Bisect 1.3

Reference Manual
http://bisect.x9c.fr

November 3, 2012

i

Contents

Chapter 1

Overview

1.1 Purpose

Bisect is a code coverage tool for the OCaml language[1]. Its name stems from the following acronym: *Bisect is an Insanely Short-sized and Elementary Coverage Tool*. The shortness of the source files can be seen as a tribute to the camlp4 tool and API bundled with the standard OCaml distribution. Over the time, features have been added and the source code is not so small anymore; however, the core functionality of Bisect is implemented only a few hundreds of lines.

Code coverage is a mean of software testing. Associated with unit or functional testing, the goal of code coverage is to measure the portion of the application source code that has actually been exercised by tests. To achieve this goal, the code coverage tool defines *points* in the source code and memorizes at runtime (that is, when tests are run) if the execution path of the program passes at these *points*. The so-called *points* are places of interest in the source code (as an example, the branches of an *if* or *match* construct are interesting *points*), to ensure that all alternatives have been tested. In practice, code coverage is often performed in three steps:

- first, the application is *instrumented*: this means that (the compiled form of) the application is enhanced in such a way that it will count at runtime how many times the application passed at a given *point*;

- then, the tests are actually run, producing some runtime data about code coverage;

- finally, a report is generated from the data produced at the previous step; this report shows which points were actually passed through during tests.

Bisect can be seen as an improved version of the `ocamlcp`/`ocamlprof`[2] couple (both of these tools being part of the standard OCaml distribution). In this respect, Bisect performs statement and condition coverage, but not path coverage. This means that it only counts how many times the application passed at each *point*, independently of which was the statement previously executed (that is, the previously visited point). At the opposite, path coverage is not only interested in *points* but also in *paths*, the goal being to ensure that every possible execution

---

[1]The official OCaml website can be reached at http://caml.inria.fr and contains the full development suite (compilers, tools, virtual machine, *etc.*) as well as links to third-party contributions.

[2]Indeed, a code coverage tool can be used as a primitive profiler because it tells how many time each execution point in the program is crossed.

path has been followed.

Code coverage is a useful software metric but, being based on tests, it cannot ensure that a program is correct. It only gives hints about where to look for untested, possibly dead, code. For program correction, one should consider more involved tools and formalisms such as *model checking*, or *proof systems*. Code coverage is still convenient in practice because it is a much simpler method that require no particular knowledge from the developer. Bisect provides several output modes (ranging from bare text to Jenkins[3]-compatible XML) in order to allow easy integration with an existing toolchain.

1.2   License

Bisect is released under the GPL version 3. This licensing scheme should not cause any problem, as instrumented applications are intended to be used during development but should not be released publicly. The GPL contamination has thus no consequence here.

1.3   Contributions

In order to improve the project, I am primarily looking for testers and bug reporters. Pointing errors in documentation and indicating where it should be enhanced is also very helpful.
Bugs and feature requests can be made at `http://bugs.x9c.fr`.
Other requests can be sent to bisect@x9c.fr.

---

[3]Continuous integration engine available at `http://http://jenkins-ci.org/`

Chapter 2

Building Bisect

2.1 Step 0: dependencies

Before starting to build Bisect, one first has to check that dependencies are already installed. The following elements are needed in order to build Bisect:

- OCaml, version 4.00.0 (trunk at revision 13061 or later for building using `ppx`);

- `make`, in its GNU Make 3.81 flavor;

- a classical Unix shell, such as `bash`;

- **optionally:** Findlib[1], version 1.3.3.

2.2 Step 1: configuration

The configuration of Argot is done by executing `./configure`. One can specify elements if they are not correctly inferred by the `configure` script; the following switches are available:

- `-ocaml-prefix` to specify the prefix path to the OCaml installation (usually `/usr/local`);

- `-ocamlfind` to specify the path to the `ocamlfind` executable;

- `-no-native-dynlink` to disable the build of the native version, even if native dynamic linking is available.

Moreover, two command-line switches allow to choose which version(s) of the instrumenter should be built:

- `-no-camlp4` to specify **not to build** the camlp4-based instrumenter (meaning that support for camlp4 is opt-out);

- `-ppx` to specify **to build** the ppx-based instrumenter (meaning that support for ppx is opt-in).

