.** These nested elements create plain text, HTML, and XML coverage reports, respectively. If none is specified, the plain text report is implied (at most one configurator of any given report type can be nested inside a given <**emmajava**> call). Configuration of these elements is described in the equivalent section of <**report**> task reference page.

## Examples

- Generate plain text and HTML reports with the default parameters:

```
<emmajava enabled="${emma.enabled}" libclasspathref="emma.lib"
          filter="${emma.filter}" sourcepath="${src.dir}"
          classname="Main" classpathref="run.classpath"
>
  <!-- since this task is an extension of stock <java>, normal <java>
       options are still available: -->
  <arg value="someargvalue" />

  <txt outfile="${coverage.dir}/coverage.txt" />
  <html outfile="${coverage.dir}/coverage.html"  />
</emmajava>
```

- Do a full metadata scan (of `run.classpath`), generate an HTML report with some customization, use a <**dirset**> to set the sourcepath:

```
<emmajava enabled="${emma.enabled}" libclasspathref="emma.lib"
          fullmetadata="yes"
          classname="Main" classpathref="run.classpath"
>
  <sourcepath>
    <dirset dir="${basedir}" >
      <include name="**/src" />
    </dirset>
  </sourcepath>

  <html outfile="${coverage.dir}/index.html"
        columns="name, method, line"
        sort="+line, +name"
        metrics="line:80"
  />
</emmajava>
```

- Don't generate any reports, just dump the raw coverage session data for now:

```
<emmajava enabled="${emma.enabled}" libclasspathref="emma.lib"
          fullmetadata="yes" dumpsessiondata="yes"
          classname="Main"
          classpathref="run.classpath"
/>
```

# 2.3. Command line usage

## Synopsis

`java emmarun` [*(1) EMMA options*] -cp *`classpath`*... *`class`* [*`args`*...]
`java emmarun` [*(1) EMMA options*] -jar *`jarfile`* [*`args`*...]
(1) [-f] [-ix *`filter patterns`*...] [-r *`report types`*...] [-sp *`sourcepath`*...] [-raw] [-out *`session data file`*] [-merge *`boolean`*] [`common options`]

alternative form:
`java emma run` {*`same as above`*...}

## Options

[common command line options]

-f, -fullmetadata
: This flag indicates whether the entire classpath (-cp) should be added to the coverage metadata (default: false). Without this flag, only the classes explicitly loaded by the JVM will be in the instrumentation set.

-ix, -filter *filter patterns*...
: This *repeatable* option adds a coverage filter. See Section 6.2, “Coverage filters” [29] for general description of EMMA coverage filters and Section 6.2.2, “filter syntax: command line” [31] specifically for command line syntax. It is equivalent to the same option of **instr** tool.

-r, -report (txt|html|xml)...
: This *repeatable* option selects the type of coverage report(s) to generate (default: txt). It is equivalent to the same option of **report** tool.

-sp, -sourcepath *list of source directories*...
: This *repeatable* option can be used to point the HTML report generator in <**emmajava**>/**emmarun** to the location of your Java source files. Equivalent to the same option of **report** tool.

-raw, -sessiondata
: This flag indicates whether the session (metadata+coverage) data resulting from this coverage run should be dumped to a file. Useful for post-run coverage report generation (default: false).

-out, -outfile *session data file*
: If -raw flag is set, this option overrides the location to store session data (default: file coverage.es in the current directory). Ignored otherwise.

-merge (y[es]|n[o])
: Indicates whether the session data should be merged into the destination outfile, if any (default: true). Any existing data is clobbered otherwise.

report generation options...
: Unlike its ANT equivalent, **emmarun** command line tool does not have dedicated options for controlling coverage report generation. If necessary, they can be set using generic -D, -properties, and other mechanisms.

  So, for example, to change the default location of the HTML report you would override the report.out.file [] property:

```
>java emmarun -Dreport.html.out.file=mycoveragedir/myfile.html ...
```

  (report.html.out.file can be abbreviated to report.out.file if the command generates a single report type)

## Examples

- Run an application and generate plain text and XML reports with default parameters:

```
>java emmarun -r txt,xml -jar SwingSet2.jar
```

- Run an application and generate an HTML report with some customization and linking to the

application source code:

```
>java emmarun -r html -Dreport.columns=name,method,line -sp src/ -jar SwingSet2.
```

- Run an application and don't generate any reports, just dump the raw coverage session data:

```
>java emmarun -raw -jar SwingSet2.jar
```

## Diagnostics

The default EMMA command line tool behavior is not to use `System.exit()` on exit unless an explicit `-exit` option is specified. If that is done, the error codes returned via `System.exit()` are as follows:

| | |
|---|---|
| `0` | Successful completion. |
| `1` | Failure due to incorrect option usage. This error code is also returned when command line usage (`-h`) is requested explicitly. |
| `2` | All other failures. |

# 3. <instr>/instr

**<instr>/instr** — offline class instrumentor.

# 3.1. Description

**<instr>/instr** is EMMA's offline class instrumentor. It adds bytecode instrumentation to all classes found in an instrumentation path that also pass through user-provided coverage filters. Additionally, it produces the *class metadata* file necessary for associating *runtime coverage data* with the original class definitions during coverage report generation.

**Instrumentation path.** Note that the classes to be instrumented are taken from a path element that is exactly like the kind taken by normal JDK tools and ANT tasks: it is a list of directories (containing `.class` files) and `.jar`/`.zip` archives (specified as an arbitrary number of `instrpath` (`-ip`) options). All non-existent or duplicate entries in the instrumentation path are effectively ignored during processing.

### `Class-Path` manifest entries

Note that **<instr>/instr** processes `Class-Path` entries in the manifests of class archives that it encounters. This is by design and is the desirable behavior (especially in the overwrite and fullcopy output modes), but care needs to be taken to avoid processing unintended implicit path segments.

**Output modes.** To accomodate different build and testsuite designs **<instr>/instr** has three different modes for how it outputs instrumented classes:

`copy`
: In this mode, all instrumented classes are output to a single destination directory, regardless of whether the source classes came from a directory or an archive. Furthemore, only the classes and

archive entries that are in the *instrumentation set* are written out. The idea here is to process just the necessary classes in as few disk I/O operations as possible.

For coverage-enabled application/testsuite runs the destination directory needs to be placed in the classpath ahead of the original classes. If this is inconvenient (say, because you need to package classes in archives before you can run), the `overwrite` mode might be a better option.

`overwrite`

This mode is similar to `copy`, except it overwrites the original class and archive files. This is ideal as a pre-packaging step turned on only when coverage-enabled application/testsuite runs are needed. Its advantage over the `copy` mode is that it can do jar-to-jar processing and eliminates the need to prepend a special output directory to the classpath. Its disadvantage is the extra CPU and disk I/O times needed to duplicate archive entries that are not being instrumented[2].

`fullcopy`

This mode is a hybrid between `copy` and `overwrite`. It offers the convenience of mixed individual class file and jar-to-jar processing without having to overwrite the original content. In this mode, the destination directory is split into two subdirectories, `classes` and `lib`, which accept instrumented class files and instrumented class archives, respectively.

Note that because in this mode **<instr>/instr** has to copy the most content (both files and archive entries that are not being instrumented), this mode could be the slowest of the three. The exact performance behavior depends on the relative speeds of your CPU and I/O subsystems and on the relative content mixes between class files and class archives in the input.

By design, in all output modes that can do jar-to-jar processing **<instr>/instr** does not compress the instrumented zip entries in the output archives. This saves CPU time needed for doing compression, usually at an acceptable cost in the increased disk space taken by the affected archive files.

**Class coverage metadata.** An important byproduct of class instrumentation is *class metadata*. As described in more detail elsewhere, EMMA coverage is based on instrumenting basic bytecode blocks. Every instrumentation run outputs a compact representation of data necessary to associate coverage of an individual basic block with its parent method and class as well as the original Java source lines that map to this basic basic (there is a metadata entry for every class in the instrumentation class set). Class metadata from each offline instrumentation run needs to be saved in a disk file, because it will be required for coverage stats computation and coverage report generation.

Note that when **<instr>/instr** writes metadata into a file, it will by default *merge* incoming metadata into the existing data in the destination file (if it exists). This behavior is also necessary to support *incremental instrumentation*, as described shortly.

