Bolt 1.4

Reference Manual

http://bolt.x9c.fr

October 26, 2012

Contents

Chapter 1

Overview

1.1 Purpose

Bolt is a logging tool for the OCaml language[1]. Its name stems from the following acronym: *Bolt is an Ocaml Logging Tool*. It is inspired by and modeled after the Apache log4j utility[2]. Bolt provides both a comprehensive library for log production, and a camlp4-based syntax extension that allows to remove log directives. The latter is useful to be able to distribute an executable that incurs no runtime penalty if logging is used only during development.

The importance of logging is frequently overlooked but (quite ironically), in the same time, the most used debugging *method* is by far the `print` statement. Bolt aims at providing OCaml developers with a framework that is comprehensive, yet easy to use. It also tries to leverage the benefits of both compile-time and run-time configuration to produce a flexible library with a manageable computational cost.

1.2 License

Bolt is distributed under the terms of the LGPL version 3. This licensing scheme allows to use Bolt in any software, not contaminating code.

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 bolt@x9c.fr.

[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] `http://logging.apache.org/log4j`

Chapter 2

Building Bolt

2.1 Step 0: dependencies

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

- OCaml, version 4.00.0;
- `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 Bolt 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.

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.

2.3 Step 2: compilation

The actual build of Bolt 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`.

[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

2.4 Step 3: installation

Bolt 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‘/bolt` (*i. e.* `$PREFIX/lib/ocaml/bolt`).

Chapter 3

Concepts and terminology

3.1 Logger

The central concept of Bolt is the one of loggers. Loggers have names that are strings composed of dot-separated components; they are thus akin to module names, and it is actually good practice to use the logger `M` to log events of the module `M`. It is possible to register several loggers with the same name; this feature is useful to record the events related to a given module to several different destinations (using possibly different filters, layout, and outputs).

Loggers are also organized into a hierarchy (meaning that logger `P` is a parent of logger `P.S`). When a log statement is executed, it is associated with a logger name. Figure 3.1 shows the hierarchy of loggers for an application using the loggers whose name appears in black. The loggers whose names appear in gray are implicitly added by Bolt in order to have a complete tree of loggers: those actually used in the program are the leaves, and the root is the special `""` (*i. e.* empty name) logger. The arrows define the is-a-child-of relation.

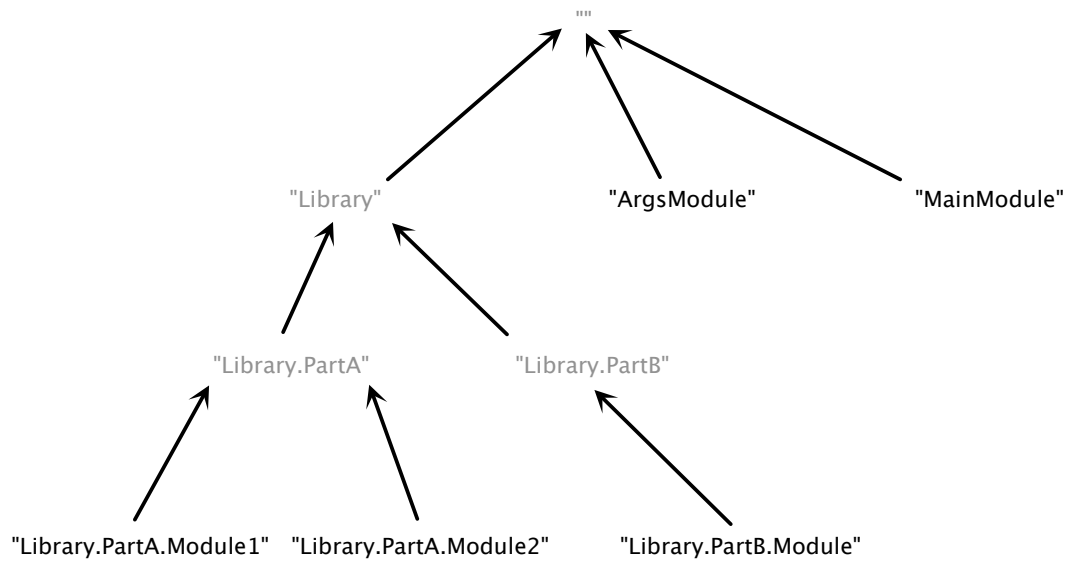

Figure 3.1: Example of a logger hierarchy.

3.2 Event

Events are the entities built each time the running program executes a log statement. The event carries all the information needed for efficient logging: message, location, time, logger name, *etc.* Some information is explicitly provided by the user (*e. g.* message or properties), while some information is implicitly computed by the runtime (*e. g.* time or process identifier).

3.3 Level

Level characterizes how critical an event is. Each logger has as associated level that indicates which levels it is interested in. An event will be recorded iff its level is below the level of logger. The levels are, in ascending order:

- `FATAL` for errors leading to program termination;
- `ERROR` for errors handled by the program;
- `WARN` for for hazardous circumstances;
- `INFO` for coarse-grained information;
- `DEBUG` for debug information;
- `TRACE` for fine-grained information.

3.4 Filter

A filter is basically a predicate over events, allowing to determine whether an event should be recorded by a condition on any element of the events. Each logger has an associated filter, ensuring that only the events satisfying the filter will be recorded.

3.5 Layout

Each logger has an associated layout that is responsible for the conversion of events into bare strings that can then be easily manipulated.

3.6 Mode

Each logger has an associated mode that is reponsible for sending data to the output. A mode allows to buffer data, in order to mitigate performances issues related to logging.

3.7 Output

Each logger has an associated output that defines where event are actually recorded. An output is thus responsible for the storage of events, once they have been converted into strings by a layout. The most simple output is the file, and in this case, safety commands that two loggers should not have the same destination.