The Java[2] version will be built only if the `ocamljava`[3] compiler is present and located by the makefile. The syntax extension will be compiled only to bytecode.

---

[1]Findlib, a library manager for OCaml, is available at http://projects.camlcity.org/projects/findlib.html.

[2]The official website for the Java Technology can be reached at http://java.sun.com.

[3]OCaml compiler generating Java bytecode, by the same author – http://www.ocamljava.org

2.3   Step 2: compilation

The actual build of Bisect is launched by executing `make all`. When build is finished, it is possible to run some simple tests by running `make tests`. Documentation can be generated by running `make doc`.

2.4   Step 3: installation

Bisect is installed by executing `make install`. According to local settings, it may be necessary to acquire privileged accesses, running for example `sudo make install`. The actual installation directory depends on the use of `ocamlfind`: if present the files are placed inside the Findlib hierarchy, otherwise they are placed in the directory `‘ocamlc -where‘/bisect` (*i. e.* `$PREFIX/lib/ocaml/bisect`).

Chapter 3

Using Bisect

As previously stated, using a code coverage tool usually requires to follow three steps: instrumentation, execution, and report. Bisect is no exception in this respect; the following sections discuss each of these three steps.

3.1 Instrumenting the application

Bisect instruments the application at compile-time using either a camlp4- or a ppx-based preprocessor. Relying on preprocessors allows the user to choose exactly which module (*i. e.* source file) of the application should be instrumented. Code samples 1 and 2 show how to instrument a file named `source.ml` during compilation (the very same effect can be achieved using either `ocamlopt` or `ocamljava` as a replacement of `ocamlc`). Code sample 3 does the same through `ocamlfind`. During this step, Bisect will produce a file named `source.cmp`[1]. Files with the `cmp` extension contain *point* information for a given source file, that is: identifiers, positions, and kinds of *points*. Of course, the usual `cmi`, `cmo`, `cmx`, and `cmj` files are also produced, depending on the compiler actually invoked. It is necessary to pass the `-I +bisect` option to the compiler because instrumentation adds calls to functions defined in the runtime modules of Bisect.

Code sample 1 Compiling and instrumenting a file (using camlp4).

```
ocamlc -c -I +bisect -pp 'camlp4o str.cma /path/to/bisect_pp.cmo' source.ml
```

Code sample 2 Compiling and instrumenting a file (using ppx).

```
ocamlc -c -I +bisect -ppx '/path/to/bisect_ppx.byte' source.ml
```

Code sample 3 Compiling and instrumenting a file through `ocamlfind`.

```
ocamlfind ocamlc -package bisect -syntax camlp4o -c source.ml
```

---

[1]This file will be stored in the very same directory as the `cmo`, `cmx`, or `cmj` file produced by the compiler.

Note: the use of `camlp4o` implies that the OCaml grammar is slightly modified. Most notably, `camlp4o` enables quotation by default. Practically, this means that characters sequences such as `<<` or `>>` now delimit quotations. This mechanism can be disabled by passing the `-no_quot` command-line switch to `camlp4o`.

3.1.1 Instrumentation mode

Since version 1.1, it is possible to select an instrumentation *mode* through the `-mode` command-line switch followed by one these values:

- `safe`, that will perform instrumentation as done in version 1.0;

- `fast`, that will perform instrumentation in order to allow instrumented code to run faster (by approximately an order of magnitude, relatively to `safe`);

- `faster`, that will perform instrumentation in order to allow instrumented code to run even slightly faster but will not be thread-safe, even if linked with the `BisectThread` module (see below).

3.1.2 Controlling instrumentation

By language constructs

It is possible to choose which language constructs should be instrumented by passing `-enable` and/or `-disable` command-line switches to either `bisect_pp.cmo` or `bisect_pp.byte`. Both switches are followed by a string describing the kinds of points the user wants to either enable or disable. The possible characters are:

- `b` for *binding*

- `s` for *sequence*

- `f` for *for*

- `i` for *if/then*

- `t` for *try*

- `w` for *while*

- `m` for *match/function*

- `c` for *class expression*

- `d` for *class initializer*

- `e` for *class method*

- `v` for *class value*

- `p` for *toplevel expression*

- `l` for *lazy operator*

By default, all point kinds are enabled. As an example, `-disable cdev` will disable instrumentation of all class constructs.