**Class metadata merging.** To avoid any ambiguities, it is necessary to completely specify how **<instr>/instr** resolves duplicate data during instrumentation path processing:

1. During a given *instrumentation run*, all directory and archive entries in the instrumentation path are processed left-to-right. All duplicates (defined as entries with the same canonical file pathnames) are skipped. As noted above, valid `Class-Path` manifest entries are also processed, in the order they are discovered. This sequence is thus the same as it would be for classloading lookup if the instrumentation path were used as a classpath.

2. It is still possible that during the same instrumentation run identical class names are encountered (e.g., if the same class name shows up in differently named archives). To stay consistent with classloading lookup rules (the first class definition in a classpath wins), **<instr>/instr** will instrument and emit metadata only for the first class definition it encounters.

[2] ZIP file format does not allow incremental updates. For every class archive in the instrumentation path, to replace the selected entries with their instrumented version EMMA has to create a temporary archive that eventually replaces the original. This implies that all zip entries not being instrumented must be copied from one archive file to the other.

3. Finally, it is possible that multiple metadata entries for idential class names are brought together when metadata from *independent* instrumentation runs is merged together. The rule here is that the *last* metadata entry wins. The last entry is defined as either the last one merged into a given metadata file or (in the case of multiple files) contained in the last file in a given input file set.

The last point is best illustrated with an example. If both `coverageA.em` and `coverageB.em` contain metadata for class `MyClass`:

```
>java emma instr -ip ... -d coverageA.em ...
>java emma instr -ip ... -d coverageB.em ...
```

then the definition in `coverageB.em` wins in all these cases:

```
>java emma report -in coverageA.em -in coverageB.em ...
>java emma report -in coverageA.em,coverageB.em ...
>java emma merge -in coverageA.em -in coverageB.em ...
```

Similar rules apply to EMMA ANT tasks.

**Incremental instrumentation and metadata merging.** As is common knowledge, when working with **javac**, either from command line or via ANT's **<javac>** task, only the classes that were modified since the last compilation get re-compiled. This is incredibly convenient for an individual developer, as it makes a complex product build incremental: small changes to the source code results in quick incremental compiles. This is indispensable for the "code some—test some—repeat" approach to software development.

EMMA can be used such that it fully preserves the incremental nature of a build. The key to this is how class metadata is merged when it is output to the same file. Suppose a developer executes the following actions (EMMA command line tools are used here for compactness, but the same is possible with an EMMA-enhanced ANT build):

```
>javac -g -d classes src/my/java/sources/*.java
>java emma instr -ip classes ...
... edit some sources ...
>javac -g -d classes src/my/java/sources/*.java
... only the changed source files get re-compiled ...
>java emma instr -ip classes ...
... only the re-compiled class files get re-instrumented ...
```

In this case **<instr>**/**instr** was either in `copy` or in `overwrite` mode and it implicitly used the same default coverage metadata repository file, `coverage.em`, for each instrumentation run. In the `copy` mode, **<instr>**/**instr** instruments only the class files whose instrumented versions in the output directory are older than their **javac**-produced original versions. In the `overwrite` mode case, **<instr>**/**instr** will instrument (and overwrite) only the classes that are not already instrumented (because those would be the classes recently recompiled by **javac**). All later metadata entries written to `coverage.em` override any earlier definitions and it all works out correctly (and very fast).

Because the metadata is always up-to-date in this scenario, the developer can run his/her tests and look at coverage stats at any time he/she runs the tests, without doing an expensive rebuild of the entire project.

## Runtime coverage data merging

Note that the rules for merging *runtime coverage data* are different: the data from different

coverage runs is assumed to correspond to the same class definitions (in most cases EMMA will abort with an error if it detects a mismatch). Basic block coverage is merged such that the final coverage profile is a *union* of all merged profiles.

The following table summarizes the major differences between **<instr>**/**instr** output modes:

**Table 2.2. <instr>/instr output mode summary**

| Mode | Supports jar-to-jar processing | Supports incremental instrumentation | Output behavior |
|---|---|---|---|
| `copy` | No | Yes | All instrumented classes are written to a single destination directory (only instrumented entities are written out), regardless of whether they come from class files or class archives. |
| `overwrite` | Yes | Yes | Instrumented (and only instrumented) classes are overwritten in-place. Instrumented (and only instrumented) archive entries are updated in their archives. |
| `fullcopy` | Yes | No | All (instrumented or not) class files are written to a `classes` subdirectory of the destination directory. All (instrumented or not) class archives are written out to a `lib` subdirectory of the destination directory. |

**Internal EMMA properties that affect class instrumentation.** Several property settings affect **<instr>**/**instr** behavior:

- instr.do_suid.compensation []
- instr.exclude_synthetic_methods []
- instr.exclude_bridge_methods []

Most of these should normally be left with their default values. instr.do_suid.compensation [] can be set to `false` to gain extra instrumentation processing speed when runtime execution does not involve class de-serialization from existing files or serialization across JVMs.

# 3.2. ANT usage

## Parameters specified as attributes

| Attribute | Description | Required |
|---|---|---|
| [`common EMMA task attributes`] | | No |
| `instrpath` | A *path-like structure* specifying the instrumentation path to use. | Either this attribute, or `instrpathref` attribute, or at |

| Attribute | Description | Required |
|---|---|---|
| | | least one nested `<instrpath>` element must be present. |
| `instrpathref` | Same as `instrpath`, but given as a reference to a path defined elsewhere. | Either this attribute, or `instrpath` attribute, or at least one nested `<instrpath>` element must be present. |
| `destdir`, `outdir` | The location to store instrumented class files (in `fullcopy` mode instrumented classes are stored in `destdir/classes` and instrumented archives are stored in `destdir/lib` subdirectories, respectively). Ignored if `mode='overwrite'`. | Yes, unless `mode='overwrite'` |
| `metadatafile`, `outfile` | The location to store class coverage metadata (default: file `coverage.em` in the current directory). Neither particular file name nor extension are required. | No |
| `merge` | Indicates whether the metadata should be merged into the destination `metadatafile`, if any (default: `true`). Any existing data is clobbered otherwise. | No |
| `mode` | Specifies the instrumentation output mode. Valid values for this property are:<br>• `copy` (default): copy only instrumented class files and archive entries into `destdir` directory.<br>• `overwrite`: overwrite input class files and archives.<br>• `fullcopy`: copy all (instrumented or not) class files and archives to `destdir/classes` and `destdir/lib`, respectively.<br>See Section 3.1, “Description” [10] for more details. | No |
| `filter` | Adds an instrumentation filter. See Section 6.2, “Coverage filters” [29] for general description of EMMA's instrumentation filters and Section 6.2.1, “filter syntax: ANT” [30] specifically for ANT syntax. | No |

## Parameters specified as nested elements

| Element | Description | Required |
|---|---|---|
| [common EMMA task nested elements] | | No |
| <instrpath> | A *path-like structure* that specifies the instrumentation path to use. | Either instrpath attribute, or instrpathref attribute, or at least one nested <instrpath> element must be present. |
| <filter> | Adds an instrumentation filter. See Section 6.2, “Coverage filters” [29] for general description of EMMA's instrumentation filters and Section 6.2.1, “filter syntax: ANT” [30] specifically for ANT syntax. | No |

**<instrpath> nested elements.** <instrpath> is a *path-like structure* [http://ant.apache.org/manual/using.html#path] used to select class files and archives to be processed for instrumentation. If a duplicate class name is encountered during a single instrumentation pass, only the first class definition will be added to the *class metadata* emitted during this instrumentation path. See Class metadata merging. [11] for more details.