3.8 Event dispatch

Every log event will be presented to all logger with that name, and to all loggers with a parent name. Each logger will decide according to its level and filter if the event should actually be recorded. Finally, all events are presented to all loggers having the special empty name (corresponding to the string `""`). The hierarchy of the loggers is a key feature that allows to easily enable or disable logging for large parts of an application. Figure 3.2 shows how a message initially created for the `Library.PartB.Module` loggers is dispatched to all loggers with parent names, including loggers that are not explicitly used in the application (those whose name appears in gray). The dashed arrows show the order in which the event is presented to the different loggers.

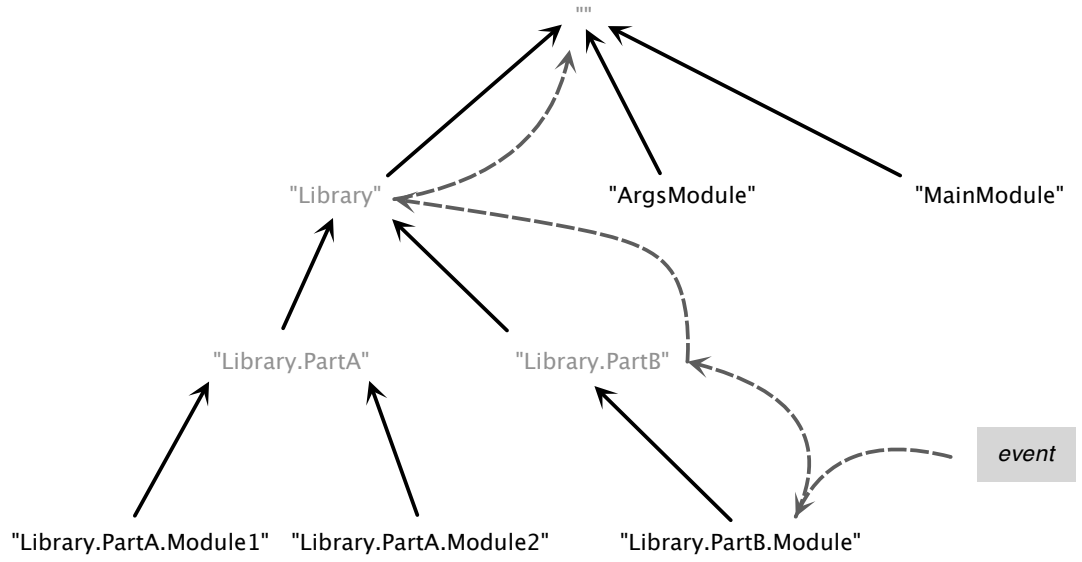

Figure 3.2: Dispatch of an event generated for the "`Library.PartB.Module`" logger.

Chapter 4

Using Bolt

4.1 Linking with the library

Linking with Bolt is usually done by adding one of the following library to the linking command-line:

- `-I +bolt bolt.cma` (for `ocamlc` compiler);
- `-I +bolt bolt.cmxa` (for `ocamlopt` compiler);
- `-I +bolt bolt.cmja` (for `ocamljava` compiler).

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

4.2 Adding log statements

There are two ways to add a log statement: either by calling explicitly the `Bolt.Logger.log` function, or by using the `bolt_pp.cmo` camlp4 syntax extension. One is advised to use the latter method: first, using the syntax extension is lightweight (elements such as line and column are automatically computed); second, it allows to remove the log statements at compilation. Indeed, it may be useful to have a development version packed with a lot of debug log statements and a distributed version that suffers no runtime penalty related to logging. Moreover, only given log statements may be removed, on a level basis.

4.2.1 Explicit logging

To log using the `Bolt.Logger.log` function, one has to call it with the following parameters (*cf.* code sample 1):

- a `string` parameter giving the name of the logger to use;
- a `Bolt.Level.t` parameter giving the level of the event to log;
- an optional `string` parameter (named *file*) giving the file associated with the log event;
- an optional `int` parameter (named *line*) giving the line number associated with the log event;

- an optional `int` parameter (named *column*) giving the column number associated with the log event;
- an optional `(string * string) list` parameter (named *properties*) giving the property list associated with the log event;
- an optional `exn option` parameter (named *error*) giving the exception associated with the log event;
- a `string` parameter giving the message associated with the log event.

Code sample 1 Explicit logging.

```
let () =
  ...
  Bolt.Logger.log "mylogger" Bolt.Level.DEBUG "some debug info";
  ...
```

4.2.2 Implicit logging

To log using the syntax extension, one has to use the Bolt-introduced `LOG` expression. This is done by passing the `-pp 'camlp4o /path/to/bolt_pp.cmo'` option to the OCaml compiler. The new `LOG` expression can be used in an OCaml program wherever an expression of type `unit` is waited. The BNF definition of this expression is as follows:

log_expr ::= LOG (*string* | *ident*) *arguments* *attributes* LEVEL *level*
arguments ::= *list of expressions* | ϵ
attributes ::= *attributes* *attribute* | ϵ
attribute ::= NAME *string* | (PROPERTIES | WITH) *expr* | (EXCEPTION | EXN) *expr*
level ::= FATAL | ERROR | WARN | INFO | DEBUG | TRACE

The string following the `LOG` keyword is the message of the log event, it can be either a literal string or an identifier whose type is string. This string can be followed by expressions; in this case the string is interpreted as a `printf` format string, using the following expressions as values for the `%` placeholders of the format string.

The attributes are optional, and have the following meaning:

- `NAME` defines the name of the logger to be used;
- `PROPERTIES` defines the properties associated with the log event (the expression should have the type `(string * string) list`);
- `EXCEPTION` defines the exception associated with the log event (the expression should have type `exn`).

Code sample 2 shows how the expression can be used. Compared to explicit logging through the `Bolt.Logger.log`, when using the `LOG` expression file, line number, and column number are determined automatically.

When no `NAME` attribute is provided, the logger name is computed from the source file name: the `.ml` suffix is removed and the result is capitalized. More, the `bolt_pp.cmo` syntax extension accepts the following parameters:

- `-logger <n>` sets the logger name to n for all `LOG` expressions of the compiled file;
- `-for-pack <P>` sets the prefix to the logger names used throughout the compiled file to "`P.`".