By excluding top-level values

Since version 1.1, the `-exclude` command-line switch allows to exclude top-level values from instrumentation. It should be followed by a comma-separated list of patterns[2]. Any top-level function matching one of the patterns will not be instrumented.

Since version 1.2, the `-exclude-file` command-line switch allows to exclude top-level values whose list is stored in a file. The contents of the file should respect the following grammar:

```
contents ::= file_list
file_list ::= file_list file | ϵ
file ::= file string [ exclusion_list ] opt_separator
opt_separator ::= ; | ϵ
exclusion_list ::= exclusion_list exclusion | ϵ
exclusion ::= name string opt_separator | regexp string opt_separator
```

By special comments

Since version 1.1, it is also possible to use *special* comments in order to precisely control instrumentation on a code area basis. The following comments are recognized:

- `(*BISECT-MARK*)` and `(*BISECT-VISIT*)` allow to consider (all the points of) a line as visited even if not at runtime[3];

- `(*BISECT-IGNORE*)` allows to ignore the line, that is generate no point for the whole line;

- `(*BISECT-IGNORE-BEGIN*)` and `(*BISECT-IGNORE-END*)` (that should be correctly parenthesized) allow to exclude whole parts of the source from being instrumented (equivalent to have `(*BISECT-IGNORE*)` comments on each line from `(*BISECT-IGNORE-BEGIN*)` to `(*BISECT-IGNORE-END*)`, both inclusive).

3.1.3   Unsafe compilation mode

When compiling in *unsafe* mode[4], the `-unsafe` switch should be passed to camlp4 instead of the compiler. Indeed, as camlp4 is building a syntax tree that is passed to the compiler, issuing the `-unsafe` switch to the compiler has no effect because it is too late: the code has been built by camlp4 in *safe* mode. In such a case, the compiler warns the user with the following message: `Warning:  option -unsafe used with a preprocessor returning a syntax tree.` The correct command-line invocations are shown by code samples 4 and 5.

3.1.4   Linking

Linking a program containing instrumented modules is not different from *classical* linking, except that one should link the Bisect library to the produced executable. This is usually done by adding one of the following to the linking command-line:

---

[2]These patterns should follow the conventions set by the `Str` module.

[3]It may be useful to avoid a lower coverage due to a line containing *e. g.* `assert false`.

[4]One should keep in mind that the usefulness of using the *unsafe* mode in an instrumented application is questionable, as the instrumentation of an application results in very degraded performances.

Code sample 4 Compiling and instrumenting a file using unsafe mode.

```
ocamlc -c -I +bisect \
  -pp 'camlp4o str.cma -unsafe /path/to/bisect_pp.cmo' \
  source.ml
```

Code sample 5 Compiling and instrumenting a file using unsafe mode through `ocamlfind`.

```
ocamlfind ocamlc -package bisect -syntax camlp4o -ppopt -unsafe -c source.ml
```

- `-I +bisect bisect.cma` (for `ocamlc` version);

- `-I +bisect bisect.cmxa` (for `ocamlopt` version);

- `-I +bisect bisect.cmja` (for `ocamljava` version).

In order to use Bisect in multithread applications, it is necessary to also link with the `BisectThread` module. This also implies to pass the `-linkall` option to the compiler.

3.2 Running the instrumented application

Running an instrumented application is not different from running any application compiled with an OCaml compiler. However, Bisect will produce runtime data in a file each time the application is run. A new file will be created at each invocation, the first one being `bisect0001.out`, the second one `bisect0002.out`, and so on. It is also possible to define the scheme used for file names by setting the `BISECT_FILE` environment variable. If `BISECT_FILE` is equal to *file*, files will be named *file$n$.out* where n is a natural number value padded with zeroes to 4 digits (*i. e.* "0001", "0002", and so on).

Bisect can also be parametrized using another environment variable: `BISECT_SILENT`. If this variable is set to either "YES" or "ON" (defaulting to "OFF", case being ignored), then Bisect will not output any message at runtime. If not silent, Bisect will output a message on the standard error in various situations such as:

- the output file for runtime data cannot be created at program initialization;

- the runtime data cannot be written at program termination.