## Examples

- In-place instrument a certain subset of already compiled classes using overwrite mode and several coverage filters:

```
<emma enabled="${emma.enabled}" >
  <instr instrpathref="${out.dir}/classes"
         mode="overwrite"
  >
    <!-- always exclude every class with a "Test" in the name: -->
    <filter excludes="*Test*" />
    <!-- don't instrument everything in "${out.dir}/classes",
         only the stuff I am working on now: -->
    <filter file="myincludes.txt" />
    <!-- additional ANT command line hook: -->
    <filter value="${emma.filter}" />
  </instr>
</emma>
```

- Don't overwrite compiled classes that later need to go into official release jars (stay in copy mode). However, use incremental instrumentation for fast personal testing:

```
<emma enabled="${emma.enabled}" >
  <instr instrpathref="${out.dir}/classes"
         outdir="${out.dir}/classes-instrumented"
         merge="yes"
         filter="${emma.filter}"
  />
</emma>
```

- Take all jars already produced by the product build and make test (coverage-enabled) copies of them:

```
<emma enabled="${emma.enabled}" >
  <instr mode="fullcopy"
         outdir="${out.instr.dir}"
         merge="no"
         filter="${emma.filter}"
  >
    <instrpath>
      <fileset dir="${out.dir}" includes="**/*.jar" />
    </instrpath>
  </instr>
</emma>
```

# 3.3. Command line usage

## Synopsis

`java emma instr` {-ip *`instrumentation path`*...} [-d *`directory`*] [-out *`metadata file`*] [-merge *`boolean`*] [-m *`output mode`*] [-ix *`filter patterns`*...] [`common options`]

## Options

[`common command line options`]

`-ip, -cp, -instrpath` *`instrumentation path`*...
: This *repeatable* option sets the instrumentation path to use. Besides the OS-specific separator character, individual path segments can also be separated with commas.

`-d, -dir, -outdir` *`directory`*
: The location to store instrumented class files (in `fullcopy` mode instrumented classes are stored in `destdir/classes` and instrumented archives are stored in `destdir/lib` subdirectories, respectively). Ignored if `-m` is `overwrite`.

`-out, -outfile` *`metadata file`*
: The location to store class coverage metadata (default: file `coverage.em` in the current directory). Neither particular file name nor extension are required.

`-merge (y[es]|n[o])`
: This flag indicates whether the metadata should be merged into the destination `-out` file, if any (default: `true`). Any existing data is clobbered otherwise.

`-m, -outmode (copy|overwrite|fullcopy)`
: Specifies the instrumentation output mode. Valid values for this property are:

  - `copy` (default): copy only instrumented class files and archive entries into `destdir` directory.
  - `overwrite`: overwrite input class files and archives.
  - `fullcopy`: copy all (instrumented or not) class files and archives to `destdir/classes` and `destdir/lib`, respectively.

  See Section 3.1, “Description” [10] for more details.

`-ix, -filter` *`filter patterns…`*
: This *repeatable* option adds an instrumentation filter. See Section 6.2, “Coverage filters” [29] for general description of EMMA coverage filters and Section 6.2.2, “filter syntax: command line” [31] specifically for command line syntax.

## Examples

- In-place instrument a certain subset of already compiled classes using `overwrite` mode and several coverage filters:

```
>java emma instr -m overwrite -ip out/classes -ix -*Test* -ix @myfilters.txt
```

- Don't overwrite compiled classes that later need to go into official release jars (stay in `copy` mode). However, use incremental instrumentation for fast personal testing:

```
>java emma instr -merge y -ip out/classes -d out/classes-instrumented
```

- Take all jars already produced by the product build and make test (coverage-enabled) copies of them:

```
>java emma instr -m fullcopy -merge no -ip out/Product.jar -d out-instrumented/
```

## Diagnostics

The default EMMA command line tool behavior is not to use `System.exit()` on exit unless an explicit `-exit` option is specified. If that is done, the error codes returned via `System.exit()` are as follows:

| | |
|---|---|
| `0` | Successful completion. |
| `1` | Failure due to incorrect option usage. This error code is also returned when command line usage (`-h`) is requested explicitly. |
| `2` | All other failures. |

# 4. <report>/report

**<report>/report** — offline coverage report generator.

# 4.1. Description

**<report>/report** is EMMA's offline coverage report generator. It reads in an arbitrary number of data files containing *class metadata* and *runtime coverage data* and generates one or several coverage reports of desired types. Several aspects of coverage reporting (detail level, column order, column sorting, coverage metrics failure markup, etc) can be customized for a given report type.

**What is reported on.** Each invocation of **<report>/report** requires a set of input *metadata* and *runtime coverage data* files. EMMA coverage stats are derived exclusively from the classes that appear in the

combined class metadata as represented by this input. To put it differently, a coverage report can reflect as much as the state of the entire product codebase or as little as one Java package or API being worked on by a given developer at the moment.

**Report depth.** To understand EMMA's approach to generating coverage reports, the following paradigm should be kept in mind:

- a given coverage report covers all entities in the *instrumentation set*, referred to as `all classes` in the reports
- `all classes` entity contains Java packages
- [for classes compiled with full debug info] Java packages contain Java source files
-
  - [for classes compiled with full debug info] Java source files contain Java classes (in general, more than one)
  - [for classes compiled without full debug info] Java packages contain Java classes
- Java classes contain methods (which, in turn, could be broken down into basic blocks)

(The reason EMMA makes a distinction between classes with and without full debug info is that without the `SourceFile` attribute in all input classes it is in general impossible to make the association between classes and their source files and that in turn impacts how metrics like *line coverage* are rolled up. The above hierarchy is easier to understand if you realize that without the full debug info the source file hierarchy level is absent.)

Correspondingly, **<report>/report** calculates and presents coverage metrics in a way that allows for drilling down into data in a top-down fashion, starting with `all classes` and going all the way to the level of individual methods and source lines (in the HTML report). Coverage metrics are rolled up at the levels of individual methods, classes, source files, packages, and for the entire instrumentation set (`all classes`). The concept of "report depth" represents how deep you are in this hierarchy.

Different report types produced by **<report>/report** differ in how they reflect this data hierarchy:

- The plain text report is a low-overhead report type for quick coverage summary viewing and processing by tools like **grep** and **Perl**. It starts with an `all classes` summary and progressively adds further drill-down sections. Because a columnar plain text format is limited in how well it can present hierarchical data, it is recommended that for report depths beyond `all` and `package` you use the HTML report instead.
- The HTML report can provide the most detail and is intended for human viewing. It starts with an `all classes` summary page and for larger report depths links it to package summary pages and then further to individual source file and class summary pages. Source/class summary pages can further embed source files and show method coverage rollups as well as highlight individual source line coverage states.
- The XML report exits for integration purposes and leverages the tree structure of an XML document to most truthfully represent the above-mentioned data hierarchy.

Because generating certain report types for very large projects can be time-consuming, reducing the default report depth is a good way to limit the amount of detail that is generated, a useful feature for individual development work.

Valid values for a report depth are `all`, `package`, `source`, `class`, and `method`. In general, a certain report depth value implies the level of detail that includes the summary for all items at that level

as well as coverage breakdown summaries for their children. The amount of information rendered for a given depth value is always inclusive of lesser depth values, so increasing the report depth always increases the amount of details that is rendered. As a special case, when full debug info is available, `class` is equivalent to `source`.

**Report units.** EMMA coverage metrics could be *unweighted* or *weighted*, that is derived from basic block coverage where each block counts either with an equal weight or with a weight proportional to the number of Java bytecode instructions in it. The default **<report>/report** behavior is to use weighted metrics. This includes all metrics that are sensitive to basic block content: line and block coverage. Weighted basic block coverage is a recommended metric for all situations, because it can simulate line coverage when no debug information has been compiled into application classes. If desired, the traditional (unweighted) metrics could be selected using the units option.

**Coverage metrics.** A very useful feature of HTML and plain text reports created by **<report>/report** is the ability to highlight entities that fail a given coverage metric. The plain text report does it by appending a "!" to a failing coverage metric and the HTML report highlights those in red. Combined with ability to sort report columns, this feature allows an individual developer to zoom in to the packages and classes that demand the most attention with respect to coverage.

**Sourcepath and source linking.** Although EMMA coverage calculations are based on basic block coverage profiling, **<report>/report** can also map block coverage to Java source file lines. If the HTML report generator is set to `method` depth and is configured with a valid source path and the instrumented classes were compiled with enough debug information, the generator will embed source files in the source file/class summary report pages and highlight covered/not covered lines accordingly.