Finally, the `bolt_pp.cmo` syntax extension recognizes a third parameter `-level <l>` where l should be either `NONE` or a level. If l is `NONE`, all `LOG` expressions will be removed from the source file; otherwise, only the `LOG` expression with a level inferior or equal to the passed value will be kept. This means that `TRACE` will keep all log statements, while `ERROR` will keep only log statements with a level equal to either `ERROR`, or `TRACE`.

Code sample 2 Implicit logging.

```
let () =
  ...
  LOG "some debug info" LEVEL DEBUG;
  ...
```

Note: when compiling in *unsafe* mode, 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 Bolt invocation is hence `-pp 'camlp4o -unsafe /path/to/bolt_pp.cmo'`.

4.3 Configuring log

There are two ways to configure log, that is to register loggers that will handle the log events produced by the application. The first way is to explicitly call `Bolt.Logger.register` while the second one is to use a configuration file that will be interpreted by Bolt at runtime.

To register (*i.e.* to create) a logger using the `Bolt.Logger.register` function, one has to call it with the following parameters:

- a `string` parameter giving the name of the logger;
- a `Bolt.Level.t` parameter giving the maximum level for events to be logged;
- a `string` parameter giving the filter of the logger;
- a `string` parameter giving the layout of the logger;
- a `Mode.t` parameter giving the mode of the logger;
- a `string` parameter giving the output of the logger;

- a `string * float option` couple that gives the parameters used for output creation: the first component is the name of the output while the second one is the optional rotate value (the actual semantics of both component is dependent on the output actually used).

To register a logger using a configuration file, one should set either the `BOLT_FILE` or the `BOLT_CONFIG` environment variable to the path of the configuration file. `BOLT_FILE` is to be used when the file is written in the old configuration format, while `BOLT_CONFIG` is to be used when the file is written in the new configuration format. If the configuration file cannot be loaded, an error message is written on the standard error unless the `BOLT_SILENT` environment variable is set to either "YES" or "ON" (defaulting to "OFF", case being ignored).

The old format of the configuration file is as follows:

- the format is line-oriented;
- comments start with the '`#`' character and end at the end of the line;
- sections start with a line of the form `[a.b.c]`, "`a.b.c`" being the name of the section;
- a section ends when a new section starts;
- at the beginning of the file, the section named "" is currently opened;
- section properties are defined by lines of the form "`key=value`";
- others lines should be empty (only populated with whitespaces and comments).

The new format of the configuration file is defined by the following grammar:

```
file ::= section_list
section_list ::= section_list section | ϵ
section ::= logger string { property_list } opt_separator
opt_separator ::= ; | ϵ
property_list ::= property_list property | ϵ
property ::= ident = property_value opt_separator
property_value ::= expr | integer | float | string
expr ::= ident | ident && ident | ident || ident
```

In both formats, each section defines a logger whose name is the section name. The following properties are used to customize the logger:

- `level` defines the level of the logger;
- `filter` defines the filter of the logger;
- `layout` defines the layout of the logger;
- `mode` defines the mode of the logger;
- `output` defines the output of the logger;
- `name` is the first parameter passed to create the actual output;

- `rotate` is the second parameter passed to create the actual output;
- `signal` indicates a signal that will trigger a rotation (possible values are `sighup`, `sigusr1`, and `sigusr2`).

The level can have one of the following values: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. The possible values for the other properties are discussed in the following sections.

The mode can have one of the following values:

- `direct`, meaning that data is sent to output as soon at it is available;
- `memory`, meaning that data is buffered and sent to output only at program termination;
- `retained`, meaning that data is buffered and periodically sent to output.

When `retained` mode is used, the `retention` parameter defines when data is actually sent to output. The string value can have one of the following form:

- an integer followed by the letter `'e'` (e. g. `"10e"`) means that events are delivered when n events have been accumulated;
- an integer followed by the letter `'b'` (e. g. `"10b"`) means that events are delivered when a string representation of n bytes has been accumulated;
- an integer followed by the letter `'s'` (e. g. `"10s"`) means that events are delivered every n seconds.

Code sample 3 and 4 exemplify typical configuration files, respectively in old and new format. They define three loggers (with names "", "`Pack.Main`", and "`Pack.Aux`"). When executed, the application will produce three files "`bymodule.result`", "`bymodule1.result`", and "`bymodule2.result`": the first file will contain the log information for the whole application while the other ones will contain respectively the log information associated with the "`Pack.Main`" and "`Pack.Aux`" loggers.

4.3.1 Predefined filters

The following filters are predefined:

- `all` keeps all events;
- `none` keeps no event;
- `trace_or_below` keeps events with level inferior or equal to `TRACE`;
- `debug_or_below` keeps events with level inferior or equal to `DEBUG`;
- `info_or_below` keeps events with level inferior or equal to `INFO`;
- `warn_or_below` keeps events with level inferior or equal to `WARN`;
- `error_or_below` keeps events with level inferior or equal to `ERROR`;
- `fatal_or_below` keeps events with level inferior or equal to `FATAL`;

Code sample 3 Example of configuration file (old format).

```
level=trace
filter=all
layout=simple
mode=direct
output=file
name=bymodule.result

[Pack.Main]
level=trace
filter=all
layout=simple
mode=direct
output=file
name=bymodule1.result

[Pack.Aux]
level=trace
filter=all
layout=simple
mode=direct
output=file
name=bymodule2.result
```

Code sample 4 Example of configuration file (new format).

```
logger "" {
  level = trace;
  filter = all;
  layout = simple;
  mode = direct;
  output = file;
  name = "bymodule.result";
}

logger "Pack.Main" {
  level = trace;
  filter = all;
  layout = simple;
  mode = direct;
  output = file;
  name = "bymodule1.result";
}

logger "Pack.Aux" {
  level = trace;
  filter = all;
  layout = simple;
  mode = direct;
  output = file;
  name = "bymodule2.result";
}
```