3.3 Generating the coverage report

3.3.1 By simply merging all data

In order to generate the coverage report for the instrumented application, it is sufficient to invoke the `bisect-report` executable (alternatively either `bisect-report.opt`, or `bisect-report.jar`). This program recognizes the following command-line switches:

- `-bisect <file>` Set output to bisect, data being written to given file

- `-combine-expr <expr>` Combine file according to given expression to produce data

- `-csv <file>` Set output to csv, data being written to given file

- `-dump <file>` Set output to bare dump, data being written to given file

- `-dump-dtd <file>` Dump the DTD to the given file

- `-html <dir>` Set output to html, files being written in given directory

- `-I <dir>` Add the directory to the search path

- `-no-folding` Disable code folding (HTML only)

- `-no-navbar` Disable navigation bar (HTML only)

- `-separator <string>` Set the separator for generated output (CSV only)

- `-summary-only` Output only a summary (text only)

- `-tab-size <int>` Set tabulation size in output (HTML only)

- `-text <file>` Set output to text, data being written to given file

- `-title <string>` Set the title for generated output (HTML only)

- `-verbose` Set verbose mode

- `-version` Print version and exit

- `-xml <file>` Set output to xml, data being written to given file

- `-xml-emma <file>` Set output to EMMA xml, data being written to given file

Wherever a destination file is waited, the use of `-` (*i. e.* minus sign) is interpreted as the standard output. The user should also provide on the command-line the list of the runtime data files that should be used to produce the report. As a result, a typical invocation is: `bisect-report -html report bisect*.out` to process all data files in the current directory and generate an HTML report into the `report` directory.

If relative file paths are used at the instrumentation step, the report executable should be launched from the same directory. Another option is of course to use absolute paths. Using absolute paths is also useful when playing with the `-pack` option. Indeed, it is possible in this case to have several source files with the same name in different directories and packed to different enclosing modules. In the case of packed modules, absolute paths allows to avoid ambiguities but are not necessary. It is in fact sufficient to have *discriminating* paths, that is: paths that always allow to distinguish files packed in different enclosing modules. It is also possible to use the `-I` command-line switch to specify a search path for source files.

When the HTML output mode is chosen, a bunch a files is produced: one `index.html` file, and one HTML file per instrumented module. The `index.html` file provides application-wide statistics about coverage, as well as links to the other files. The module files provide module-wide statistics, as well as a duplicate of the module source, enhanced with *point* information. *Points* are represented in the source as special comments having the form `(*[n]*)` where n indicates how many times the *point* was passed at runtime. For easier appreciation, colors are also used to annotate source lines:

- a line will be green-colored if it contains *points* whose values are all strictly positive;

- a line will be red-colored if it contains *points* whose values are all equal to zero;

- a line will be yellow-colored if it contains some *points* whose values are all equal to zero, and some others whose values are strictly positive;

- a line will not be colored at all if it contains no *point*.

When another output mode is chosen, only one file is produced (or none, if - is used) containing the whole coverage information. The appendix details the various file formats.

3.3.2 By defining how data sets should be combined

Since version 1.2, it is possible to perform some computation on data files. The aforementioned command-line `bisect-report -html report bisect*.out` combines the data of all files matching the `bisect*.out` pattern, but it may be useful to specify how data should be combined. This is done through the `-combine-expr` command-line switch that should be followed by an expression. Using this switch is intended to replace the list of files to process, leading to the command-line `bisect-report -html report -combine-expr 'expr'`.

The expression should be well-formed according to the following grammar:

```
expr      ::= expr binop expr | ( expr ) | func_name ( expr ) | value
binop     ::= + | - | * | /
func_name ::= sum | nonnull
value     ::= single_file | file_set | integer
```

where:

- a single file is given by its path between quotes (*e. g.* `"file"`);

- a file set is given by its pattern between angle brackets (*e. g.* `<file*.out>`);

- `+` and `-` allow to respectively add and subtract point values from two single files;

- `*` and `/` allow to respectively multiply and divide point values from a single file by an integer;

- function `nonnull` allows to replace every point value that is different from `0` by `1`;

- function `sum` allows to convert values from a file set to make them appear as coming from a single file.