### Sourcepath and coverage stats

Referencing the original Java source files is optional during coverage report generation and does not affect how EMMA coverage stats are computed (these stats are based entirely on the *class metadata* and the debug info available in the `.class` data at the instrumentation time). However, to avoid report generation errors it is your responsibility to ensure that the versions of Java sources used for reporting are the same as the ones used during instrumentation.

# 4.2. ANT usage

## Parameters specified as attributes

| Attribute | Description | Required |
|---|---|---|
| [common EMMA task attributes] | | No |
| `sourcepath` | An optional source path to use for report generation (a *path-like structure*). It is interpreted as a list of directories (separated by the OS-specific classpath separator or comma) containing `.java` source files. The local path names within each directory should reflect class package names. (Currently, only the HTML report generator uses this data, and only at `method` report depth.) | No |
| `sourcepathref` | Same as `sourcepath`, but given as a reference to a path defined elsewhere. | No |
| `units` | Specifies whether weighted or unweighted coverage metrics are calculated. Valid values are:<br>• `instr` (default): use metrics weighted by | No |

| Attribute | Description | Required |
|---|---|---|
| | bytecode instruction count;<br>• `count`: use traditional metric definitions (each basic block has equal weight). | |
| `depth` | Specifies the amount of detail to be included in the generated coverage reports, as described in Report depth. [18]. Valid values (in order of increasing level of detail) are:<br>• `all`<br>• `package`<br>• `source`<br>• `class`<br>• `method`<br>(default values are report type-specific, see below) | No |
| `columns` | Specifies which report columns and in which order to use for report generation, as a comma-separated list of column ids. Valid column ids are the name of the item reported on and various types of coverage: `name`, `class`, `method`, `block`, and `line`. Coverage types that are not available for a given item type and debug info level are automatically ignored. Reports can use only a subset of all possible columns (and different report types can use different subsets). Duplicate column names are ignored. Setting this attribute is the same as setting the report.columns [] property for all report types (default values are report type-specific, see below). | No |
| `sort` | Specifies report column sorting order, as a comma-separated list of column ids prefixed with "+" for ascending or "-" for descending directions. The first column id is the primary sort and subsequent column ids are secondary sorts, in the order given. Only the column ids specified by `columns` attribute are considered. Setting this attribute is the same as setting the report.sort [] property for all report types (default: `+block,+name,+method,+class`). | No |
| `metrics` | Specifies the threshold coverage metric values for a given set of columns (all coverage percentages that are below a given threshold are marked up in the report types that support this). The value is a comma-separated list of column id-value pairs, with the value being the minimum required coverage percentage. Setting this attribute is the same as setting the report.metrics [] property for | No |

| Attribute | Description | Required |
|---|---|---|
| | all report types (default: `method:70,block:80,line:80,class:100`). | |
| `encoding` | Sets the charset id for report output files (this is best done at the individual report type level). Setting this attribute is the same as setting the report.out.encoding [] property for all report types (default values are report type-specific, see below). | No |

## Parameters specified as nested elements

| Element | Description | Required |
|---|---|---|
| [`common EMMA task nested elements`] | | No |
| `<infileset>`, `<fileset>` | A *FileSet* that selects a set of *metadata* and *coverage data* files that form the basis of coverage calculations in the generated report(s). It is an error not to include any metadata or any coverage data within this set of files. | Yes |
| `<sourcepath>` | A *path-like structure* that specifies an optional source path to use for HTML report generation. | No |
| `<txt>` | Instructs <**report**> to generate a plain-text coverage report. The report can be further customized as shown below. | At least one of `<txt>`, `<html>`, `<xml>` is required |
| `<html>` | Instructs <**report**> to generate an HTML coverage report. The report can be further customized as shown below. | At least one of `<txt>`, `<html>`, `<xml>` is required |
| `<xml>` | Instructs <**report**> to generate an XML coverage report. The report can be further customized as shown below. | At least one of `<txt>`, `<html>`, `<xml>` is required |

**`<infileset>` nested elements.** `<infileset>` nested elements are configured as any other *FileSet* [http://ant.apache.org/manual/CoreTypes/fileset.html] data type in ANT. Additionally, EMMA's version of *FileSet* data type allows `file` attribute in ANT versions earlier than 1.5.x (which is useful for selecting a single file by its known name without using an explicit *PatternSet*).

**`<sourcepath>` nested elements.** `<sourcepath>` is a *path-like structure* [http://ant.apache.org/manual/using.html#path] that can be used to point <**report**>/**report** to the location of your Java source files. If the HTML report depth is set to `method` and the instrumented classes were compiled with enough debug information, the report generator will embed whichever source files it can find inside the HTML report pages and highlight covered/not covered lines.

**`<txt>`, `<html>`, and `<xml>` nested elements.** These nested elements create plain text, HTML, and XML coverage reports, respectively. At least one report type must be specified (at most one configurator of any given report type can be nested inside a given <**report**>). All of them accept the same set of

report configuration attributes (if a particular attribute is not specified for an element, its value is inherited from the **<report>** parent. If the parent task does not specify an attribute value either, the usual EMMA property inheritance rules determine the eventual value):

| Attribute | Description | Required |
|---|---|---|
| `units` | Overrides the coverage metric units for a given report type. It is perhaps best to set this at the parent **<report>** level, so that all generated reports use consistent units. | No |
| `depth` | Overrides the report depth for a given report type. The default values are:<br>• `txt` report: `all`<br>• `html` report: `method`<br>• `xml` report: `method` | No |
| `columns` | Overrides the report column selection and order for a given report type.The default values are:<br>• `txt` report: `class,method,block,line,name`<br>• `html` report: `name,class,method,block,line`<br>• `xml` report: `name,class,method,block,line` | No |
| `sort` | Overrides the report column sort order for a given report type. | No |
| `metrics` | Overrides the report coverage metrics thresholds for a given report type. It is perhaps best to set this at the parent **<report>** level, so that all generated reports use consistent metrics. | No |
| `outfile` | Overrides the default report output file location. The default settings are:<br>• `txt` report: file `coverage.txt` in the current directory<br>• `html` report: file `coverage/index.html` (note that the HTML report is usually split over multiple files, so it is best to specify a file pathname that is inside a dedicated subdirectory)<br>• `xml` report: file `coverage.xml` in the current directory | No |
| `encoding` | Overrides the output file charset encoding used for a given report type. The default values are: | No |

| Attribute | Description | Required |
|---|---|---|
| | • `txt` report: mirrors the `file.encoding` JRE system property<br>• `html` report: `ISO-8859-1`<br>• `xml` report: `UTF-8` | |

## Examples

- Generate plain text and XML report types, all with default settings:

```
<emma enabled="${emma.enabled}" >
  <report >
    <!-- collect all EMMA data dumps (metadata and runtime): -->
    <infileset dir="${coverage.dir}" includes="*.em, *.ec" />

    <txt />
    <xml />
  </report>
</emma>
```

- Generate three report types, with common metrics and column sorting, but with different report depth and column orderings:

```
<emma enabled="${emma.enabled}" >
  <report sourcepath="${src.dir}"
          sort="+block,+name,+method,+class"
          metrics="method:70,block:80,line:80,class:100"
  >
    <infileset dir="${coverage.dir}" includes="*.em, *.ec" />

    <!-- for every type of report desired, configure a nested
         element; various report parameters
         can be inherited from the parent <report>
         and individually overridden for each report type:
    -->
    <txt outfile="${coverage.dir}/coverage.txt"
         depth="package"
         columns="class,method,block,line,name"
    />
    <xml outfile="${coverage.dir}/coverage.xml"
         depth="package"
    />
    <html outfile="${coverage.dir}/coverage.html"
         depth="method"
         columns="name,class,method,block,line"
    />
  </report>
</emma>
```

- Generate an HTML report with some customization, load metadata and runtime coverage data from a single session file, use a <dirset> to set the sourcepath:

```
<emma enabled="${emma.enabled}" >
  <report >
    <infileset file="${coverage.dir}/coverage.es" />

    <sourcepath>
      <dirset dir="${basedir}" >
        <include name="**/src" />
      </dirset>
    </sourcepath>

    <html outfile="${coverage.dir}/index.html"
          columns="name, method, line"
          sort="+line, +name"
          metrics="line:80"
    />
  </report>
</emma>
```

# 4.3. Command line usage

## Synopsis

`java emma report` {-in *`data files`*...} {-r *`report types`*...} [-sp *`sourcepath`*...] [`common options`]

## Options

[`common command line options`]

`-in, -input` *`meta/coverage data files`*...
This *repeatable* option selects a set of *metadata* and *coverage data* files that form the basis of coverage calculations in the generated report(s). It is an error not to include any metadata or any coverage data within this set of files.