- `trace_or_above` keeps events with level superior or equal to `TRACE`;
- `debug_or_above` keeps events with level superior or equal to `DEBUG`;
- `info_or_above` keeps events with level superior or equal to `INFO`;
- `warn_or_above` keeps events with level superior or equal to `WARN`;
- `error_or_above` keeps events with level superior or equal to `ERROR`;
- `fatal_or_above` keeps events with level superior or equal to `FATAL`;
- `file_defined` keeps events with an actual filename;
- `file_undefined` keeps events with no filename;
- `line_defined` keeps events with a strictly positive line number;
- `line_undefined` keeps events with a negative or null line number;
- `column_defined` keeps events with a strictly positive column number;
- `column_undefined` keeps events with a negative or null column number;
- `message_defined` keeps events with a non-empty message;
- `message_undefined` keeps events with an empty message;
- `message_paje` keeps events whose message is the Pajé identifier;
- `message_not_paje` keeps events whose message is not the Pajé identifier;
- `message_daikon` keeps events whose message is the Daikon identifier;
- `message_not_daikon` keeps events whose message is not the Daikon identifier;
- `properties_empty` keeps events with an empty property list;
- `properties_not_empty` keeps events with an non-empty property list;
- `exception_some` keeps events with an exception;
- `exception_none` keeps events with no exception.

4.3.2 Predefined layouts

Bolt predefines the following non-configurable layouts:

- `minimal` with format: `MESSAGE`;
- `simple` with format: `LEVEL - MESSAGE`;
- `default` with format: `TIME [FILE LINE] LEVEL MESSAGE`;
- `paje`, and `paje_noheader` whose format is the Pajé trace format[1] (the two format only differ in that the latter one does not output definitions, which is useful when one wants to merge several files) – see chapter 7 for more information;
- `daikon_decls`, and `daikon_trace` that respectively follow Daikon[2] declaration (*i.e.* pro-

[1] http://paje.sourceforge.net
[2] http://groups.csail.mit.edu/pag/daikon/

gram points, and associated variable types) and trace format (*i.e.* actual variable values for the various program points visits) – see chapter 6 for more information;

- `html` whose format is HTML, storing events into a table;
- `xml`, or `log4j` whose format is XML (compatible with log4j).

Pattern and comma-separated layouts
Two other layouts are predefined:

- `pattern` whose actual format is specified by defining a property named `pattern`
  This property is a string that can contain `$(x)` elements where x is a key (defined below) or `$(x:n)` where x is a key and n is a padding instruction (the absolute value of n is the total width; the padding is left if n is negative, and right if n is positive)
  it is also possible to specify through the `pattern-header-file` (respectively `pattern-footer-file`) property the name of a file whose contents is used as the header (respectively footer) that is written at start (respectively end) as well as at each rotation
- `csv` whose actual format is specified by properties named `csv-separator` and `csv-elements`
  `csv-separator` is the string to be used as the separator between values
  `csv-elements` is a whitespace-separated list of the keys of the values to render

The following keys are available for use by the `pattern` and `csv` layouts:

- `id` event identifier;
- `hostname` host name of running program;
- `process` process identifier of running program (*i.e.* *pid*);
- `thread` thread identifier;
- `sec` seconds of event timestamp;
- `min` minutes of event timestamp;
- `hour` hour of event timestamp;
- `mday` day of month of event timestamp;
- `month` month of year of event timestamp;
- `year` year of event timestamp;
- `wday` day of week of event timestamp;
- `time` event timestamp;
- `relative` time elapsed between initialization and event creation;
- `level` event level;
- `logger` event logger;

- `origin` first logger that received the event;
- `file` event file;
- `filebase` event file (without directory information);
- `line` event line;
- `column` event column;
- `message` event message;
- `properties` property list of event (formatted as `"[k1: v1; ...; kn: vn]"`);
- `exception` event exception;
- `backtrace` event exception backtrace.

4.3.3 Predefined outputs

There are five predefined outputs, namely `void`, `growlnotify`[3], `bell`, `say`, and `file`.

The `void` output discards all data.

The `growlnotify` output sends the data to the Growl application through the `growlnotify` command-line utility.

The `bell` output discards all data, and simply prints a "bell" character on the standard output.

The `say` output uses the MacOS X command-line utility `say` to invoke the text-to-speech engine in order to actually vocalize the data.

The `file` output writes data to a bare file, the `name` property (or the `string` value when using `Bolt.Logger.register`) defines the path of the file to be used[4], and the `rotate` property (or the `float option` value when using `Bolt.Logger.register`) gives the rate in seconds at which files will be rotated. It is also possible to use the `signal` property (set to one one the following values: `SIGHUP`, `SIGUSR1`, `SIGUSR2`) in order to request rotation upon signal reception.

When using rotation or several program instances in parallel, it is convenient for the name to contain a piece of information ensuring that the file name will be unique; otherwise, the same file will be written over and over again. In version 1.0, Bolt supported the `%` special character that was substituted by a timestamp. Since version 1.1, Bolt additionally supports a more general `$(key)` substitution mechanism with the following keys:

- `time` as a bare alternative to `%`;
- `pid` that designates the process identifier;
- `hostname` that designates the process hostname (useful when using a shared file system);
- *var* that designates any environment variable available from the process.

[3] Command-line utility associated with the Growl program available at http://growl.info/ for MacOS X, and http://www.growlforwindows.com for Windows.

[4] Two special filenames are recognized: <stdout> for standard output, and <stderr> for standard error.

Chapter 5

Reviewing generated log

Once the log information has been produced by the application, the developer and/or the user will have to review it. Obviously, the review depends on the chosen layout. When the layout is one among `simple`, `default`, or `pattern`, review can easily be done using classical Unix commands such as `grep`, `cut`, `sed`, *etc.*

When the layout is `csv`, it can be equally convenient to use either one of the aforementioned command-line tools, or any piece of software able to read CSV files such as a database, or a spreadsheet.