Using `-combine-expr` permits sophisticated analysis of program runs, thus allowing fine-grained debugging. Suppose that you are able to produce two runs of a program, one exhibiting a bug and the other one not exhibiting it. The expression

$$\texttt{notnull("first.out") + notnull("second.out")*2}$$

will produce a report where;

- a point value of `0` means that no run evaluated the related expression;

- a point value of `1` means that only the first run evaluated the related expression;

- a point value of 2 means that only the second run evaluated the related expression;

- a point value of 3 means that both run evaluated the related expression.

It then far easier to spot the area where the bug stems from.

Chapter 4

Complete example

Code sample 6 shows the makefile used for the compilation (with instrumentation), run, and report phases for a one-file application: `source.ml`. Code sample 7 shows the same information when relying on `ocamlfind`.

Code sample 6 Example makefile.

```
default: clean compile run report

clean:
        rm -fr report
        rm -f *.cm* *.out bytecode

compile:
        ocamlc -c -I +bisect \
            -pp "camlp4o str.cma `ocamlc -where`/bisect/bisect_pp.cmo" source.ml
        ocamlc -o bytecode -I +bisect bisect.cma source.cmo

run:
        BISECT_FILE=coverage ./bytecode

report:
        bisect-report -dump - -html report coverage*.out
```

It is also possible to compile the `source.ml` file through the `ocamlbuild` tool. The most convenient way is to first define a new `bisect` tag in a `myocamlbuild.ml` plugin. This tag will add the necessary elements when compiling or linking a file using the Bisect features, as shown by code sample 8. Then, it is sufficient to use the newly introduced tag in the `_tags` file to use bisect, as shown by code sample 9.

Finally, `ocamlbuild` can also leverage `ocamlfind`, leading to the following command-line invocation: `ocamlbuild -use-ocamlfind -tag 'package(bisect)' -tag 'syntax(camlp4o)' -tag 'syntax(bisect_pp)' source.byte`.

Code sample 7 Example makefile (`ocamlfind`-based).

```
default: clean compile run report

clean:
        rm -fr report
        rm -f *.cm* *.out bytecode

compile:
        ocamlfind ocamlc \
            -package bisect -linkpkg -syntax camlp4o -o bytecode source.ml

run:
        BISECT_FILE=coverage ./bytecode

report:
        ocamlfind bisect/bisect-report -html report coverage*.out
```

Code sample 8 `myocamlbuild.ml` plugin file.

```
open Ocamlbuild_plugin
open Ocamlbuild_pack

let () =
  dispatch begin function
    | After_rules ->
        flag ["bisect"; "pp"]
          (S [A"camlp4o"; A"str.cma"; A"/path/to/bisect/bisect_pp.cmo"]);
        flag ["bisect"; "compile"]
          (S [A"-I"; A"/path/to/bisect"]);
        flag ["bisect"; "link"; "byte"]
          (S [A"-I"; A"/path/to/bisect"; A"bisect.cma"]);
        flag ["bisect"; "link"; "native"]
          (S [A"-I"; A"/path/to/bisect"; A"bisect.cmxa"]);
        flag ["bisect"; "link"; "java"]
          (S [A"-I"; A"/path/to/bisect"; A"bisect.cmja"])
    | _ -> ()
end
```

Code sample 9 `_tags` file.

```
<source.*>: bisect
```

Chapter 5

Known issues

Bisect suffers from the following issues:

- Bisect, being based on either camlp4 or ppx, performs a purely syntactic treatment. It can thus sometimes produce unaccurate results due to semantics subtleties. For a concrete example consider lazy operators: in expressions such as `e1 && e2` or `e1 || e2`, Bisect adds *points* to both *e1* and *e2* to allow the user to know which parts of the whole expression were actually evaluated. However, it is possible that the programmer redefined one of these operator in such a way that its new semantics is no more lazy (*e. g.* `let (&&) = (+)`). In this case, Bisect will still add points to subexpressions even if they appear useless[1]. A dual issue would occur if a programmer defined a new operator with lazy semantics (*e. g.* `external (++) :  bool -> bool -> bool = "%sequor"`), in this case Bisect will not define *points* for subexpressions while they would clearly be of interest.