`-r, -report (txt|html|xml)`...
This *repeatable* option selects report type(s) to be generated.

`-sp, -sourcepath` *`list of source directories`*...
This *repeatable* option sets the (optional) source path to use for report generation. It is interpreted as a list of directories (separated by the OS-specific classpath separator or comma) containing `.java` source files. The local path names within each directory should reflect class package names. (Currently, only the HTML report generator uses this data, and only at `method` report depth.)

`report generation options`...
Unlike its ANT equivalent, **report** command line tool does not have dedicated options for controlling coverage report generation. If necessary, they can be set using generic -D, -properties, and other mechanisms.

So, for example, to change the default location of the HTML report you would override the report.out.file [] property:

```
>java emma report -Dreport.html.out.file=mydir/mycoverage.html ...
```

(`report.html.out.file` can be abbreviated to `report.out.file` if the command

generates a single report type)

## Examples

- Generate plain text and XML report types, both with their default settings:

```
>java emma report -r txt,xml -in coverage.em -in coverage.ec
```

- Generate the HTML report only, but override the default output location:

```
>java emma report -r html -in coverage.em,coverage.ec -sp src/ -Dreport.html.out
```

- Generate three report types, with common metrics and column sorting, but with different report depth and column orderings. Use `-properties` option to pull in a large number of report property overrides:

```
>java emma report -r txt,xml,html -props my.properties -in coverage.em,coverage.
```

where file `my.properties` contains:

```
report.sort          = +block,+name,+method,+class
report.metrics       = method:70,block:80,line:80,class:100
report.depth         = package

report.txt.out.file  = coverage/coverage.txt
report.txt.columns   = class,method,block,line,name

report.xml.out.file  = coverage/coverage.xml

report.html.out.file = coverage/coverage.html
report.html.depth    = method
report.html.columns  = name,class,method,block,line
```

- Generate an HTML report with some customization, load metadata and runtime coverage data from a single session file:

```
>java emma report -r html -in coverage.es -sp src/   \
                  -Dreport.columns=name,method,line  \
                  -Dreport.sort=+line,+name          \
                  -Dreport.metrics=line:80
```

## Diagnostics

The default EMMA command line tool behavior is not to use `System.exit()` on exit unless an explicit `-exit` option is specified. If that is done, the error codes returned via `System.exit()` are as follows:

| | |
|---|---|
| `0` | Successful completion. |
| `1` | Failure due to incorrect option usage. This error code is also returned when command line |

| | |
|---|---|
| | usage (`-h`) is requested explicitly. |
| 2 | All other failures. |

# 5. <merge>/merge

**<merge>/merge** — offline meta- and runtime coverage data compressor.

## 5.1. Description

**<merge>/merge** is EMMA's offline meta- and runtime coverage data compressor. It reads in an arbitrary number of data files containing *class metadata* and/or *runtime coverage data* and compresses all of it into a single session data file.

**Why merge?** Despite the fact that all other EMMA tools can do in-memory merging of an arbitrary number of input data files, there are valid reason for using **<merge>/merge** tool:

keeping everything together
: Coverage metrics for a particular application could be determined by a large set of meta- and runtime coverage data files, not necessarily collected in a single application run. For example, a Swing client could run in one JVM and a remoted server in another, possibly on a different host machine. Or a testsuite could be spread over a sequence of forked JVM processes.

  Collecting all EMMA data in a single file could be a simple matter of convenience: such a *coverage session data* file is a memento of a particular state of application coverage. Such a session data file contains all the data necessary to regenerate all coverage reports (for source code embedding in a coverage report you also need to preserve the particular versions of sources used at the instrumentation time: a source revision control system is a good solution for this).

data compaction
: When merging data into existing files, for reasons that have to do with performance and making file writes as transactional as possible, EMMA tools use an append-like technique. Once a given data record is written to a file, it is never overwritten (rather, later data writes implicitly override it). What this means is that EMMA's un-merged data files may not always store data in the most compact way possible. Processing them with **<merge>/merge** eliminates wasted file storage and recovers disk space.

## 5.2. ANT usage

### Parameters specified as attributes

| Attribute | Description | Required |
|---|---|---|
| `[common EMMA task attributes]` | | No |
| `mergefile`, `outfile`, `tofile`, `file` | Overrides the location to store merged data (default: file `coverage.es` in the current directory). | No. |

### Parameters specified as nested elements

| Element | Description | Required |
|---|---|---|
| [common EMMA task nested elements] | | No |
| <infileset>, <fileset> | A *FileSet* that selects an arbitrary mix of *metadata* and *coverage data* to be merged. | Yes |

**<infileset> nested elements.** <infileset> nested elements are configured as any other *FileSet* [http://ant.apache.org/manual/CoreTypes/fileset.html] data type in ANT. Additionally, EMMA's version of *FileSet* data type allows `file` attribute in ANT versions earlier than 1.5.x (which is useful for selecting a single file by its known name without using an explicit *PatternSet*).

## Examples

- Collect all EMMA metadata and runtime coverage data files and merge them into a single session file:

```
<emma>
  <merge outfile="${coverage.dir}/coverage.es" >
    <fileset dir="${coverage.dir}" includes="*.em, *.ec" />
  </merge>
</emma>
```

- Compact a single data file:

```
<emma>
  <merge outfile="${coverage.dir}/coverage.es" >
    <fileset file="${coverage.dir}/coverage.es" />
  </merge>
</emma>
```

# 5.3. Command line usage

## Synopsis

`java emma merge` {-in *`data files`*...} [-out *`data file`*] [`common options`]

## Options

[`common command line options`]

`-in, -input` *`meta/coverage data files`*...
This *repeatable* option selects an arbitrary mix of *metadata* and *coverage data* files to be merged.

`-out, -outfile` *`output merge data file`*
Overrides the location to store merged data (default: file `coverage.es` in the current directory).

## Examples

- Collect all EMMA metadata and runtime coverage data files and merge them into a single session file:

```
>java emma merge -in coverage.em -in coverage.ec -out coverage.es
```

- Compact a single data file:

```
>java emma merge -in coverage.es -out coverage.es
```

### Diagnostics

The default EMMA command line tool behavior is not to use `System.exit()` on exit unless an explicit `-exit` option is specified. If that is done, the error codes returned via `System.exit()` are as follows:

| | |
|---|---|
| `0` | Successful completion. |
| `1` | Failure due to incorrect option usage. This error code is also returned when command line usage (`-h`) is requested explicitly. |
| `2` | All other failures. |

# 6. Defining the instrumentation set

## 6.1. How EMMA determines which classes get instrumented

Although EMMA's instrumentation is very fast (it is usually fast enough so that the overall processing time is dominated by file I/O), the key to making EMMA into an even faster tool for individual development is to make EMMA do just the right amount of work, i.e. define the right *instrumentation set* of classes.

Understanding what gets instrumented is also important for another reason: EMMA coverage reports are based exclusively on the classes in the instrumentation set as implied by coverage metadata.

**Instrumentation set.** The set of classes that get instrumented in a given invocation of a tool like **<instr>/instr** or **<emmajava>/emmarun** is determined by the following rules:

i. First, a set of classes *eligible for instrumentation* is determined by an *instrumentation path*. For **<instr>/instr** this is set via an explicit option and for **<emmajava>/emmarun** it is the same as the application classpath. Note that only Java classes contain executable code and are eligible for instrumentation: Java interfaces are never instrumented by EMMA[3].

ii. Next, the set of eligible classes as determined by the above step can be further narrowed down by a set of coverage *inclusion and exclusion filters*.

iii. Finally,

- in the offline processing mode, all remaining eligible classes are in fact instrumented and added to the metadata;

[3] Strictly speaking, Java interfaces can contain executable bytecode, but it usually corresponds to field initializer expressions that execute unconditionally when the interface is loaded.