When the layout is `html`, the evident way to review log is to use a browser in order to have a graphical rendering of the log event. Figure 5.1 shows a small log rendered as a webpage.
When the layout is `xml`, a dedicated tool such as a GUI can be helpful. For this reason, the XML layout of Bolt produces log4j-compatible XML files allowing the use of the Apache Chainsaw application[1]. Code sample 5 shows a XML file that could be used to wrap the XML data produced by Bolt (in `bolt.xml` file) in such a way that Chainsaw can load it. This code sample is a reproduction of the one provided in the Javadoc of the log4j `org.apache.log4j.xml.XMLLayout` class[2]. Figure 5.2 shows a screenshot of Chainsaw.

[1] http://logging.apache.org/chainsaw/
[2] http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/xml/XMLLayout.html

Identifier	Time	Millis	Level	File	Message
1	07:10:37	1	TRACE	source.ml 8	application start
2	07:10:37	2	DEBUG	source.ml 2	funct(3)
3	07:10:37	2	DEBUG	source.ml 2	funct(7)
4	07:10:37	3	TRACE	source.ml 11	application end

Generated by Bolt 1.3

Figure 5.1: Reviewing an HTML log file with a browser.

Code sample 5 Wrapping produced XML data into a Chainsaw-compatible XML.

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

<!DOCTYPE log4j:eventSet SYSTEM "log4j.dtd" [<!ENTITY data SYSTEM "bolt.xml">]>

<log4j:eventSet version="1.2" xmlns:log4j="http://jakarta.apache.org/log4j/">
        &data;
</log4j:eventSet>
```

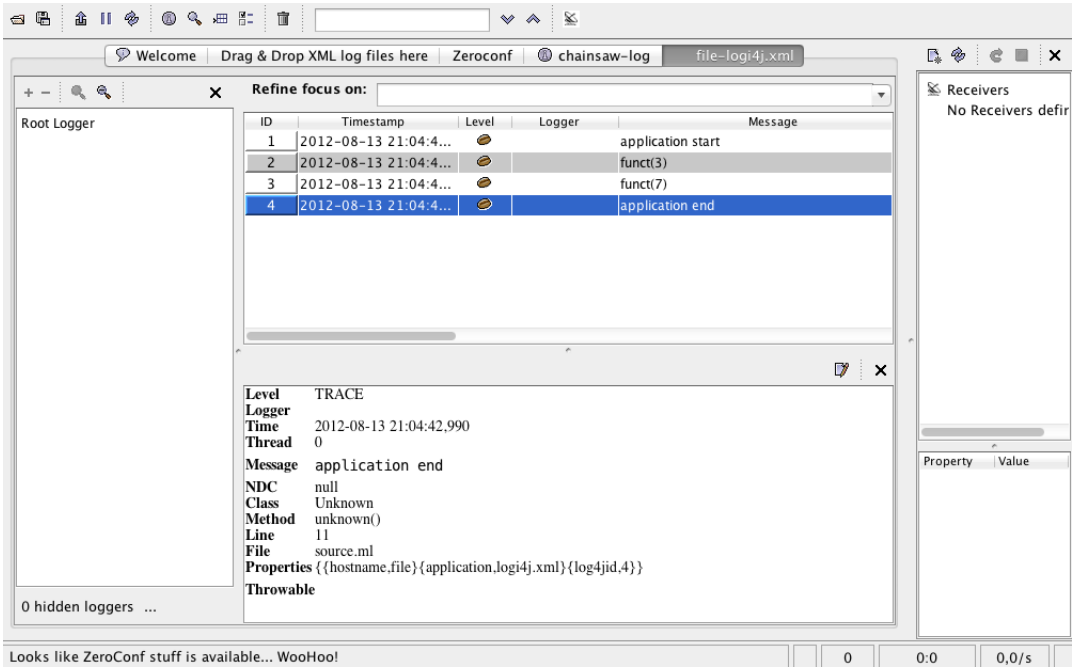

Figure 5.2: Reviewing an XML log file with Chainsaw.

Chapter 6

Daikon support

6.1 Overview

Daikon[1] is an invariant detector that detects *likely* invariants from execution traces. First, programs are instrumented in order to produce traces containing the values of variables at different points. Then, after (several) execution(s), Daikon processes the traces and outputs a list of *likely* invariants. The invariants are qualified as *likely* because Daikon is only able to assert that they hold for the given set of executions, but not for *any* execution.

6.2 Configuration

When using the Daikon tool, it is necessary to produce two elements: both the traces with all recorded variable, and a definition of these variables. To this end, Bolt uses two different layouts: `daikon_decls` for variable declaration, and `daikon_dtrace` for actual traces. Typically, this leads to a configuration file akin to the one presented by code sample 6 (old configuration format), or code sample 7 (new configuration format).

6.3 Instrumentation

As previously said, the programs need to be instrumented in order to produce traces that will be analyzed by Daikon after execution. Such instrumentation is done through logging statements with the designated `Daikon.t` value as message, and properties are used to indicate which variables should be recorded. This leads to log statements with one of the following form:

- `LOG Daikon.t WITH Daikon.enter "fn" [`*`variables`*`];` to record the enter in a function whose name is `fn` with parameters `variables`;
- `LOG Daikon.t WITH Daikon.exit "fn"` *`variable`* `[`*`variables`*`];` to record the exit from a function whose name is `fn` with result `variable` and parameters `variables`;
- `LOG Daikon.t WITH Daikon.point "pi" [`*`variables`*`];` to record the values `variables` at any source point with identifier `pi`.

Note: the Daikon layouts ignore the `EXCEPTION` part of the log statements.

[1] http://groups.csail.mit.edu/pag/daikon/

Code sample 6 Daikon configuration (old format).

```
[]
level=trace
filter=all
layout=daikon_decls
mode=direct
output=file
name=daikon.decls

[]
level=trace
filter=all
layout=daikon_dtrace
mode=direct
output=file
name=daikon.dtrace
```

Code sample 7 Daikon configuration (new format).