- when linking the tested application, the `Bisect` module should be linked as (one of) the first ones; indeed, the Bisect runtime performs some operations at initialization, such as determining the target file for runtime information: the current working directory should hence not have been modified by another module or should have been modified purposely (it is also possible to use `BISECT_FILE` to specify an absolute path);

- for performance reasons, OCaml `int`s are used store store *point* data; it implies that one should not use the report executable on a 32-bit architecture if the tested application has been instrumented and run on a 64-bit architecture[2].

---

[1]One may notice that it could not be possible to overcome this problem by keeping track of local (*i. e.* file) redefinitions, as the redefinition may occur in another module that has been opened.

[2]This is indeed an over-cautious recommendation, as the OCaml runtime gracefully handles platforms differences; one should only get inaccurate results (but not false results: neither an unvisited will be considered as visited, nor the opposite) when working at the 32-bit limit.

Appendix A

File formats

A.1 CSV file format

The CSV mode outputs statistics line by line: first for the whole application, and then for each file. Each line has the following format: first the path of the source file (- being used for the overall application), then 14×2 integer values (13 for the various point kinds, plus one for the total). Each integer couple consists, for each point kind, of (i) the number of visited points and (ii) the total number of points. The point kinds are output in the following order: *let bindings, sequence, for loops, if/then constructs, try/with constructs, while loops, match/function constructs, class expressions, class initializers, class methods, class values, top level expressions, lazy operators.* Code sample 10 shows such an output.

Code sample 10 CSV file format.

```
-;3;3;5;5;1;1;0;0;0;0;0;0;2;2;0;0;0;0;0;0;0;0;0;0;2;2;13;13
source.ml;3;3;5;5;1;1;0;0;0;0;0;0;2;2;0;0;0;0;0;0;0;0;0;0;2;2;13;13
```

A.2 Text file format

The text mode outputs statistics first for the overall application, and then for each file. The statistics always take the same form, that is the ratio *number of visited points over total number of points* for each point kind, followed by the ratio for all point kinds. Code sample 11 shows such an output.

A.3 XML file format

The XML mode outputs both statistics and information for each of the points in the source files. Code sample 12 shows the DTD for produced XML files (it can be generated using the **-dump-dtd** command-line option). Statistics are output for the whole application and for each file inside **<summary** elements, while information relative to each point is encoded into **<point** elements. Code sample 13 shows an XML output.

Code sample 11 Text file format.

```
Summary:
 - 'binding' points: 3/3 (100.00 %)
 - 'sequence' points: 5/5 (100.00 %)
 - 'for' points: 1/1 (100.00 %)
 - 'if/then' points: none
 - 'try' points: none
 - 'while' points: none
 - 'match/function' points: 2/2 (100.00 %)
 - 'class expression' points: none
 - 'class initializer' points: none
 - 'class method' points: none
 - 'class value' points: none
 - 'toplevel expression' points: none
 - 'lazy operator' points: 2/2 (100.00 %)
 - total: 13/13 (100.00 %)
File 'source.ml':
 - 'binding' points: 3/3 (100.00 %)
 - 'sequence' points: 5/5 (100.00 %)
 - 'for' points: 1/1 (100.00 %)
 - 'if/then' points: none
 - 'try' points: none
 - 'while' points: none
 - 'match/function' points: 2/2 (100.00 %)
 - 'class expression' points: none
 - 'class initializer' points: none
 - 'class method' points: none
 - 'class value' points: none
 - 'toplevel expression' points: none
 - 'lazy operator' points: 2/2 (100.00 %)
 - total: 13/13 (100.00 %)
```

Code sample 12 DTD for produced XML files.

```
<!ELEMENT bisect-report (summary,file*)>

<!ELEMENT file (summary,point*)>
<!ATTLIST file path CDATA #REQUIRED>

<!ELEMENT summary (element*)>

<!ELEMENT element EMPTY>
<!ATTLIST element kind CDATA #REQUIRED>
<!ATTLIST element count CDATA #REQUIRED>
<!ATTLIST element total CDATA #REQUIRED>

<!ELEMENT point EMPTY>
<!ATTLIST point offset CDATA #REQUIRED>
<!ATTLIST point count CDATA #REQUIRED>
<!ATTLIST point kind CDATA #REQUIRED>
```

Code sample 13 XML file format.