- in the on-the-fly processing mode, <**emmajava**>/**emmarun** behavior with respect to the remaining eligible classes depends on whether the *full classpath scan mode* (`fullmetadata` (`-f`) option) is turned on:
  - by default, this mode is not on and only the classes actually used by the application are instrumented and added to the coverage metadata (this happens on demand);
  - if this mode is on, all remaining eligible classes are added to the metadata (this happens before the application starts running).

What this means in practice is that you choose the right set of class directories and archives via the instrumentation path option and then narrow it further down via a number of coverage filters, described next.

# 6.2. Coverage filters

**Wildcards.** EMMA coverage filters are lists of familiar `*`,`?`-wildcard class name patterns:

- *full* Java class names are implied, with "`.`"'s (dots) as Java package separators;
- a "`*`" in a pattern stands for zero or more class name characters;
- a "`?`" in a pattern stands for exactly one class name character

### Filters work "across directories"

You should not think of coverage filters as applying at the *file* level. Instead, think of them as applying at the classpath level. Thus, if you exclude "`*Test*`", for example, it will have effect on all application classes that are subject to instrumentation, no matter from how many classpath directories and/or *.jar*s they come.

**Inclusions and exclusions.** Sometimes all you want is to zoom in to a particular Java package and don't want to specify a long list of exclusion patterns (that also needs to be updated each time someone on your team adds a new Java package to the application). Other times, you do want to include all of the application's classes but would like to exclude just a few packages or name patterns (e.g., your testcases or perhaps a package that is well-tested and can be excluded from coverage analysis).

EMMA coverage filters support all such scenarios and more, in a reasonably concise and intuitive fashion. Each coverage filter pattern is either an inclusion or an exclusion pattern:

- By default, a pattern is an inclusion pattern. This can also be made explicit by prefixing it with a "`+`" (plus sign).
- A pattern is an exclusion pattern if it is prefixed with a "`-`" (minus sign).

Informally, the way inclusions and exclusions work can be summarized as follows: a class name is in the instrumentation set if it is included and not excluded. An *empty* list of inclusion patterns implicitly includes everything (i.e., it is like "`+*`") and an empty list of exclusion patterns implicitly excludes nothing. Furthermore, a class name is included if it matches at least one of the inclusion patterns and not excluded by any exclusion pattern. It is best to show the formal matching rule via pseudocode, followed by some examples:

**Inclusion/exclusion matching algorithm.**

```
if (inclusions is not empty)
{
    boolean included = false;

    foreach (pattern in inclusions)
    {
        if (pattern matches classname)
        {
            included = true;
            break;
        }
    }

    if (not included) return classname_is_excluded;
}

if (exclusions is not empty)
{
    foreach (pattern in exclusions)
    {
        if (pattern matches classname) return classname_is_excluded;
    }
}

return classname_is_included;
```

## 6.2.1. filter syntax: ANT

EMMA ANT (sub)tasks that can take instrumentation filter strings either as a `filter` attribute or as `<filter>` nested elements follow the same specification syntax:

**`filter` attribute.** The attribute value is a filter string, which is a list of inclusion/exclusion patterns, separated with white space and/or commas. Each inclusion/exclusion pattern is a `*`,`?`-wildcard class name mask, prefixed with `+` and `-` for inclusion and exclusion, respectively. It is legal to omit a prefix, in which case the inclusion prefix, `+`, is implied.

It is also possible to specify a list of inclusion/exclusion patterns to be loaded from an external file. To do so, you can set the `filter` attribute to an `@`-prefixed file name. The file should contain a list of inclusion/exclusion patterns, one per line (empty lines and lines starting with a "#" are ignored).

**`<filter>` nested element.** This nested element can be configured through a combination of the following attributes:

| Attribute | Description |
|---|---|
| `value` | Specifies a list of inclusion/exclusion patterns, separated with white space and/or commas. Each inclusion/exclusion pattern is a `*`,`?`-wildcard class name mask, prefixed with `+` and `-` for inclusion and exclusion, respectively. It is legal to omit a prefix, in which case the inclusion prefix, `+`, is implied. Note that this attribute allows you to specify a mixed list of inclusions and exclusions. |
| `includes` | Specifies a list of patterns, separated with white space and/or commas. Each pattern is a `*`,`?`-wildcard class name mask, interpreted as an inclusion pattern. All explicit `+`/`-` prefixes are ignored. |
| `excludes` | Specifies a list of patterns, separated with white space and/or commas. Each pattern is a `*`,`?`-wildcard class name mask, interpreted as an exclusion pattern. All explicit `+`/`-` prefixes are ignored. |

| Attribute | Description |
|---|---|
| `file` | Specifies a list of patterns to be loaded from a file with a given name. The file should contain a list of inclusion/exclusion patterns, one per line (empty lines and lines starting with a "#" are ignored). Each inclusion/exclusion pattern is a `*`,`?`-wildcard class name mask, prefixed with `+` and `-` for inclusion and exclusion, respectively. It is legal to omit a prefix, in which case the inclusion prefix, `+`, is implied. Note that such a file can contain a mixed list of inclusions and exclusions. |

### ANT tip

Note that if any of these attributes is set to an empty string it is ignored. This is convenient for providing team-wide ANT filter "hooks" that are overridden differently by individual developers using ANT command line (`-Demma.filter=...`, etc).

### Examples

Note that all these different ways of specifying instrumentation filters can be used in a combination. The result is a union of all specified patterns. The following examples all specify the same set of inclusion/exclusion patterns:

- ```
  <filter includes="com.foo.*" excludes="com.foo.test.*, com.foo.*Test*" />
  ```

- ```
  <filter includes="com.foo.*" />
  <filter excludes="com.foo.test.*, com.foo.*Test*" />
  ```

- ```
  <filter value="+com.foo.*, -com.foo.test.*, -com.foo.*Test*" />
  ```

- ```
  <filter excludes="com.foo.*Test*" file="myfilters.txt" />
  ```

  where `myfilters.txt` file contains these lines:

  ```
  -com.foo.test.*
  +com.foo.*
  ```

## 6.2.2. filter syntax: command line

EMMA command line tools that can accept instrumentation filter strings do it via the `-ix` option. This *repeatable* option should be set to a list of inclusion/exclusion patterns, separated with white space and/or commas. Each inclusion/exclusion pattern is a `*`,`?`-wildcard class name mask, prefixed with `+` and `-` for inclusion and exclusion, respectively. It is legal to omit a prefix, in which case the inclusion prefix, `+`, is implied.

It is also possible to specify a list of inclusion/exclusion patterns to be loaded from an external file. To

do so, you can set the option value to point to an @-prefixed file name. The file should contain a list of inclusion/exclusion patterns, one per line (empty lines and lines starting with a "#" are ignored).

### Examples

Note that all these different ways of specifying instrumentation filters can be used in a combination. The result is a union of all specified patterns. The following examples all specify the same set of inclusion/exclusion patterns:

- ```
  >java emma ... -ix +com.foo.*,-com.foo.test.*,-com.foo.*Test* ...
  ```

- ```
  >java emma ... -ix com.foo.* -ix -com.foo.test.*,-com.foo.*Test* ...
  ```

- ```
  >java emma ... -ix -com.foo.*Test* -ix @myfilters.txt ...
  ```

  where `myfilters.txt` file contains these lines:

  ```
  -com.foo.test.*
  +com.foo.*
  ```

# 7. Common ANT task and command line options

## 7.1. Common ANT task attributes and nested elements

All EMMA tasks and subtasks have a set of common attributes and nested elements:

**Table 2.3. Common EMMA ANT task attributes**

| Attribute | Description | Required |
|---|---|---|
| `enabled` | If set to `false`, disables the corresponding EMMA ANT task or subtask. This can be used to effect a simple form of build control flow (default: `true`). | No (everything is enabled by default) |
| `verbosity` | Sets the verbosity level for a given task or an entire <emma> group of tasks. Valid values in the order of increasing verbosity are:<br>• `silent` (same as `severe`): only severe errors are reported;<br>• `quiet` (same as `warning`): only warnings | No (defaults to `info`) |
  ```

| Attribute | Description | Required |
|---|---|---|
| | and severe errors are reported;<br>• `info`: default verbosity level, with EMMA reporting on completion of various activities, warnings, and errors<br>• `verbose`: this setting makes EMMA chattier than normal, with extra progress reporting;<br>• `trace1`, `trace2`, `trace3`: these settings enable internal tracing (useful mostly for debugging).<br>(default: `info`). | |
| `properties` | This option specifies a pathname for an EMMA property override file, in the standard `java.util.Properties` format. See Chapter 3, *EMMA Property Reference* [35] for more information on setting EMMA properties. | No |

**Table 2.4. Common EMMA ANT task nested elements**

| Element | Description | Required |
|---|---|---|
| `<property>` | This nested element sets a single EMMA property. It is the ANT equivalent of `-D` command line option and is provided mostly for internal testing (EMMA ANT tasks provide proper attributes and nested elements for all public tool settings). See Chapter 3, *EMMA Property Reference* [35] for more information on setting EMMA properties. | No |

**`<property>` nested element.** This nested element can be used to set a generic EMMA property as a name-value pair using the following attributes:

| Attribute | Description | Required |
|---|---|---|
| `name` | Property name (without "`emma.`" prefix). | Yes |
| `value` | Property value. | Yes |

# 7.2. Common command line options

All EMMA command line tools have a set of common options:

`-p, -props, -properties` *`.properties file`*
: This option specifies a pathname for an EMMA property override file, in the standard `java.util.Properties` format. The pathname could be either relative (in which case it is resolved relative to the JVM's current working directory) or absolute. See Chapter 3, *EMMA Property Reference* [35] for more information on setting EMMA properties.

This option sets a single EMMA property. Note that this is different from using the JVM `-D` option (the *`name`* is not "`emma.`"-prefixed). See Chapter 3, *EMMA Property Reference* [35] for more information on EMMA properties.

`-exit`
: To enable tool chaining and integration via `main()` entry methods, EMMA command line tools do not terminate via `java.lang.System.exit()` by default. If desired for shell and makefile integration, this can be changed by using this option on the command line. The *Diagnostics* section on every tool's reference page details the error codes the tool returns to the operating system.

`-silent`
`-quiet`
`-verbose`
: These options set EMMA tool verbosity levels to be much lower than normal, lower than normal, and above normal, respectively. These options are shortcuts to setting the verbosity.level [] property.

`-h, -help`
: This option causes EMMA tools to print their usage summaries to `System.out`. The longer form of the option results in slightly more detailed usage printout.

# Chapter 3. EMMA Property Reference

The behavior of EMMA tools and runtime is influenced by a number of EMMA properties (see Section 2, “EMMA property summary” [36] for the full list). These properties address several needs:

- It could be tedious or impractical to specify all individual command line options for certain aspects of EMMA behavior (e.g., coverage report generation properties). A large set of EMMA property value overrides can be kept in a file that is referenced with a single option or from the classpath.
- Certain kinds of instrumented runtimes cannot be configured easily via command line options (e.g., servlets or Enterprise Java Beans (EJBs)). Again, other ways of passing property overrides are need in such cases, such the JVM system options or classpath resource files.

# 1. Specifying EMMA properties

Given an EMMA *`property`* (from the tables in Section 2, “EMMA property summary” [36]), it can be specified in several different ways:

- as a JVM system property named `emma.`*`property`*;
- as a property named `emma.`*`property`* defined in an external `java.util.Properties` file;
- as a property named *`property`* defined in a `java.util.Properties` classloader resource;
- as an explicit ANT task `<property>` or command line (`-D`) property override.

The general rule is that when a property is provided at the JVM system level, its name must be prefixed with “`emma.`” in order to be in EMMA namespace.

## Examples

The following shows different ways of overriding the default coverage session dump file pathname from ANT or command line. Using `emma.properties` or `properties` ANT is convenient when you want to load a large block of EMMA property settings without keeping them in your makefile or `build.xml`:

- From ANT:

```
<emmajava enabled="${emma.enabled}" libclasspathref="emma.lib"
          dumpsessiondata="yes" properties="my.properties"
          classname="Main"
          classpathref="run.classpath"
/>
```

  where file `my.properties` sets `session.out.file` to some *`mydir/myfile`* value.

- From command line:

```
>java emmarun -Dsession.out.file=mydir/myfile ...
>java emmarun -properties my.properties ...
>java -cp {directory containing emma.properties} emmarun ...
```