```
logger "" {
  level = trace;
  filter = all;
  layout = daikon_decls;
  mode = direct;
  output = file;
  name = "daikon.decls";
}

logger "" {
  level = trace;
  filter = all;
  layout = daikon_dtrace;
  mode = direct;
  output = file;
  name = "daikon.dtrace";
}
```

Values, independently of their *kind* (parameters, return values, bare variables) are encoded using a variable-building function from the `Daikon` module. All these functions take as first parameter the name of the value (as a string), and as second parameter the value itself. As of version 1.4, they are:

- t for type t;
- t`_option` for type t `option`;
- t`_list` for type t `list`;
- t`_array` for type t `array`;

where t is one of `bool`, `int`, `float`, or `string`.
It is also possible to use the combinators `tuple2`, `tuple3`, `tuple4`, `tuple5`, or `make_variable_builder` in order to group values.
Code sample 8 shows a program that has been instrumented to record values at the start and end of the `f` function with type `int -> int`, as well as at intermediary point.

Code sample 8 Daikon-instrumented program.

```
let f x =
  LOG Daikon.t
    WITH Daikon.enter "f" [Daikon.int "x" x] LEVEL TRACE;
  let tmp = x * x in
  LOG Daikon.t
    WITH Daikon.point "interm" [Daikon.int "x" x; Daikon.int "tmp" tmp] LEVEL TRACE;
  let res = tmp mod 2 in
  LOG Daikon.t
    WITH Daikon.exit "f" (Daikon.int "res" res) [Daikon.int "x" x] LEVEL TRACE;
  res

let () =
  let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] in
  let l = List.map f l in
  List.iter (Printf.printf "%d\n") l
```

6.4 Review

Once the program has been run, files "`daikon.decls`" and "`daikon.dtrace`" have been generated and can be passed to Daikon for analysis. The result of the analysis is shown at code sample 9. A total of six invariants have been computed. The first three link the function result with its paremeter:

- the `x` parameter is not modified by the function;
- the result of the function is either `0`, or `1`;
- the result of the function is always below or equal to `x`.

The last three link the `tmp` variable with the function parameter:

- `tmp` modulo `x` is always equal to `0`;
- `tmp` is `x` squared;
- `tmp` is greater than or equal to `x`.

Code sample 9 Result of Daikon analysis.

```
================================================================================
f:::ENTER
================================================================================
f:::EXIT1
"x" == orig("x")
"res" one of { 0, 1 }
"res" <= "x"
================================================================================
interm:::POINT
"tmp" % "x" == 0
"tmp" == "x"**2
"tmp" >= "x"
```

Chapter 7

Pajé support

7.1 Overview

Pajé[1] is a *metaformat*, or *self-describing* format[2], that was designed to enable easy and information-rich tracing of distributed systems. It defines several kinds of events that allow to indicate: a discrete event, a state change, a communication start, a communication end, *etc.* In its 1.4 version, Bolt supports the 1.2.3 version of the Pajé file format.

Pajé traces are based on the notion of *container*, which can be used to represent any entity in the system. Containers are organized in a type hierarchy, allowing for example a process to contain several threads. A container can be characterized at any time by its *state*, which can be changed according to an *event*. A container can also be characterized by some numeric *variables* whose value can change according to *events*. Finally, two containers can be associated through a *link* in order to model communications.

All Pajé data is modelled by a *type hierarchy* where containers are tree nodes while event, states, variables, and link are tree leaves. The Pajé format also uses the concept of *alias* such that types and containers are referred by their *alias* rather than by their *names*, allowing shorter traces.

7.2 Configuration

When using the Pajé format, it is necessary to produce two elements: the definition of events used through the traces, and the events actually occurring during a program run. Bolt automatically generate the definitions of events, using the set of predefined event kind from the Pajé format definition. This means that, in the current implementation, it is not possible to add new kinds of events. Nevertheless, it is still possible to add new fields to predefined events. Producing traces in the Pajé format is triggered by choosing the `paje` layout as shown by code sample 10 (old configuration format), or code sample 11 (new configuration format).

[1] The homepage of the project is located at `http://paje.sourceforge.net`.

[2] The document describing the file format is available at `http://paje.sourceforge.net/download/publication/lang-paje.pdf`.

Code sample 10 Pajé configuration (old format).

```
[]
level=trace
filter=message_paje
layout=paje
mode=direct
output=file
name=paje.trace
```

Code sample 11 Pajé configuration (new format).

```
logger "" {
  level = trace;
  filter = message_paje;
  layout = paje;
  mode = direct;
  output = file;
  name = "paje.trace";
}
```

7.3 Instrumentation

7.3.1 General sketch

In order to produce a trace containing Pajé events, it is necessary to use logging statements with the special `Paje.t` value as message, events kind and properties being passed as log properties. The complete list of event kind can be consulted in the ocamldoc-generated documentation for the `Paje` module, available in the `ocamldoc` directory after execution of `make doc`. Code sample 12 shows three events: one recording a discrete phenomenon, and two indicating changes of an element state.

Note: the Pajé layout ignores the `EXCEPTION` part of the log statements.

7.3.2 Defining types

Before using a type, it is necessary to define it through a dedicated event. Types can be defined for any Pajé entity (that is container, state, event, variable, or link). In every function, the `properties` parameter allows one to specify properties that are not defined by the file format; all properties defined by the file format being passed through named parameters. The `name` and `alias` types are bare synonyms for `string`.

`define_container_type : name:name -> ?typ:string -> ?alias:alias -> properties -> properties` defines a new type for containers, with an optional parent type.

`define_state_type : name:name -> typ:string -> ?alias:alias -> properties -> properties` defines a new type for states.

Code sample 12 Pajé example.

```
LOG Paje.t
  PROPERTIES Paje.new_event ~typ:"mail" ~container:"cnt" ~value:msg []
  LEVEL TRACE;

(...)

LOG Paje.t
  PROPERTIES Paje.set_state ~typ:"state" ~container:"cnt" ~value:"computing" []
  LEVEL TRACE;

(...)