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<bisect-report>
  <summary>
    <element kind="binding" count="1" total="1"/>
    <element kind="sequence" count="0" total="0"/>
    <element kind="for" count="0" total="0"/>
    <element kind="if/then" count="0" total="0"/>
    <element kind="try" count="0" total="0"/>
    <element kind="while" count="0" total="0"/>
    <element kind="match/function" count="0" total="0"/>
    <element kind="class expression" count="0" total="0"/>
    <element kind="class initializer" count="0" total="0"/>
    <element kind="class method" count="0" total="0"/>
    <element kind="class value" count="0" total="0"/>
    <element kind="toplevel expression" count="0" total="0"/>
    <element kind="lazy operator" count="0" total="0"/>
    <element kind="total" count="1" total="1"/>
  </summary>
  <file path="source.ml">
    <summary>
      <element kind="binding" count="1" total="1"/>
      <element kind="sequence" count="0" total="0"/>
      <element kind="for" count="0" total="0"/>
      <element kind="if/then" count="0" total="0"/>
      <element kind="try" count="0" total="0"/>
      <element kind="while" count="0" total="0"/>
      <element kind="match/function" count="0" total="0"/>
      <element kind="class expression" count="0" total="0"/>
      <element kind="class initializer" count="0" total="0"/>
      <element kind="class method" count="0" total="0"/>
      <element kind="class value" count="0" total="0"/>
      <element kind="toplevel expression" count="0" total="0"/>
      <element kind="lazy operator" count="0" total="0"/>
      <element kind="total" count="1" total="1"/>
    </summary>
    <point offset="11" count="1" kind="binding"/>
  </file>
</bisect-report>
```

A.4   XML EMMA-compatible format

This mode outputs only overall statistics, in a format that is compatible with EMMA[1]. This compatibility allows to use Bisect output in tools that provide support for EMMA, notably giving an easy way to use Bisect with continuous integration servers like Jenkins.

EMMA defines only four categories for coverage: classes, methods, blocks, and lines. Bisect defining more point kinds, the following mapping is used:

- *class expressions*, *class initializers*, and *class values* are merged into the *class* category;

- *class methods* are mapped to the *method* category;

- *let bindings*, *sequence*, *for loops*, *if/then constructs*, *try/with constructs*, *while loops*, *match/function constructs*, and *lazy operators* are merged into the *block* category;

- *top level expressions* are mapped to the *line* category.

Another element should be noted regarding this output mode: for all the categories, any 0/0 value is replaced by a 1/1 value. This replacement is justified by the fact that 0/0 may result in 0% while 1/1 results in 100%, and one would not want to have a build failure in Jenkins due to low coverage. Code sample 14 shows an EMMA-compatible XML output.

Code sample 14 XML EMMA file format.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<report>
  <stats>
    <packages value="1"/>
    <classes value="1"/>
    <methods value="1"/>
    <srcfiles value="1"/>
    <srclines value="1"/>
  </stats>
  <data>
    <all name="all classes">
      <coverage type="class, %" value="100% (1/1)"/>
      <coverage type="method, %" value="100% (1/1)"/>
      <coverage type="block, %" value="100% (1/1)"/>
      <coverage type="line, %" value="100% (1/1)"/>
    </all>
  </data>
</report>
```

A.5   Dump format

The *dump* format is mainly used for debugging, only displaying the various points and their associated counts for each file. Code sample 15 shows such a dump.

---

[1]EMMA is a code coverage tool for Java - http://emma.sourceforge.net/

Code sample 15 *Dump* format.

```
file "source.ml"
  point              sequence at offset     17:      1
  point              sequence at offset     42:      1
  point                   for at offset     64:      5
  point              sequence at offset    118:      1
  point              sequence at offset    144:      1
  point                   for at offset    166:      3
  point        match/function at offset    253:      1
  point        match/function at offset    278:      1
  point        match/function at offset    297:      0
```