```
>java -Demma.session.out.file=mydir/myfile emmarun ...
>java -Demma.properties=my.properties emmarun ...
```

where file `my.properties` sets `session.out.file` to some *`mydir/myfile`* value.

**EMMA property lookup order.** Because of multiple ways to specify the same EMMA property, it is necessary to document the exact property lookup mechanism in order to disambiguate potential conflicts. The following lists all possible property lookup layers, in the order from the least specific to the most specific (in other words, later definitions override earlier definitions):

1. Certain properties like `report.txt.out.encoding`, start out with their default values reflecting Java built-in system properties like `file.encoding`. Other properties have their defaults set in EMMA code.

2. A property can be set in an external `java.util.Properties` file whose pathname is the value of the JVM system property `emma.properties` (`-Demma.properties=`*`filename`*). EMMA properties in such a file are *not* prefixed with "`emma.`".

3. A property can be set at the JVM system property level, using `-Demma.`*`property=value`* syntax.

4. A property can be set in a classloader resource named `emma.properties` using the usual `java.util.Properties` format (*without* the "`emma.`" prefix). The resource is looked up first in the most specific of the {current, thread context} pair of classloaders. If neither one is more specific, the lookup is done in the current classloader. If the system classloader is more specific than either the current or the context classloader, then the system classloader is used.

5. A property can be set in an external `java.util.Properties` file whose pathname is the value of a `properties` ANT task attribute or a `-properties` command line option. EMMA properties in such a file are *not* prefixed with "`emma.`".

6. A property can be set via a `property` ANT task nested element (`<property name='`*`name`*`' value='`*`value`*`'/>`) or a `-D` EMMA command line option (`-D`*`property=value`*). (*Without* the "`emma.`" prefix: don't confuse this with JVM system properties.) Note that in the **`<report>`** task case this mechanism is largely redundant, because all report generation properties have dedicated ANT task attributes.

**Property shortcuts.** In the special case of report generation properties (`report.*`) there is one additional complication. Any property name that follows the `report.`*`name`* patterns is actually a report property *shortcut* in the sense that is applies to all report types (plain text, HTML, XML). For some aspects of report generation (e.g., `report.units`) this is very appropriate, but for others (e.g., `report.out.encoding`) you are likely to want report type-specific settings. To do so, you can specify a `report.`*`report_type.name`* property. For example, `report.txt.out.encoding` is more specific than `report.out.encoding` as far as the plain-text report generator is concerned. Note that the **`<report>`** ANT task makes this more convenient that the command line case, because it provides convenience override attributes on all nested `<txt>`, `<html>`, etc elements.

# 2. EMMA property summary

### Table 3.1. EMMA file output properties

| | |
|---|---|
| Property: | *coverage.out.file* |
| Default: | `coverage.ec` |
| Tools affected: | EMMA runtime (EMMA-instrumented classes) |
| Description: | In the offline instrumentation mode, setting this property is currently the only way to override the pathname of the runtime coverage output file. If the pathname is not absolute, it is resolved relative to the current JRE directory (`user.dir` system property). Any existing data in the target file is overwritten unless the `coverage.out.merge` mode is `true`. |

| | |
|---|---|
| Property: | *coverage.out.merge* |
| Default: | `true` |
| Tools affected: | EMMA runtime (EMMA-instrumented classes) |
| Description: | When runtime coverage data is dumped by EMMA runtime, this property specifies whether any existing data in the target file should be merged into or overwritten. |

| | |
|---|---|
| Property: | *metadata.out.file* |
| Default: | `coverage.em` |
| Tools affected: | **<instr>/instr** |
| Description: | For tools that can output class instrumentation metadata, this property sets the output file pathname. If the pathname is not absolute, it is resolved relative to the current JRE directory (`user.dir` system property). Any existing data in the target file is overwritten unless the `metadata.out.merge` mode is `true`. |

| | |
|---|---|
| Property: | *metadata.out.merge* |
| Default: | `true` |
| Tools affected: | **<instr>/instr** |
| Description: | For tools that can output class instrumentation metadata, this property specifies whether any existing data in the target file should be merged into or overwritten. |

| | |
|---|---|
| Property: | *session.out.file* |
| Default: | `coverage.es` |
| Tools affected: | **<emmajava>/emmarun**, **<merge>/merge** |

| | |
|---|---|
| Description: | For tools that can output combined metadata+coverage (i.e., session) data, this property sets the output file pathname. If the pathname is not absolute, it is resolved relative to the current JRE directory (`user.dir` system property). Any existing data in the target file is overwritten unless the `session.out.merge` mode is `true`. |
| Property: | *`session.out.merge`* |
| Default: | `true` |
| Tools affected: | **<emmajava>/emmarun**, **<merge>/merge** |
| Description: | For tools that can output combined metadata+coverage (i.e., session) data, this property specifies whether any existing data in the target file should be merged into or overwritten. |

**Table 3.2. EMMA report generation properties**

| | |
|---|---|
| Property: | *`report.units`* |
| Default: | `instr` |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property selects either weighted (`instr`) or unweighted (`count`) coverage metrics. See Report units. [19] for more details. |
| Property: | *`report.depth`* |
| Default: | *`report.depth`* is set to `method` and overridden for the plain-text report type:<br>• *`report.txt.depth`*: `all` |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property selects the report depth level covered by the report. Valid values (in order of increasing level of detail) are `all`, `package`, `source`, `class`, and `method`. See Report depth. [18] for more details. |
| Property: | *`report.columns`* |
| Default: | *`report.columns`* is set to `name,class,method,block,line` and overridden for the plain-text report type: |

| | |
|---|---|
| | • *report.txt.columns*: `class,method,block,line,name` |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property specifies which coverage metrics and in which left-to-right order to render in the report output (the XML report is an exception because it is not columnar: instead, the column order is used for top-to-bottom XML element rendering). Valid column ids are the item name and various types of coverage: `name`, `class` (class coverage), `method` (method coverage), `block` (block coverage), and `line` (line coverage). Coverage types that are not available for a given item type and debug info level are automatically ignored. It is perfectly legal so use only a subset of all possible metrics (e.g., reporting both block and line coverages is somewhat of an overkill). |
| Property: | *report.sort* |
| Default: | `+block,+name,+method,+class` |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property specifies how to sort data by coverage metrics: which metrics to sort by, sort directions, and sort order. It should be set to a comma-separated list of metric ids (`name`, `class`, `method`, `block`, and `line`), with each metric id prefixed with "+" for ascending or "-" for descending sort direction. Multiple sorts are applied in the left-to-right order of the metric ids as specified by this property. It is perfectly legal to sort only a subset of all possible metrics/columns. |
| Property: | *report.metrics* |
| Default: | `method:70,block:80,line:80,class:100` |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property specifies how to highlight data that fails minimum coverage requirements (only applies to plain-text and HTML reports). It should be set to a comma-separated list of metric id-value pairs, with the value being the minimum required coverage percentage (separated by a colon). Metrics ids are `name`, `class`, `method`, `block`, and `line`. It is not necessary to specify the required percentage for every metric used in a given report. |
| Property: | *report.out.file* |
| Default: | *report.out.file* is not set and is instead overridden for all report types:<br>• *report.txt.out.file*: `coverage.txt`<br>• *report.html.out.file*: `coverage/index.html` |

| | |
|---|---|
| | • *report.xml.out.file*: coverage.xml |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property can be used to override the default locations for the output files. When a relative pathname is specified, it is resolved relative to the current JRE directory (user.dir system property). Note that the HTML report generator creates secondary HTML files beyond the report home page file. These files will be put in a subdirectory that is a sibling of the home page file and it thus makes sense to have at least one subdirectory level specified for report.html.out.file (as is the case with the default value). EMMA will create any intermediate output directories as needed. Output files are always overwritten. |
| Property: | *report.out.encoding* |
| Default: | *report.out.encoding* defaults to the JRE file.encoding system property and is overridden for the HTML and XML report types:<br>• *report.html.out.encoding*: ISO-8859-1<br>• *report.xml.out.encoding*: UTF-8 |
| Tools affected: | **<emmajava>/emmarun**, **<report>/report** |
| Description: | During coverage report generation, this property can be set to customize the character encoding used for the output files. Default values provided by EMMA should be adequate in most situations. |

**Table 3.3. EMMA instrumentation properties**

| | |
|---|---|
| Property: | *instr.do_suid_compensation* |
| Default: | true |
| Tools affected: | **<emmajava>/emmarun**, **<instr>/instr** |
| Description: | Because the bytecode instrumentor used by EMMA needs to synthesize a static class initializer when there is none already and because class initializers are considered for serialization UID calculations by the default SUID algorithm, EMMA will add a compensating serialVersionUID field to the instrumented class if this property set to true. This is useful for running applications and tests when Java classes are (de)serialized from previously saved content or across client-server connections when not all JVMs are running instrumented classes. This property can be set to false to slightly speed up EMMA processing if such compensation is not needed (e.g., when no serialization is used). |
| | |

| | |
|---|---|
| Property: | *instr.exclude_synthetic_methods* |
| Default: | `true` |
| Tools affected: | **<emmajava>/emmarun**, **<instr>/instr** |
| Description: | Depending on the compiler used, the original application classes can contain any number of "synthetic" methods that implement certain Java language features (class literals, inner class accessors, etc) through bytecode that has no representation in the original sources. In most cases, excluding these methods from instrumentation and coverage reporting is the correct thing to do. This is the default EMMA behavior. |
| Property: | *instr.exclude_bridge_methods* |
| Default: | `true` |
| Tools affected: | **<emmajava>/emmarun**, **<instr>/instr** |
| Description: | J2SE 1.5 compilers introduce a new kind of "synthetic" methods used for mapping new features like generics to the existing class format. These methods do not have a source code representation. In most cases, excluding these methods from instrumentation and coverage reporting is the correct thing to do. This is the default EMMA behavior. |

### Table 3.4. EMMA logging properties

| | |
|---|---|
| Property: | *verbosity.level* |
| Default: | `info` |
| Tools affected: | all tools, including EMMA runtime (EMMA-instrumented classes) |
| Description: | This property sets the verbosity of EMMA logging. Valid values are (in the order of increasing verbosity):<br>• `silent` (same as `severe`): only severe errors are reported;<br>• `quiet` (same as `warning`): only warnings and severe errors are reported;<br>• `info`: default verbosity level, with EMMA reporting on completion of various activities, warnings, and errors<br>• `verbose`: this setting makes EMMA chattier than normal, with extra progress reporting;<br>• `trace1`, `trace2`, `trace3`: these settings enable internal tracing (useful mostly for debugging). |

# Glossary

| | |
|---|---|
| line coverage | Coverage metric whereby every source line is given a coverage percentage. EMMA derives line coverage from basic block coverage. The details of this computation are published elsewhere. |
| instrumentation run | A single invocation of the **instr** command line tool or the **<instr>** ANT task. |
| instrumentation set | A set of Java `.class` definitions that are included in an EMMA report. Note that this set is usually smaller than any set of files input into an EMMA tool because not all input classes are executable and users can do additional filtering. |
| class metadata | A set of binary Java class descriptors that record details of class structure like the number of methods in a class, their name and basic block structure. As long as the original classes are not recompiled, the same metadata can be used for any number of coverage profiling runs. See also Class coverage metadata. [11]. |
| coverage runtime data | A set of binary Java class descriptors that reflect a particular coverage profiling run (which method and basic blocks were hit for which class, etc). As long as they correspond to the same metadata, several such data sets can be merged together. |
| coverage session data | A combination of *metadata* and *runtime coverage data*, usually obtained during an on-the-fly application session or as a result of using **<merge>**/**merge** processor. When this combination is consistent (the runtime data corresponds to the classes in the metadata), it is sufficient for a coverage report to be produced. |
| mergeable option | See repeatable option. |
| repeatable option | An ANT task nested element or a command line option that could be used multiple times within a given tool invocation, with the resulting property value being a union of all provided inputs (with duplicates removed, if that makes sense for a given property). In most cases, an individual input value can also be a comma-separated list of values. |