LOG Paje.t
  PROPERTIES Paje.set_state ~typ:"state" ~container:"cnt" ~value:"waiting" []
  LEVEL TRACE;
```

`define_event_type : name:name -> typ:string -> ?alias:alias -> properties -> propertie` defines a new type for events.

`define_variable_type : name:name -> typ:string -> color:color -> ?alias:alias -> properties -> properties` defines a new type for variables.

`define_link_type : name:name -> typ:string -> start_container_type:string -> end_container_type:string -> ?alias:alias -> properties -> properties` defines a new type for links.

`define_entity_value : name:name -> typ:string -> color:color -> ?alias:alias -> properties -> properties` defines a new type for entity values.

7.3.3 Managing containers

`create_container : name:name -> typ:string -> ?container:name -> ?alias:alias -> properties -> properties` is used to create new containers.

`destroy_container : name:name -> typ:string -> properties -> properties` is used to destroy existing containers.

7.3.4 Recording states

`set_state : typ:string -> container:name -> value:string -> properties -> properties` changes the state of a container to a given value.

`push_state : typ:string -> container:name -> value:string -> properties -> properties` pushes the current value of a container to its own state, before changing to the given value.

`pop_state : typ:string -> container:name -> properties -> properties` changes the value of a container, by popping a previously pushed value.

`reset_state : typ:string -> container:name -> properties -> properties` clears previously saved states of a container.

7.3.5 Recording events

`new_event : typ:string -> container:name -> value:string -> properties -> properties` is used to indicate a discrete event.

7.3.6 Recording variables

`set_variable : typ:string -> container:name -> value:float -> properties -> properties` changes the value of a variable to the passed value.

`add_variable : typ:string -> container:name -> value:float -> properties -> properties` changes the value of a variable by increasing it by a given amount.

`sub_variable : typ:string -> container:name -> value:float -> properties -> properties` changes the value of a variable by decreasing it by a given amount.

7.3.7 Recording links

`typ:string -> container:name -> start_container:name -> value:string -> key:string -> properties -> properties` is used to indicate the beginning of a link from a container, the key/value pair being used to match with associated end of a link.

`end_link : typ:string -> container:name -> end_container:name -> value:string -> key:string -> properties -> properties` is used to indicate the end of a link from a container, the key/value pair being used to match with associated beginning of a link.

7.3.8 Functorized interface

While the Pajé format gives great flexibility, it is frequent to have a set of static types defined at program start and then used during trace generation. For this reason, Bolt provides a functorized interface that allows to define types in a module to be passed to the `Paje.Make` functor. As a result, one obtains a module akin to Pajé with the previously presented functions, except that:

- functions for defining types are missing, functor application entails automatic registration of types defined in the functor parameter;
- other functions do not accept bare strings as types, but a type from the functor parameter.

7.4 Review

The traces using the Pajé format are not easily analyzed directly by a user. It is thus necessary to use a dedicated tool such as Pajé itself, or the `ViTE`[3] trace visualizer. `ViTE` will depicted

[3] Available at http://vite.gforge.inria.fr

states through colored rectangle, discrete events through small discs, and communication events (named *links* in the Pajé format) through arrows. This visual representation is a great help for the understanding of a system with multiple communicating entities. Moreover, the `ViTE` tool is able to compute statistics about states. Figure 7.1 shows a typical `ViTe` representation of log events.

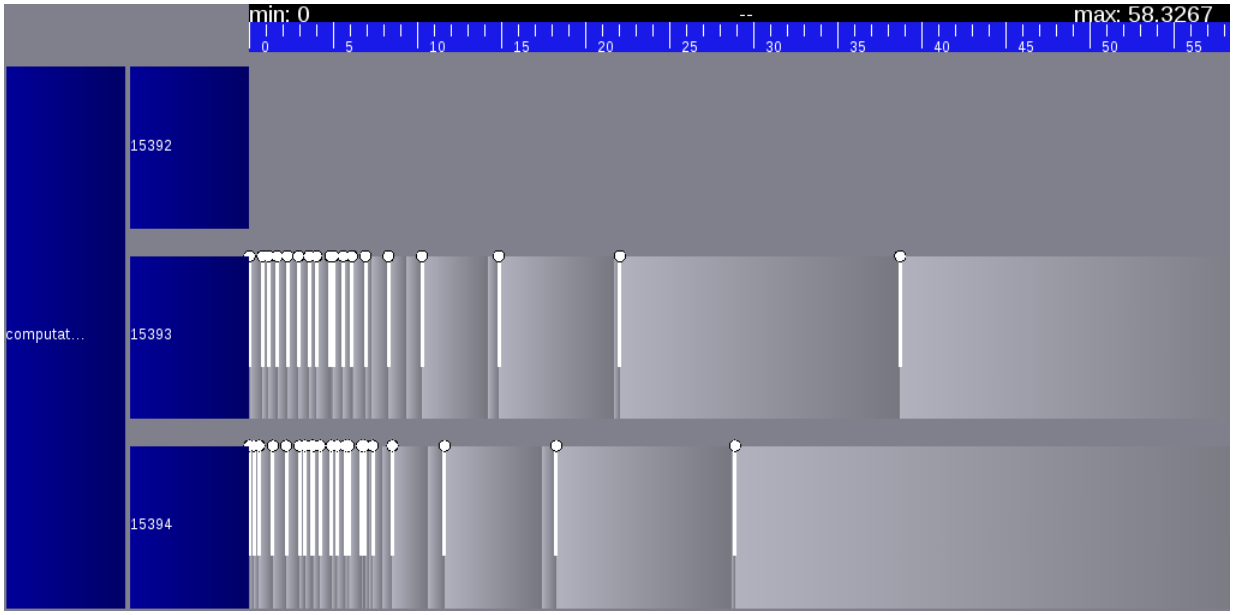

Figure 7.1: Example of ViTE visualization.

Chapter 8

Complete example

Code sample 13 shows a short program using the implicit logging feature of Bolt. The program can be compiled and executed by the `Makefile` shown by code sample 14. The `compile` target underlines that compilation should be done through the Bolt preprocessor, and that link entails references to the `unix`, and `dynlink` libraries (both being shipped with the standard OCaml distribution).

Code sample 13 Source example.

```
let funct n =
  LOG "funct(%d)" n LEVEL DEBUG;
  for i = 1 to n do
    print_endline "..."
  done

let () =
  LOG "application start" LEVEL TRACE;
  funct 3;
  funct 7;
  LOG "application end" LEVEL TRACE
```

The targets `run-old` and `run-new` of the `Makefile` show that the environment variable `BOLT_FILE` or `BOLT_CONFIG` should be set to the path of the configuration file defining the actual runtime-configuration of logging. The related configuration files are respectively represented by code samples 15 and 16. As a result of execution, a plain text file named `log` will be produced, and can be viewed using the `view` target of the Makefile. Code sample 17 shows the contents of the `log` file.

It is also possible to compile the `source.ml` file through the `ocamlbuild` tool. The most convenient way is to first define a new `bolt` tag in a `myocamlbuild.ml` plugin. This tag will add the necessary elements when compiling or linking a file using the Bolt features, as shown by code sample 18. Then, it is sufficient to use the newly-introduced tag in the `_tags` file to use Bolt, as shown by code sample 19.

Code sample 14 Makefile example.

```
DEPENDENCIES=unix.cma dynlink.cma

default: clean compile run-new view

clean:
        rm -f *.cm* log bytecode

compile:
        ocamlc -c -I +bolt bolt.cma \
            -pp 'camlp4o path/to/bolt/bolt_pp.cmo' source.ml
        ocamlc -o bytecode -I +bolt $(DEPENDENCIES) bolt.cma source.cmo

run-old:
        BOLT_FILE=config.old ./bytecode

run-new:
        BOLT_CONFIG=config.new ./bytecode

view:
        cat log
```

Code sample 15 Configuration file (old format).

```
level=trace
filter=all
layout=default
mode=direct
output=file
name=log
```

Code sample 16 Configuration file (new format).

```
logger "" {
  level = trace;
  filter = all;
  layout = default;
  mode = direct;
  output = file;
  name = "log";
}
```

Code sample 17 Generated log.

```
      2 [       source.ml 8    ] TRACE - application start
      4 [       source.ml 2    ] DEBUG - funct(3)
      4 [       source.ml 2    ] DEBUG - funct(7)
      4 [       source.ml 11   ] TRACE - application end
```

Code sample 18 `myocamlbuild.ml` plugin file.

```
open Ocamlbuild_plugin
open Ocamlbuild_pack

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

Code sample 19 `_tags` file.

```
<source.*>: use_unix, use_dynlink, bolt
```

Chapter 9

Customizing Bolt

It is possible to customize Bolt by defining new filters, layouts, and outputs. This is easily done by using respectively the `Bolt.Filter.register`, `Bolt.Layout.register`, and `Bolt.Output.register` functions. The following sections give examples of how this can be done. More information about the actual types and functions can be found in the `ocamldoc`-generated documentation (available in the `ocamldoc` directory, generation being triggered by the `make doc` command).

9.1 Defining a custom filter

A filter is barely a function from `Bolt.Event.t` to `bool`. It is possible to write explicitly the function, or to rely on predefined filters assembled through combinators provided by the `Bolt.Event` module. Code sample 20 shows the definition of two filters: the first one is explicitly coded and only keep events whose line number is even, while the second one is encoded through combinators and keep events with neither exception nor property.

Code sample 20 Custom filters.

```
let () =
  Bolt.Filter.register
    "even_line"
    (fun e -> (e.Bolt.Event.line mod 2) = 0)

let () =
  Bolt.Filter.register
    "no_exception_and_no_property"
    (let open Bolt.Filter in
     exception_none &&& properties_empty)
```

9.2 Defining a custom layout

A layout is a triple containing: a header (as a `string list`), a footer (as a `string list`), and a rendering function (as a function from `Bolt.Event.t` to `string`). Code sample 21 shows

the definition of a layout with empty header and footer, and a rendering function based on `Printf.sprintf`.

Code sample 21 Custom layout.

```
let () =
  Bolt.Layout.register
    "printf_layout"
    ([], (* header *)
     [], (* footer *)
     (fun e ->
       Printf.sprintf "file \"%s\" says \"%s\" with level \"%s\" (line: %d)"
         e.Bolt.Event.file
         e.Bolt.Event.message
         (Bolt.Level.to_string e.Bolt.Event.level)
         e.Bolt.Event.line))
```

9.3 Defining a custom output

An output is a function taking a `string` parameter, a `Bolt.Output.rotation` parameter, and a `Bolt.Layout.t` parameter to build an `Bolt.Output.impl` object. The semantics of the string parameter is to be defined by the output itself (for the `file` output, it is the filename of the destination). The `Bolt.Output.rotation` parameters defines when a rotation should happen (based on time and/or signal interception). Finally, the layout indicates header and footer to write at each rotation.

The `Bolt.Output.impl` object to be built by a layout should define two methods:

- `write : string -> unit` that will be called to record strings rendered through the layout;
- `close : unit` that will be called at the end of the program in order to perform clean-up operations.

Code sample 22 shows the definition of a layout using the `mail` system command to log elements.

9.4 Compiling custom elements

In order to compile source files containing the definition of custom elements, it is sufficient to add the path to the Bolt library to the compiler search path (*i.e.* through `-I +bolt`).

9.5 Using custom elements

When custom elements have been registered using the previously mentioned functions, they can be used from the configuration files or from the `Bolt.Logger.register` function. However, it is necessary for the custom elements to be registered before *any* log event concerned with these

Code sample 22 Custom output.

```
let () =
  let send_mail _ _ _ =
    object
      method write msg =
        try
         let command = Printf.sprintf "mail -s %S dest@domain.com" msg in
         ignore (Sys.command command)
        with _ -> ()
      method close = ()
    end in
  Bolt.Output.register "send_mail" send_mail
```

custom elements is built. Otherwise, elements won't be found and Bolt will resort to default values.

A good practice is to define the new filters, layouts, and outputs in modules that are not part of the application. One should not forget to pass the `-linkall` switch to the compilers when linking such modules. Another option is to avoid linking these modules with the application, and to use the `BOLT_PLUGINS` environment variable to load them. The `BOLT_PLUGINS` environment variable contains a comma-separated list of files that will be loaded through `Dynlink`.